UNIVERSITÄT PASSAU
Fakultät für Mathematik und Informatik

Constant Propagation in Explicitly Parallel Programs[1]

Jens Knoop

Fakultät für Mathematik und Informatik
Universität Passau
Innstraße 33
D–94032 Passau
Germany
knoop@fmi.uni-passau.de

MIP-9809
July 1998

[1]This is an extended version of an article going to appear in Proceedings of the 4th European Conference on Parallel Processing (Euro-Par'98), Southampton, UK, September 1 - 4, 1998.

Abstract

Constant propagation (*CP*) is a powerful, practically relevant optimization of sequential programs. However, only few approaches have been proposed aiming at making CP available for parallel programs. In fact, because of the computational complexity paraphrased by the catch-phrase "state explosion problem," the successful transfer of sequential techniques is currently essentially restricted to bitvector-based optimizations. Because of their structural simplicity they can be enhanced to parallel programs at almost no costs on the implementation and computation side. CP, however, is beyond this class. Here, however, we present a powerful algorithm for constant propagation in parallel programs, which is based on an extension of the framework underlying the successful transfer of bitvector problems, and which can be implemented as *easily* and as *efficiently* as its sequential counterpart for *simple constants* computed by state-of-the-art sequential optimizers.

Keywords

(Explicit) parallelism, interleaving semantics, shared memory, synchronization, optimizing parallel compilers, data-flow analysis, program optimization, constant propagation, simple constants.

Contents

1 Motivation

In comparison to automatic parallelization (cf. [24, 25]), optimization of *parallel* programs attracted so far only little attention. An important reason may be that straightforward adaptions of sequential techniques typically fail (cf. [17]), and that the costs of rigorous, correct adaptions are unacceptably high because of the combinatorial explosion of the number of interleavings manifesting the possible executions of a parallel program. However, the example of unidirectional bitvector (UBv) problems (cf. [6]) shows that there are exceptions: UBv-problems can be solved as *easily* and as *efficiently* as for sequential programs (cf. [14]). This opened the way for the successful transfer of the classical bitvector-based optimizations to the parallel setting at almost no costs on the implementation and computation side. In [9] and [13] this has been demonstrated for *partial dead-code elimination* (cf. [12]), and *partial redundancy elimination* (cf. [18, 11]).

Constant propagation (*CP*) (cf. [7, 8, 19]), however, is beyond bitvector-based optimization. It is a powerful and widely used optimization of sequential programs, which improves performance by replacing terms, which can be determined at compile-time to evaluate to a unique constant value at run-time by that value. In this article we develop an algorithm for constant propagation in parallel programs, for short an algorithm for *parallel constant propagation* (*PCP*), which is based on a slight extension of the framework of [14]. Like bitvector-based optimizations this algorithm for parallel constant propagation can be implemented as *easily* and as *efficiently* as its sequential counterpart. For PCP, this is the algorithm for *simple constants* (*SCs*) (cf. [7, 8, 19]), which are computed by state-of-the-art optimizers of sequential programs.

The power of this algorithm is demonstrated in the example of Figure 1, where it is unique to achieve the optimization of Figure 2. It detects that y has the value 15 before entering the parallel statement, and that this value is maintained by it. Similarly, it detects that b, c, and e have the constant values 3, 5, and 2 before and after leaving the parallel assignment. Moreover, it detects that independently of the interleaving sequence taken at run-time d and x are assigned the constant values 3 and 17 inside the parallel assignment. Together this allows our algorithm not only to detect the constancy of the right-hand side terms at the edges **48**, **56**, **57**, **60**, and **61** after the parallel statement, but also the constancy of the right-hand side terms at the edges **22**, **37**, and **45** inside the parallel statement.

Importantly, this result is obtained without any performance penalty in comparison to sequential CP. Fundamental in order to achieve this is to conceptually decompose PCP in a way such that it behaves differently on sequential and parallel program parts. Intuitively, on sequential program parts our PCP-algorithm coincides with its sequential counterpart for SCs, on parallel program parts it computes a safe approximation of SCs, which is tailored for being computable under side-conditions that allow us as for UBv-problems to capture the phenomena of *interference* and *synchronization* of parallel components completely without having to consider interleavings at all.

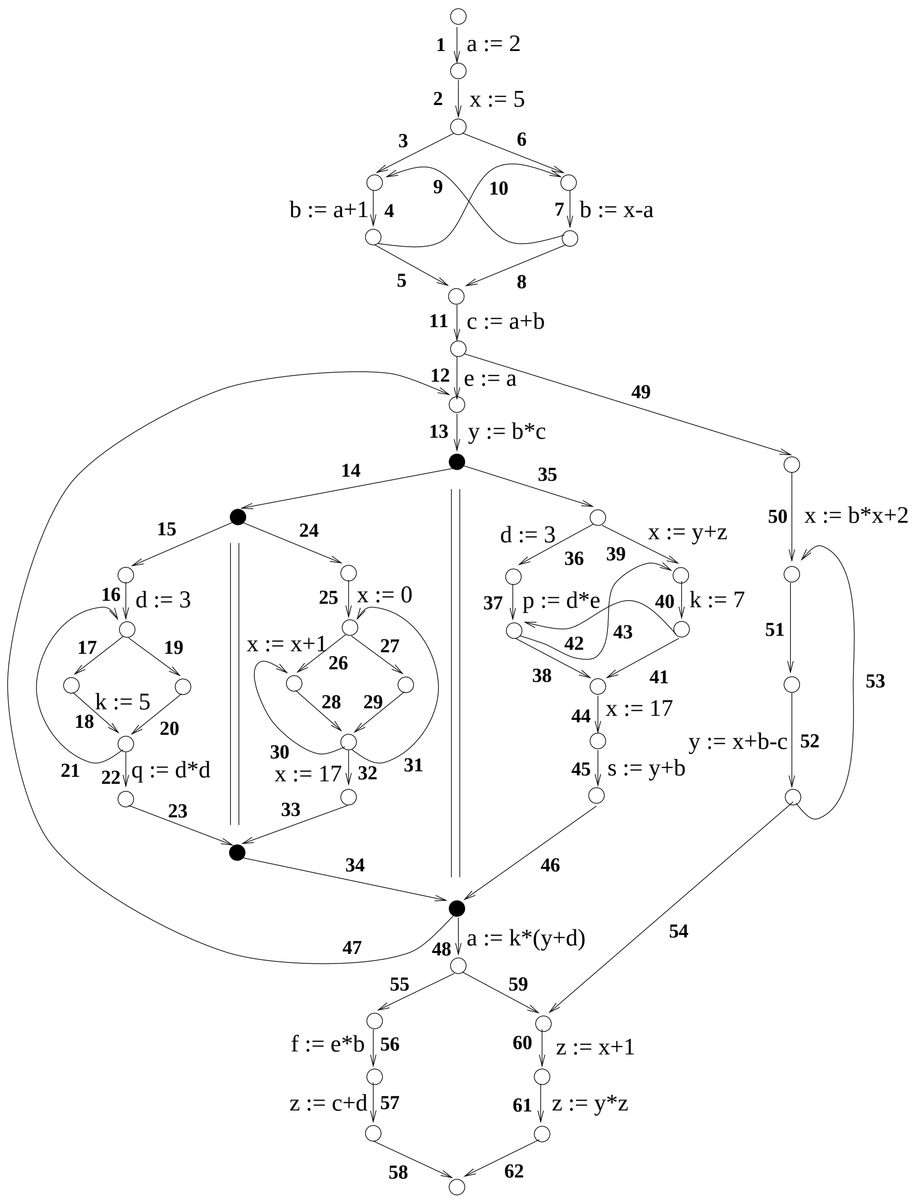

Figure 1: Motivating example.

Summarizing, the central contributions of the article are as follows: (1) Extending the framework of [14] to a specific class of non-bitvector problems. (2) Developing based on this extension an algorithm for parallel constant propagation, which works for reducible and irreducible control flow, and is as efficient and can be implemented as easily as its sequential counterpart for SCs.

Related Work

To the best of the author's knowledge the PCP-algorithm proposed here is together with an algorithm that was recently proposed by Lee, Midkiff, and Padua [15], the

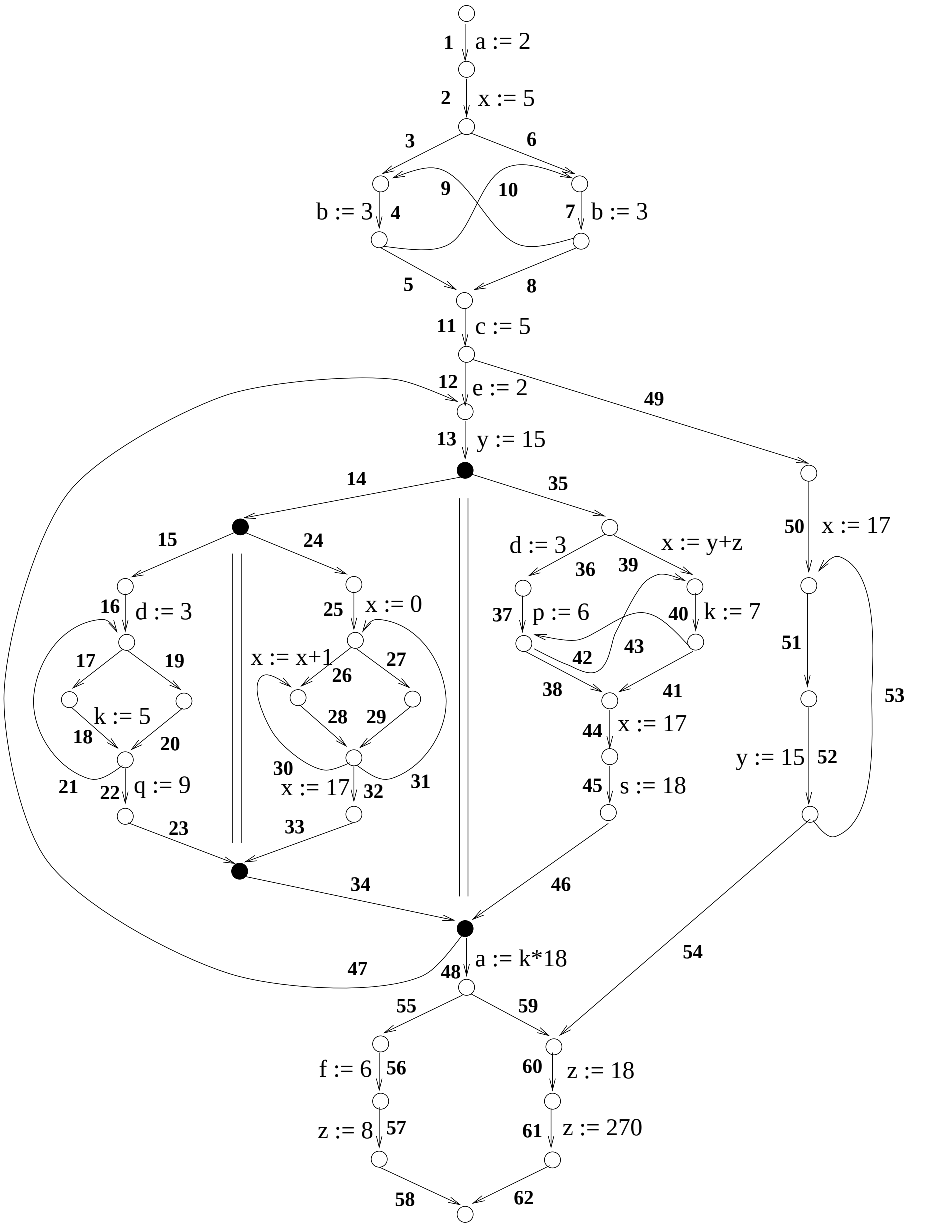

Figure 2: The power of parallel constant propagation.

only algorithm for constant propagation in parallel programs. Both algorithms, however, are incomparable to each other as they are developed for different settings. In the approach of [15] parallel statements must not occur inside of loops, while on the other hand explicit synchronization via set and wait statements is considered. In the approach here, we do not consider explicit synchronization statements, but there are no restrictions on the structure of the program. In particular, parallel statements may occur inside of loops as shown in Figures 1 and 2. Moreover, both approaches differ in the program representation they are working on. While the algorithm of [15] works on the concurrent static single assignment form, a specifically designed new parallel variant of the static single assignment (SSA) form known from sequential programs (cf. [1, 4, 20]), the algorithm here works on an ordinary (parallel) flow graph representation of a program.

2 Preliminaries

We develop our PCP-algorithm for the parallel setup, which has been introduced and presented in detail in [14]. As in [14] we consider parallel imperative programs with shared memory and interleaving semantics. Parallelism is syntactically expressed by means of a `par` statement. This introduces the phenomena of interference and synchronization into the setting, and we thus do not consider special synchronization or exception statements, which allows us to focus on the central features of our approach. However, it is not limited to this setting. E.g., dynamic process creation can be integrated along the lines of [2, 23]. As usual, we assume that there are neither jumps leading into a component of a parallel statement from outside nor vice versa.

2.1 Parallel Flow Graphs

Similar to [5, 22], we represent a parallel program by a *parallel flow graph* $G = (N, E, \mathtt{s}, \mathtt{e})$ with node set N, edge set E, and a unique *start node* $\mathtt{s}$ and *end node* $\mathtt{e}$, which are assumed to be free of incoming and outgoing edges, respectively. Following [10] we consider edge-labelled flow graphs. Thus, edges $e \in E$ represent both the elementary statements, i.e., the empty statement `skip`and assignments of the form $x := t$ with x a variable and t a term inductively built from variables, constants, and operators, and the nondeterministic branching structure of the progam under consideration, while nodes $n \in N$ represent program points. As shown in Figure 1, parallel statements are represented by subgraphs which are encapsulated by a `ParBegin` node and a `ParEnd` node. For clarity, these nodes are blackened in the figures throughout this article, and additionally, the components of subgraphs representing a parallel statement are separated by two parallels.

Conventions. Let G be a parallel flow graph. Then we denote by $\mathcal{G}_\mathcal{P}(G)$ the set of subgraphs of G representing a parallel statement, and by $\mathcal{G}_\mathcal{C}(G')$ the set of component graphs of $G' \in \mathcal{G}_\mathcal{P}(G)$. Additionally, we use $\mathcal{G}_\mathcal{C}(G)$ as an abbreviation of $\bigcup \{ \mathcal{G}_\mathcal{C}(G') \mid G' \in \mathcal{G}_\mathcal{P}(G) \}$. Moreover, we assume that all graphs of $\mathcal{G}_\mathcal{P}(G)$ and $\mathcal{G}_\mathcal{C}(G)$ are well-formed, i.e., they have a unique start node and end node and each of their nodes lies on a path connecting them. By $\mathcal{G}_\mathcal{P}^{max}(G)$ we denote the set of maximal parallel subgraphs of G. Intuitively, they represent the outermost parallel statements of the underlying program:

$$\mathcal{G}_\mathcal{P}^{max}(G) =_{df} \{ G' \in \mathcal{G}_\mathcal{P}(G) \mid \forall\, G'' \in \mathcal{G}_\mathcal{P}(G).\ G' \subseteq G'' \Rightarrow G' = G'' \}$$

The sets of start nodes and end nodes of graphs in $\mathcal{G}_\mathcal{P}(G)$, i.e., the sets of nodes annotated by `ParBegin` and `ParEnd`, are abbreviated by N_N and N_X, respectively.

Moreover, by $source(e)$ and $dest(e)$ we denote the start node and end node of an edge $e \in E$. Finally, $E_{seq} \subseteq E$ denotes the set of edges of E occurring in a sequential part of G, and conversely, $E_{par} =_{df} E \setminus E_{seq}$ denotes the set of edges occurring in a parallel part of G.

Interleaving Predecessors and Parallel Paths. Intuitively, we consider as predecessors of a node $n \in N$ the statements which can be executed immediately before reaching (or equivalently: before leaving) n. As we are considering edge-labelled graphs, the predecessors of a node $n \in N$ are thus given by the set of edges, whose statements can be executed immediately before leaving n, i.e., before executing the statement of an edge leaving n. In a sequential program part, the set of predecessors is thus simply given by the set of incoming edges of n, called here the set of *static* predecessors of n. In a parallel program part, however, the interleaving of parallel components must also be taken into account: for a program point n lying in a component of a `par` statement s, this set must be enriched by the assignment of any edge of another component of s. We denote these latter edges as *interleaving predecessors* of n and denote them by $IntPred(n)$. In the flow graph of Figure 1, the set of static predecessors of node $source(\mathbf{22})$ e.g. is given by $\{\mathbf{18}, \mathbf{20}\}$, whereas its set of interleaving predecessors is given by $\{\mathbf{24}, \ldots, \mathbf{33}, \mathbf{35}, \ldots, \mathbf{46}\}$.

In essence, a path of a parallel program is a specific interleaving sequence. Formally, this can be defined elegantly via a translation of the parallel program to a (much larger) nondeterministic program, in which every interleaving of the underlying parallel program is explicitly represented: an edge sequence of a parallel program is a *parallel path* if and only if it is a path in the corresponding "product" program. We denote the set of all parallel paths from m to n by $\mathbf{PP}[m, n]$. Moreover, we assume that every node n lies on a parallel path from $\mathbf{s}$ to $\mathbf{e}$.

2.2 Semantics of Terms

As mentioned before, we consider terms $t \in \mathbf{T}$, which are inductively built from variables $v \in \mathbf{V}$, constants $c \in \mathbf{C}$, and operators $op \in \mathbf{Op}$ of arity $r \geq 1$. Their *semantics* is induced by an *interpretation* $I = (\mathbf{D}' \cup \{\perp, \top\}, I_0)$, where $\mathbf{D}'$ denotes a non-empty data domain, $\perp$ and $\top$ two new data not in $\mathbf{D}'$, and I_0 a function mapping every constant $c \in \mathbf{C}$ to a datum $I_0(c) \in \mathbf{D}'$, and every r–ary operator $op \in \mathbf{Op}$ to a total, strict function $I_0(op) : \mathbf{D}^r \to \mathbf{D}$, $\mathbf{D} =_{df} \mathbf{D}' \cup \{\perp, \top\}$ (i.e., $I_0(op)(d_1, \ldots, d_r) = \perp$, whenever there is a j, $1 \leq j \leq r$, with $d_j = \perp$). $\Sigma =_{df} \{\sigma \mid \sigma : \mathbf{V} \to \mathbf{D}\}$ denotes the set of *states*, and $\sigma_\perp$ the distinct *start state* assigning $\perp$ to all variables $v \in \mathbf{V}$. This choice reflects that we do not assume anything about the context of a program being optimized. The *semantics* of a term $t \in \mathbf{T}$ is then given by the evaluation function $\mathcal{E} : \mathbf{T} \to (\Sigma \to \mathbf{D})$, which is inductively defined by:

$$\forall t \in \mathbf{T}\ \forall \sigma \in \Sigma.\ \mathcal{E}(t)(\sigma) =_{df} \begin{cases} \sigma(x) & \text{if } t = x \in \mathbf{V} \\ I_0(c) & \text{if } t = c \in \mathbf{C} \\ I_0(op)(\mathcal{E}(t_1)(\sigma), \ldots, \mathcal{E}(t_r)(\sigma)) & \text{if } t = op(t_1, \ldots, t_r) \end{cases}$$

In the following we assume $\mathbf{D}' \subseteq \mathbf{T}$, i.e., the set of data $\mathbf{D}'$ is identified with the set of constants $\mathbf{C}$. Introducing the *state transformation function* $\theta_\iota : \Sigma \to \Sigma$, $\iota \equiv x := t$, which is defined by

$$\forall\, \sigma \in \Sigma\ \forall\, y \in \mathbf{V}.\ \theta_\iota(\sigma)(y) =_{df} \begin{cases} \mathcal{E}(t)(\sigma) & \text{if } y = x \\ \sigma(y) & \text{otherwise} \end{cases}$$

we can define the set of states Σ_n, which are possible at some program point $n \in N$. In this definition θ_p denotes the straightforward extension of the state transformation functions to parallel paths:

$$\Sigma_n =_{df} \{\, \theta_p(\sigma_\perp) \mid p \in \mathbf{PP}[\mathbf{s}, n]\}$$

This allows us to define the set $\mathcal{C}_n$ of all terms yielding a unique constant value at run-time at some program point $n \in N$:

$$\mathcal{C}_n =_{df} \bigcup_{d \in \mathbf{D}'} \mathcal{C}_n^d$$

Here $\mathcal{C}_n^d$, $d \in \mathbf{D}'$, denotes the set of terms evaluating to d at point n. Formally, this set is defined by

$$\mathcal{C}_n^d =_{df} \{t \in \mathbf{T} \mid \forall\, \sigma \in \Sigma_n.\ \mathcal{E}(t)(\sigma) = d\}$$

Unfortunately, even for sequential programs the sets $\mathcal{C}_n$, $n \in N$, are in general not decidable (cf. [19]). Simple constants recalled next are an efficiently decidable subset of $\mathcal{C}_n$, which are computed by state-of-the-art sequential optimizers, i.e., by optimizers of sequential programs.

3 Simple Constants

Intuitively, a term is a (sequential) *simple constant* (*SC*) (cf. [8]), if it is a program constant or all its operands are simple constants. In the example of Figure 3, the right-hand side terms of the assignments to a, b, c and d are SCs, whereas the one of the assignment to e, though it is a constant of value 5, and the one of the assignment to f, which in fact is not a constant, are not. Data-flow analysis provides the means for computing SCs.

3.1 Data-flow Analysis

In essence, *data-flow analysis* (*DFA*) provides information about the program states, which may occur at a specific program point at run-time. Theoretically well-founded are DFAs based on *abstract interpretation* (cf. [3, 16]). Usually, the abstract semantics is specified by a *local semantic functional* $[\![\ ]\!] : E \to (\mathcal{L} \to \mathcal{L})$ giving abstract meaning to every program statement (here: every edge $e \in E$ of a sequential or parallel flow graph G) in terms of a transformation function on a complete lattice $(\mathcal{L}, \sqcap, \sqsubseteq, \perp, \top)$. Its elements express the DFA-information of interest.

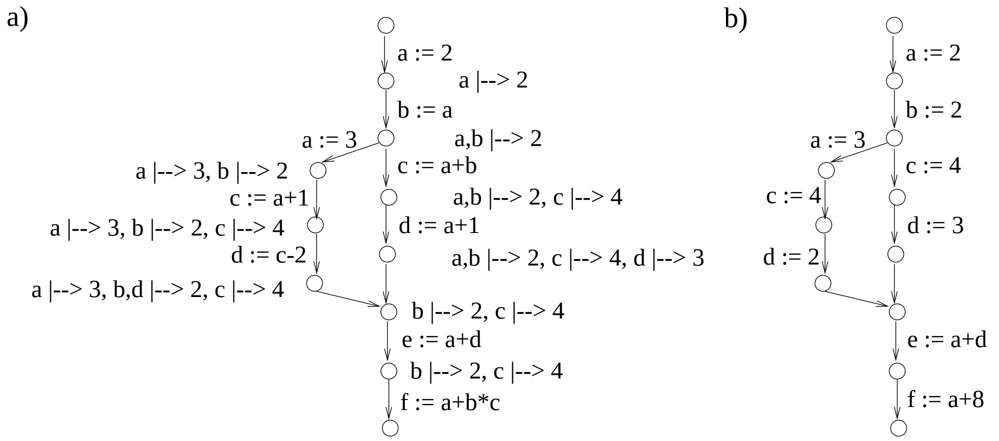

Figure 3: Simple constants.

Local semantic functionals can easily be extended to capture finite paths, which is the key to the *meet over all paths* (*MOP*) approach in the sense of Kam and Ullman [7]. It yields the intuitively desired solution of a DFA-problem because it directly mimics possible program executions: it "meets" all informations belonging to a program path leading from $\mathbf{s}$ to a program point $n \in N$.

$$\textbf{The } \textit{MOP}\textbf{-Solution: } \forall\, l_0 \in \mathcal{L}.\ \textit{MOP}(n)(l_0) = \bigsqcap \{\, [\![\, p \,]\!](l_0) \mid p \in \mathbf{P}[\mathbf{s}, n] \,\}$$

Unfortunately, this is in general not effective, which leads us to the *maximal fixed point* (*MFP*) approach in the sense of Kam and Ullman [7]. Intuitively, it approximates the *greatest* solution of a system of equations imposing consistency constraints on an annotation of the program with DFA-information with respect to some start information $l_0 \in \mathcal{L}$:

$$\mathbf{mfp}(n) = \begin{cases} l_0 & \textit{if } n = \mathbf{s} \\ \bigsqcap \{\, [\![\, (m,n) \,]\!](\mathbf{mfp}(m)) \mid (m,n) \in E \,\} & \textit{otherwise} \end{cases}$$

The greatest solution of this equation system, which we denote by $\mathtt{mfp}_{l_0}$, can effectively be computed, if the semantic functions $[\![\, e \,]\!]$, $e \in E$, are monotonic, and $\mathcal{L}$ is of finite height. It defines the solution of the *MFP*-approach:

$$\textbf{The } \textit{MFP}\textbf{-Solution: } \forall\, l_0 \in \mathcal{L}\ \forall\, n \in N.\ \textit{MFP}(n)(l_0) =_{df} \mathtt{mfp}_{l_0}(n)$$

The following theorem, which relates both solutions, is central. It gives sufficient conditions for the safety (correctness) and even precision of the *MFP*-solution with respect to the *MOP*-solution (cf. [7, 8]):

Theorem 3.1 (Safety and Coincidence Theorem)

1. Safety: $\textit{MFP}(n) \sqsubseteq \textit{MOP}(n)$, *if all* $[\![\, e \,]\!]$, $e \in E$, *are monotonic.*
2. Coincidence: $\textit{MFP}(n) = \textit{MOP}(n)$, *if all* $[\![\, e \,]\!]$, $e \in E$, *are distributive.*

3.2 Computing Simple Constants

Considering $\mathbf{D}$ a flat lattice as illustrated in Figure 4(a), the set of states Σ together with the pointwise ordering is a complete lattice, too. Having this in mind, the computation of SCs relies on the local semantic functional $[\![\ ]\!]_{sc} : E \rightarrow (\Sigma \rightarrow \Sigma)$ defined by $[\![e]\!]_{sc} =_{df} \theta_e$ for each edge $e \in E$, and the start state $\sigma_\perp$. Since the number of variables occurring in a program is finite, we have (cf. [7]):

Lemma 3.2 *Σ is of finite height, and all functions $[\![e]\!]_{sc}$, $e \in E$, are monotonic.*

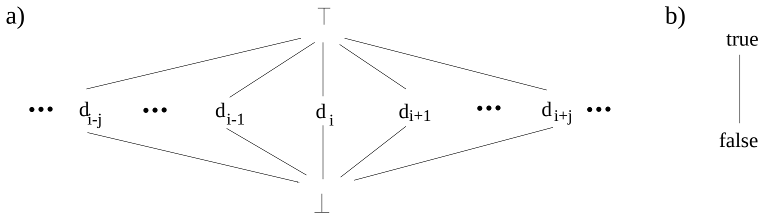

Figure 4: Flat (data) lattices.

As proved in [7] the functions $[\![e]\!]_{sc}$, $e \in E$, are in general not distributive. However, as pointed out in the previous section monotonicity is sufficient for the effective computability of the *MFP*-solution of the SC-problem, which we denote by $MFP^{sc}_{\sigma_\perp}$. It induces the formal definition of (sequential) *simple constants*.

$$\mathcal{C}^{sc}_n =_{df} \mathcal{C}^{sc,mfp}_n =_{df} \{ t \in \mathbf{T} \mid \exists\, d \in \mathbf{D}'.\ \mathcal{E}(t)(MFP^{sc}_{\sigma_\perp}(n)) = d \}$$

Defining dually the set of constants

$$\mathcal{C}^{sc,mop}_n =_{df} \{ t \in \mathbf{T} \mid \exists\, d \in \mathbf{D}'.\ \mathcal{E}(t)(MOP^{sc}_{\sigma_\perp}(n)) = d \}$$

induced by the $MOP^{sc}_{\sigma_\perp}$-solution, we obtain by means of Theorem 3.1(1) (cf. [7]):

Theorem 3.3 (SC-Correctness) $\forall\, n \in N.\ \mathcal{C}_n \supseteq \mathcal{C}^{sc,mop}_n \supseteq \mathcal{C}^{sc,mfp}_n = \mathcal{C}^{sc}_n$.

4 Parallel Constant Progagation

In a parallel program, the validity of the SC-property for a term which occurs in a parallel statement depends mutually on the validity of this property for other terms; verifying it cannot be separated from the interleaving sequences. The costs of investigating them explicitly, however, are prohibitive because of the well-known *state explosion problem*. We therefore introduce a safe approximation of SCs, called *strong constants* (*StCs*). Like the solution of a unidirectional bitvector (UBv) problem they can be computed without having to consider interleavings at all. The computation of StCs is based on the local semantic functional $[\![\ ]\!]_{stc} : E \rightarrow (\Sigma \rightarrow \Sigma)$, which is defined by $[\![e]\!]_{stc}(\sigma) =_{df} \theta^{stc}_e(\sigma)$ for every edge $e \in E$ and state $\sigma \in \Sigma$. In this definition θ_e^{stc} denotes the state transformation function of Section 2.2, where, however, the evaluation function $\mathcal{E}$ is replaced by the simpler $\mathcal{E}_{stc}$ defined as follows:

$$\forall\, t \in \mathbf{T}\ \forall\, \sigma \in \Sigma.\ \mathcal{E}_{stc}(t)(\sigma) =_{df} \begin{cases} I_0(c) & \text{if } t = c \in \mathbf{C} \\ \perp & \text{otherwise} \end{cases}$$

The set of strong constants at n, denoted by $\mathcal{C}_n^{stc}$, is then defined analogously to the corresponding set of SCs. Denoting by $\mathcal{F}_{\mathbf{D}} =_{df} \{\, Cst_d \mid d \in \mathbf{D}\} \cup \{Id_{\mathbf{D}}\}$ the set of constant functions Cst_d on $\mathbf{D}$, $d \in \mathbf{D}$, enlarged by the identity $Id_{\mathbf{D}}$ on $\mathbf{D}$, we obtain:

Lemma 4.1 $\forall\, e \in E\ \forall\, v \in \mathbf{V}.\ [\![\, e \,]\!]_{stc}|_v \in \mathcal{F}_{\mathbf{D}} \backslash \{Cst_\top\}$, *where* $|_v$ *denotes the restriction of* $[\![\, e \,]\!]_{stc}$ *to* v.

Intuitively, Lemma 4.1 means that (1) all variables are "decoupled": each variable in the domain of a state $\sigma \in \Sigma$ can be considered like an (independent) slot of a bitvector. (2), the transformation function relevant for a variable is either a constant function or the identity on $\mathbf{D}$. This is quite similar to UBv-problems. There, for every bit (slot) of the bitvector the set of data is given by the flat lattice $\mathcal{B}$ of Boolean truth values as illustrated in Figure 4(b), and the transformation functions relevant for a slot are the two constant functions and the identity on $\mathcal{B}$. However, there is an important difference, too. $\mathcal{F}_{\mathbf{D}}$ is not closed under pointwise meet: consider the pointwise meet of $Id_{\mathbf{D}}$ and Cst_d for some $d \in \mathbf{D}$. It is given by the "peak"-function p_d with $p_d(x) = d$, if $x = d$, and $\perp$ otherwise. These two facts together with an extension allowing us to mimic the effect of pointwise meet in a setting with only constant functions and the identity are the key for extending the framework of [14], which focuses on the scenario of UBv-problems, to the scenario of StC-like problems.

4.1 Parallel Data-flow Analysis

As in the sequential case, the abstract semantics of a parallel flow graph is specified by a local semantic functional $[\![\ ]\!] : E \to (\mathcal{L} \to \mathcal{L})$, which gives meaning to its statements in terms of functions on an (arbitrary) complete lattice $\mathcal{L}$. In analogy to their sequential counterparts, these functionals can easily be extended to capture finite *parallel paths*. Hence, the definition of the parallel variant of the *MOP*-approach, the *PMOP*-approach and its solution, is obvious:

The ***PMOP*****-Solution:** $\forall\, l_0 \in \mathcal{L}.\ PMOP(n)(l_0) = \bigsqcap \{\, [\![\, p \,]\!](l_0) \mid p \in \mathbf{PP}[\mathbf{s}, n]\,\}$

The point of this section now is to show that as for UBv-problems also for the structurally more complex StC-like problems the *PMOP*-solution can efficiently be computed by means of a fixed point computation.

StC-like problems. As UBv-problems, StC-like problems are characterized by the simplicity of their local semantic functional $[\![\ ]\!] : E \to (I\!\!D \to I\!\!D)$. It specifies the effect of an edge e on a single component of the problem under consideration (considering StCs, for each variable $v \in \mathbf{V}$), where $I\!\!D$ is a flat lattice of some underlying (finite) set $I\!D'$ enlarged by two new elements $\perp$ and $\top$ (cf. Figure 4(a)), and where every function $[\![e]\!]$, $e \in E$, is an element of $\mathcal{F}_{I\!D} =_{df} \{ Cst_d \mid d \in I\!D \} \cup \{ Id_{I\!D} \}$. As shown before, $\mathcal{F}_{I\!D}$ is not closed under pointwise meet. The possibly resulting peak-functions, however, are essential for this problem type in order to be always able to model the effect when control flows together. Fortunately, this can be mimiced in a setting, where all semantic functions are constant ones (or the identity). To this end we extend $I\!D$ to a lattice $I\!D_X$ by inserting a second layer as shown in Figure 5, considering then the set of functions $\mathcal{F}_{I\!D_X} =_{df} (\mathcal{F}_{I\!D} \backslash \{ Id_{I\!D} \}) \cup \{ Id_{I\!D_X} \} \cup \{ Cst_{d^p} \mid d \in I\!D \}$. The functions Cst_{d^p}, $d \in I\!D'$, are constant functions like their respective counterparts Cst_d, however, they play the role of the peak-functions p_d, and in fact, after the termination of the analysis, they will be interpreted as peak-functions.

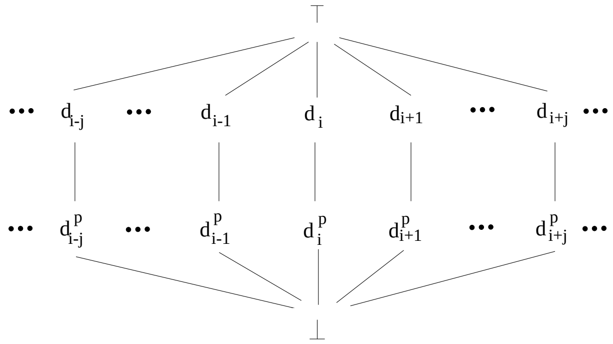

Figure 5: Extended lattice.

Intuitively, considering StCs and synchronization, the "doubling" of functions allows us to distinguish between components of a parallel statement assigning a variable under consideration a unique constant value not modified afterwards along *all* paths (Cst_d), and those doing this along *some* paths (Cst_{d^p}). While for the former components the variable is a total StC (or shorter: an StC) after their termination (as the property holds for all paths), it is a *partial* StC for the latter ones (as the property holds only for some paths). Keeping this separately in our data domain, gives us the handle for treating synchronization and interference precisely. E.g., in the program of Figure 1, d is a total StC for the parallel component starting with edge **15**, and a partial one for the component starting with edge **35**. Hence, d is an StC of value 3 after termination of the complete parallel statement as none of the parallel "relatives" destroys this property established by the left-most component because it is a partial StC of value 3, too, for the right-most component, and it is transparent for the middle one. On the other hand, d would not be an StC after the termination of the parallel statement, if, let's say, d would be a partial StC of value 5 of the right-most component.

Obviously, all functions of $\mathcal{F}_{I\!D_X}$ are distributive. Moreover, together with the orderings $\sqsubseteq_{seq}$ and $\sqsubseteq_{par}$ displayed in Figure 6, they form two complete lattices with least element $Cst_\perp$ and greatest element $Cst_\top$, and least element $Cst_\perp$ and greatest element $Id_{I\!D_X}$, respectively. Moreover, $\mathcal{F}_{I\!D_X}$ is closed under function composition (for both orderings). Intuitively, the "meet" operation with respect to $\sqsubseteq_{par}$ models

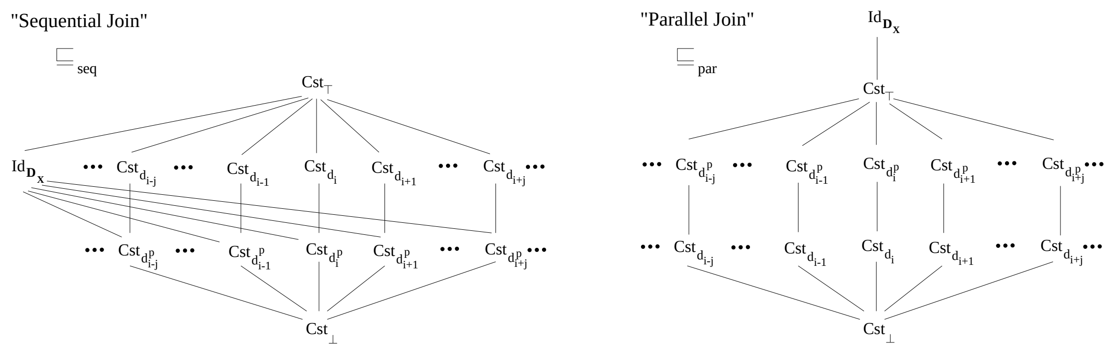

Figure 6: The function lattices for “sequential” and “parallel join”.

the merge of information in “parallel” join nodes, i.e., end nodes of parallel statements, which requires that all their parallel components terminated. Conversely, the “meet” operation modelling the merge of information at points, where sequential control flows together, which actually would be given by the pointwise (indicated below by the index “pw”) meet-operation on functions of $\mathcal{F}_{I\!D}$, is here equivalently modelled by the “meet” operation with respect to $\sqsubseteq_{seq}$. In the following we drop this index. Hence, $\sqcap$ and $\sqsubseteq$ expand to $\sqcap_{seq}$ and $\sqsubseteq_{seq}$, respectively. Recalling Lemma 4.1 and identifying the functions Cst_{d^p}, $d \in I\!D'$, with the peak-functions p_d, we obtain for every sequential flow graph G and start information of $\mathcal{F}$:

Lemma 4.2 $\forall\, n \in N.\ MOP_{pw}(n) = MOP(n) = MFP(n)$

This lemma together with Main Lemma 4.3, of which in [14] its variant for bitvector problems has been considered, is the key for the efficient computation of the effects of “interference” and “synchronization” for StC-like problems.

Lemma 4.3 (Main-Lemma)
Let $f_i : \mathcal{F}_{I\!D_X} \to \mathcal{F}_{I\!D_X},\ 1 \leq i \leq q,\ q \in I\!N$, *be functions of* $\mathcal{F}_{I\!D_X}$. *Then:* $\exists\, k \in \{1, ..., q\}.\ f_q \circ ... \circ f_2 \circ f_1 = f_k \ \wedge\ \forall\, j \in \{k+1, ..., q\}.\ f_j = Id_{I\!D_X}$.

Interference. The relevance of Main Lemma 4.3 for StC-like problems is the same as for UBv-problems. It restricts the means of possible interference within a parallel program: each possible interference is due to a single statement within a parallel component, i.e., due to an interleaving predecessor of a node n. Combining this observation with the fact that for $e \in IntPred(n)$, there exists a parallel path leading to n whose last step requires the execution of e, we obtain that the potential of interference, which in general would be given in terms of paths, is fully characterized by the set of its interleaving predecessors $IntPred(n)$. In fact, considering the computation of universal properties which are described by maximal fixed points (the computation of minimal fixed points requires the dual argument), the obvious existence of a path to n that does not require the execution of any statement of $IntPred(n)$ implies that the only effect of interference is “destruction.” In the bitvector situation considered in [14], this observation is boiled down to a predicate *NonDestruct* inducing the constant function "true" or "false" indicating the presence or absence of an interleaving predecessor destroying the property under consideration. Similarly, we here define for every node $n \in N$ the function

$$InterferenceEff(n) =_{df} \bigsqcap \{ [\![e]\!] \mid e \in IntPred(n) \wedge [\![e]\!] \neq Id_{I\!D} \}$$

These constant functions $InterferenceEff(n)$, $n \in N$, are precomputable, and suffice for modelling interference (cf. [14]). In fact, similar to [14] they can be computed as a by-product of the hierarchical procedure $\mathcal{A}$ described next.

Synchronization. Leaving a parallel statement requires the termination of all its parallel components. As for UBv-problems, the information required to model this effect can hierarchically be computed by means of an algorithm, which only considers purely sequential programs. The central idea coincides with that of interprocedural DFA (cf. [21]): one needs to compute the effect of complete subgraphs, in this case of complete parallel components. This information is computed in an "innermost" fashion and then propagated to the next surrounding parallel statement. The complete three-step procedure $\mathcal{A}$ is as follows:

1. Terminate, if $\hat{G}$ does not contain any parallel statement. Otherwise, select successively all maximal flow graphs G' occurring in a graph of $\mathcal{G}_{\mathcal{P}}(\hat{G})$ that do not contain a nested parallel statement, and determine the effect $[\![G']\!]$ of this (purely sequential) graph according to the equational system

$$[\![n]\!] = \begin{cases} Id_{I\!D} & \text{if } n = start(G') \\ \bigsqcap \{ [\![(m,n)]\!]' \circ [\![m]\!] \mid m \in pred(n) \} & \text{otherwise} \end{cases}$$

2. Compute the effect $[\![\bar{G}]\!]^*$ of the innermost parallel statements $\bar{G}$ of $\hat{G}$ by

$$[\![\bar{G}]\!]^* = \bigsqcap_{par} \{ [\![G']\!]^* \mid G' \in \mathcal{G}_{\mathcal{C}}(\bar{G}) \} \quad \text{(Synchronization)}$$

3. Transform $\hat{G}$ by replacing every innermost parallel statement $\bar{G} = (\bar{N}, \bar{E}, \bar{\mathbf{s}}, \bar{\mathbf{e}})$ by $(\{\bar{\mathbf{s}}, \bar{\mathbf{e}}\}, \{(\bar{\mathbf{s}}, \bar{\mathbf{e}})\}, \bar{\mathbf{s}}, \bar{\mathbf{e}})$, define the local semantics of $(\bar{\mathbf{s}}, \bar{\mathbf{e}})$ by $[\![\bar{G}]\!]^*$, and set the function *InterferenceEff* of $\bar{\mathbf{s}}$ and $\bar{\mathbf{e}}$ to

$$\bigsqcap \{ [\![e]\!] \mid e \in Edges(\mathcal{G}_{\mathcal{C}}(\bar{G})) \wedge [\![e]\!] \neq Id_{I\!D} \}$$

where *Edges* yields the set of edges of a graph. Continue with step 1.

In essence, procedure $\mathcal{A}$ is a hierarchical adaptation of the functional version of the *maximal fixed point* (*MFP*) approach in the sense of Kam and Ullman [7] to the parallel setting (cf. [21]). We only consider the second step realizing the synchronization at end nodes of parallel statements in more detail here. Central is the following lemma, which can be proved by means of Lemma 4.2 and Main Lemma 4.3, and where *end* denotes a function mapping a graph to its end node.

Lemma 4.4 *For the PMOP-solution of a parallel flow graph G, which only consists of purely sequential parallel components holds:*

$$PMOP_{pw}(end(G)) = PMOP(end(G)) = \bigsqcap_{par}\{\, [\![\, G' \,]\!]^* \mid G' \in \mathcal{G}_{\mathcal{C}}(G)\,\}$$

The point for proving Lemma 4.4 is that a single statement is responsible for the entire effect of a path. Thus, the effect of every complete path through a parallel statement is already given by the projection of this path onto the parallel component containing the vital statement. This is exploited in the synchronization step above. Formally, Lemma 4.4 can be proved by means of Main Lemma 4.3 together with Lemma 4.1 and Lemma 4.2, and the identification of the functions Cst_{d^p} with the peak-function p_d. The correctness of the complete hierarchical process is then a consequence of Theorem 4.5, which generalizes a similar theorem of [14].

Theorem 4.5 (Hierarchical Coincidence Theorem)
Let $G' \in \mathcal{G}_{\mathcal{P}}(G)$ be a parallel flow graph, and let $[\![\ ]\!] : E \to \mathcal{F}_{\mathbb{D}}$ be a local semantic functional. Then we have:

$$PMOP_{pw}(end(G')) = PMOP(end(G')) = [\![\, G' \,]\!]^*$$

Subsequently, the information on StCs on parallel program parts can be fed into the analysis for SCs on sequential program parts. In spirit, this follows the lines of [14], however, the process must be refined here in order to take the specific meaning of peak-functions represented by Cst_{d^p} into account.

4.2 The Application: Parallel Constant Propagation

In this section we present our basic algorithm for PCP. It is based on the local semantic functional $[\![\ ]\!]_{pcp} : E \to (\Sigma \to \Sigma)$ defined by

$$\forall\, e \in E.\ [\![\, e \,]\!] =_{df} \begin{cases} [\![\, e \,]\!]_{sc} & \text{if } e \in E_{seq} \\ [\![\, e \,]\!]_{stc} & \text{otherwise} \end{cases}$$

For sequential program parts, it coincides with the functional for SCs, for parallel program parts with the functional for StCs. The computation of parallel constants proceeds then essentially in three steps. First, hierarchically computing (cf. procedure $\mathcal{A}$) the semantics of parallel statements according to Section 4.1. Second, computing the data-flow information valid at program points of sequential program parts. And third, propagating the information valid at entry nodes of parallel statements into their components.

The first step has been considered in the previous section. The second step proceeds almost as in the sequential case because at this level parallel statements are considered "super-edges," whose semantics ($[\![\ ]\!]^*$) has been computed in the first step. The third step, finally, can similarly be organized as the corresponding step of the algorithm of [14]. In fact, the complete three-step procedure evolves as an adaption of this algorithm.

Denoting the final annotation computed by the PCP-algorithm by Ann^{pcp} : $N \rightarrow \Sigma$, the set of constants detected, called *parallel simple constants* (*PSCs*), is given by

$$\mathcal{C}_n^{psc} =_{df} \{t \in \mathbf{T} \mid \exists\, d \in \mathbf{D}'.\ \mathcal{E}(t)(Ann_n^{pcp}) = d\}$$

The program transformation is then very simple. For every node $n \in N$, and all its outgoing edges annotated by an assignment $x := t$ with $SubTerms(t) \cap \mathcal{C}_n^{psc} = \{t_1, \ldots, t_k\} \neq \emptyset$, where $SubTerms(t)$ denotes the set of all subterms of t containing at least one operator, replace the terms being maximal in $\{t_1, \ldots, t_k\}$ by the appropriate constant.

The correctness of this approach, whose power has been demonstrated in the example of Figure 1, is a consequence of Theorem 4.6:

Theorem 4.6 (PSC-Correctness) $\forall\, n \in N.\ \mathcal{C}_n \supseteq \mathcal{C}_n^{psc}$.

Aggressive PCP. The basic PCP-algorithm as proposed above can detect the constancy of complex terms inside of parallel statements as e.g. the constancy of $y + b$ at edge **45** in Figure 1. In general, this is the source of *second-order effects* (cf. [20]). In Appendix A this is demonstrated by an example which is a slightly enriched version of the motivating example of Section 1. Second-order effects of PCP can be captured by incrementally reanalyzing affected program parts starting with the enclosing parallel statement. In effect, this leads to a more powerful, aggressive variant of PCP (cf. Figures 7 and 8). Similarly, this holds for further extensions, e.g. to subscripted variables. We concentrated in this article on scalar variables. However, along the lines of related sequential algorithms, our approach can be extended to subscripted variables, too.

5 Conclusions

Extending the framework of [14], we developed an algorithm for constant propagation in parallel programs, which can be implemented as easily and as efficiently as its sequential counterpart for SCs. The key for achieving this was to decompose the algorithm such that it behaves differently on sequential and parallel program parts. On sequential parts this allows our algorithm to be as precise as the algorithm for SCs; on parallel parts this allows it to capture the phenomena of interference and synchronization without having to consider any interleavings. Together with the successful earlier transfer of bitvector-based optimizations to the parallel setting, complemented in this article by the successful transfer of CP, a problem beyond this class, we hope to encourage compiler writers to integrate classical sequential optimizations into parallel compilers.

References

[1] M. M. Brandis and H. Mössenböck. Single-pass generation of static single assignment form for structured languages. *ACM Transactions on Programming Languages and Systems*, 16(6):1684 – 1698, 1994.

[2] J.-H. Chow and W. L. Harrison. State space reduction in abstract interpretation of parallel programs. In *Proceedings of the International Conference on Computer Languages (ICCL'94) (Toulouse, France)*, pages 277 – 288, 1994.

[3] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the 4th Annual Symposium on Principles of Programming Languages (POPL'77) (Los Angeles, California)*, pages 238 – 252. ACM, New York, 1977.

[4] R. Cytron, J. Ferrante, B. K. Rosen, and M. N. Wegman. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451 – 490, 1991.

[5] D. Grunwald and H. Srinivasan. Data flow equations for explicitly parallel programs. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Parallel Programming (PPOPP'93)*, volume *28*,7 of *ACM SIGPLAN Notices*, pages 159–168, 1993.

[6] M. S. Hecht. *Flow Analysis of Computer Programs.* Elsevier, North-Holland, 1977.

[7] J. B. Kam and J. D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7:305 – 317, 1977.

[8] G. A. Kildall. A unified approach to global program optimization. In *Conference Record of the 1st Annual Symposium on Principles of Programming Languages (POPL'73) (Boston, Massachusetts)*, pages 194 – 206. ACM, New York, 1973.

[9] J. Knoop. Eliminating partially dead code in explicitly parallel programs. *Theoretical Computer Science*, 196(1-2):365 – 393, 1998. (Special issue devoted to *Euro-Par'96*).

[10] J. Knoop, D. Koschützki, and B. Steffen. Basic-block graphs: Living dinosaurs? In *Proceedings of the 7th International Conference on Compiler Construction (CC'98) (Lisbon, Portugal)*, Lecture Notes in Computer Science, vol. 1383, pages 65 – 79. Springer-Verlag, Heidelberg, Germany, 1998.

[11] J. Knoop, O. Rüthing, and B. Steffen. Optimal code motion: Theory and practice. *ACM Transactions on Programming Languages and Systems*, 16(4):1117–1155, 1994.

[12] J. Knoop, O. Rüthing, and B. Steffen. Partial dead code elimination. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'94) (Orlando, Florida)*, volume *29*,6 of *ACM SIGPLAN Notices*, pages 147 – 158, 1994.

[13] J. Knoop, B. Steffen, and J. Vollmer. Code motion for parallel programs. In *Proceedings of the Poster Session of the 6th International Conference on Compiler Construction (CC'96) (Linköping, Sweden)*, pages 81 – 88. Technical Report LiTH-IDA-R-96-12, 1996.

[14] 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.

[15] J. Lee, S. P. Midkiff, and D. A. Padua. Concurrent static single assignment form and constant propagation for explicitly parallel programs. In *Proceedings of the 10th International Conference on Languages and Compilers for Parallel Computing (LCPC'97) (Minneapolis, Minnesota)*, Lecture Notes in Computer Science, vol. 1366, pages 114 – 130. Springer-Verlag, Heidelberg, Germany, 1997.

[16] K. Marriot. Frameworks for abstract interpretation. *Acta Informatica*, 30:103 – 129, 1993.

[17] S. P. Midkiff and D. A. Padua. Issues in the optimization of parallel programs. In *Proceedings of the International Conference on Parallel Processing, Volume II (St. Charles, Illinois)*, pages 105 – 113, 1990.

[18] E. Morel and C. Renvoise. Global optimization by suppression of partial redundancies. *Communications of the ACM*, 22(2):96 – 103, 1979.

[19] J. H. Reif and R. Lewis. Symbolic evaluation and the global value graph. In *Conference Record of the 4th Annual Symposium on Principles of Programming Languages (POPL'77) (Los Angeles, California)*, pages 104 – 118. ACM, New York, 1977.

[20] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *Conference Record of the 15th Annual Symposium on Principles of Programming Languages (POPL'88) (San Diego, California)*, pages 2 – 27. ACM, New York, 1988.

[21] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 7, pages 189 – 233. Prentice Hall, Englewood Cliffs, New Jersey, 1981.

[22] H. Srinivasan, J. Hook, and M. Wolfe. Static single assignment form for explicitly parallel programs. In *Conference Record of the 20th Annual Symposium on Principles of Programming Languages* (*POPL'93*) (*Charleston, South Carolina*), pages 260 – 272. ACM, New York, 1993.

[23] J. Vollmer. Data flow analysis of parallel programs. In *Proceedings of the 3rd IEEE International Conference on Parallel Architectures and Compilation Techniques* (*PACT'95*) (*Limassol, Cyprus*), pages 168 – 177. IEEE Computer Society, Los Alamitos, 1995.

[24] M. Wolfe. *High performance compilers for parallel computing.* Addison-Wesley Publishing Company, Redwood City, California, 1996.

[25] H. Zima and B. Chapman. *Supercompilers for parallel and vector computers.* Addison-Wesley Publishing Company, ACM Press, New York, 1991.

A Aggressive PCP: Resolving Second-order Effects

This section illustrates the impact of resolving second-order effects of parallel constant propagation. Figure 7 shows the example considered for demonstration together with the values of those terms, whose constancy is detected by the basic variant of the PCP-algorithm. Figure 8 then shows the effect of its incremental reapplication, which is triggered by detecting the constancy of the term $d * d$ at edge **22**, for resolving the second-order effects showing up at the edges **48** and **60**.

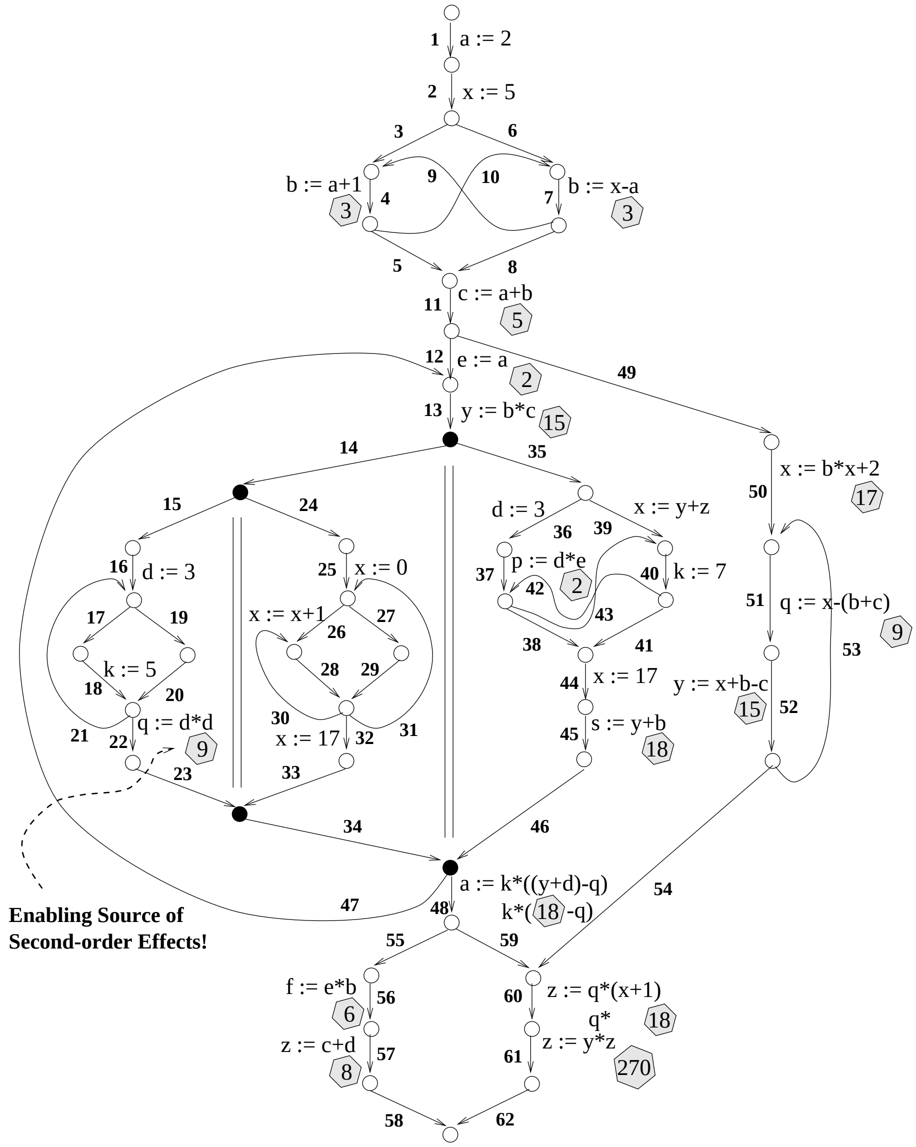

Figure 7: Illustrating first and second-order effects.

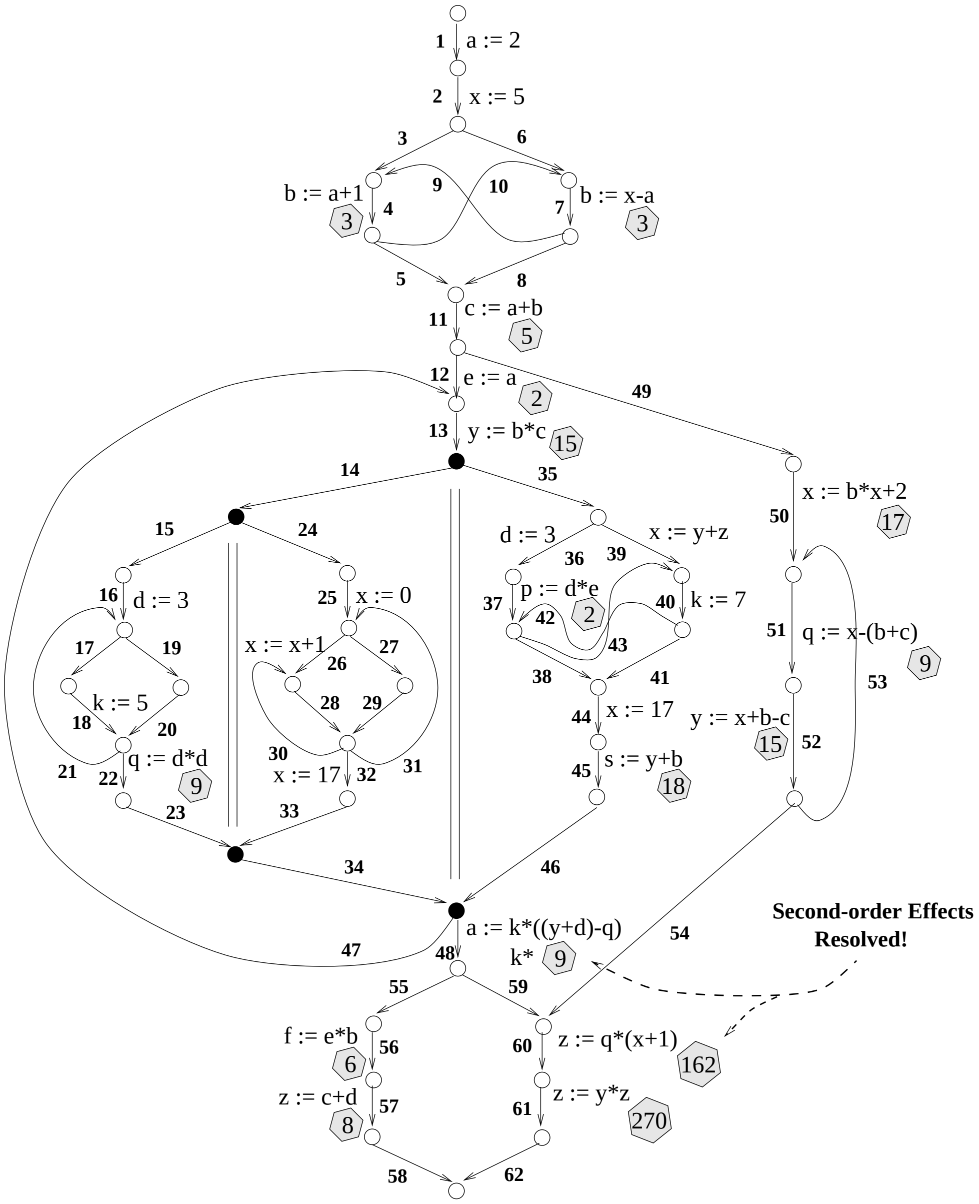

Figure 8: Resolving second-order effects.