Parallel Constant Propagation

Jens Knoop

Universität Passau, D-94030 Passau, Germany
knoop@fmi.uni-passau.de
phone: ++49-851-509-3090 fax: ++49-...-3092

Abstract. *Constant propagation* (*CP*) is a powerful, practically relevant optimization of sequential programs. However, systematic adaptions to the parallel setting are still missing. In fact, because of the computational complexity paraphrased by the catch-phrase "state explosion problem", the successful transfer of sequential techniques is currently restricted to bitvector-based optimizations, which because of their structural simplicity can be enhanced to parallel programs at almost no costs on the implementation and computation side. CP, however, is beyond this class. Here, we show how to enhance the framework underlying the transfer of bitvector problems obtaining the basis for developing a powerful algorithm for *parallel constant propagation* (*PCP*). This algorithm can be implemented as *easily* and as *efficiently* as its sequential counterpart for *simple constants* computed by state-of-the-art sequential optimizers.

1 Motivation

In comparison to automatic parallelization (cf. [15,16]), optimization of *parallel* programs attracted so far only little attention. An important reason may be that straightforward adaptions of sequential techniques typically fail (cf. [10]), 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. [2]) shows that there are exceptions: UBv-problems can be solved as *easily* and as *efficiently* as for sequential programs (cf. [9]). 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 [6] and [8] this has been demonstrated for *partial dead-code elimination* (cf. [7]), and *partial redundancy elimination* (cf. [11]).

Constant propagation (*CP*) (cf. [4,3,12]), however, the application considered in this article, is beyond bitvector-based optimization. CP is a powerful and widely used optimization of sequential programs. It improves performance by replacing terms, which at compile-time can be determined to yield a unique constant value at run-time by that value. Enhancing the framework of [9], we develop in this article an algorithm for *parallel constant propagation* (*PCP*). Like the bitvector-based optimizations it can be implemented as *easily* and as *efficiently* as its sequential counterpart for *simple constants* (*SCs*) (cf. [4,3,12]), which are computed by state-of-the-art optimizers of sequential programs.

The power of this algorithm is demonstrated in Figure 1. 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. We do not know of any other algorithm achieving this.

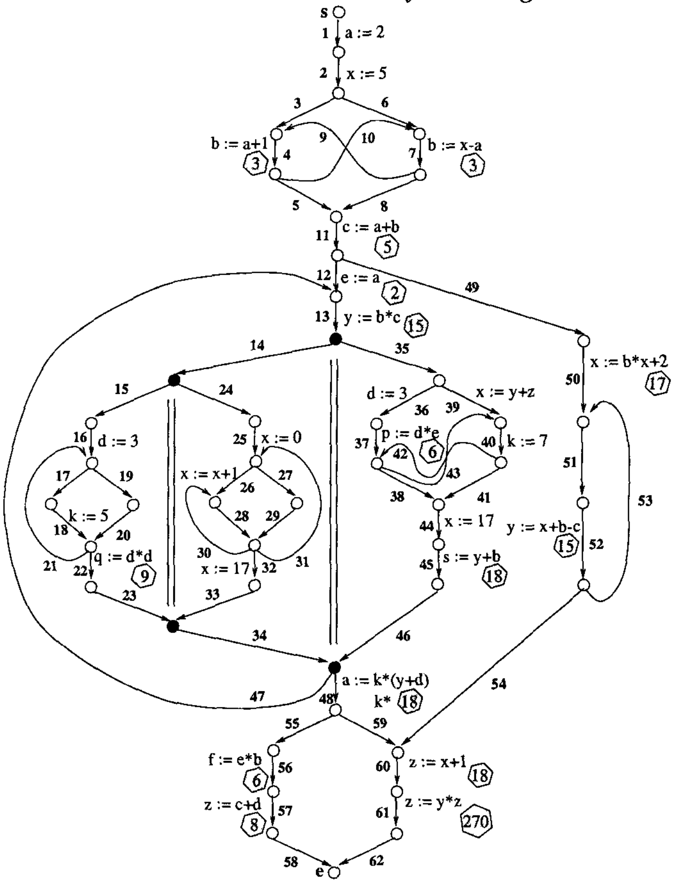

Fig. 1. The power of *parallel constant propagation.*

Importantly, this result is obtained without any performance penalty in comparison to sequential CP. Fundamental for achieving this is to conceptually decompose PCP to behave 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 with respect to side-conditions allowing us as for unidirectional bitvector problems to capture the phenomena of *interference* and *synchronization* of parallel components completely without having to consider interleavings at all. Summarizing, the central contributions of this article are as follows: (1) Extending the framework of [9] to a specific class of non-bitvector problems. (2) Developing on this basis a PCP-algorithm, which works for reducible and irreducible control flow, and is as efficient and can as easily be implemented as its sequential counterpart for SCs. An extended presentation can be found in [5].

2 Preliminaries

This section sketches our parallel setup, which has been presented in detail in [9]. We consider parallel imperative programs with shared memory and interleaving semantics. Parallelism is syntactically expressed by means of a par statement. As usual, we assume that there are neither jumps leading into a component of a parallel statement from outside nor vice versa. As shown in Figure 1, we represent a parallel program by an edge-labelled *parallel flow graph* G with node set N, and edge set E. Moreover, 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* $I = (\mathbf{D'} \cup \{\bot, \top\}, I_0)$, where $\mathbf{D'}$ denotes a non-empty data domain, $\bot$ 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 $I_0(op) : \mathbf{D}^r \to \mathbf{D}$, $\mathbf{D} =_{df} \mathbf{D'} \cup \{\bot, \top\}$ (i.e., $I_0(op)(d_1, \ldots, d_r) = \bot$, whenever there is a j, $1 \leq j \leq r$, with $d_j = \bot$). $\Sigma = \{\sigma \mid \sigma : \mathbf{V} \to \mathbf{D}\}$ denotes the set of *states*, and $\sigma_\bot$ the distinct *start state* assigning $\bot$ to all variables $v \in \mathbf{V}$ (this reflects that we do not assume anything about the context of a program being optimized). The *semantics* of terms $t \in \mathbf{T}$ is then given by the evaluation function $\mathcal{E} : \mathbf{T} \to (\Sigma \to \mathbf{D})$. It 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 $\mathbf{C}$. We introduce the *state transformation function* $\theta_\iota : \Sigma \to \Sigma$, $\iota \equiv x := t$, where $\theta_\iota(\sigma)(y)$ is defined by $\mathcal{E}(t)(\sigma)$, if $y = x$, and by $\sigma(y)$ otherwise. Denoting the set of all *parallel program paths*, i.e., the set of interleaving sequences, from the unique start node $\mathbf{s}$ to a program point n by $\mathbf{PP}[\mathbf{s}, n]$, the set of states Σ_n being possible at n is given by $\Sigma_n =_{df} \{\theta_p(\sigma_\bot) \mid p \in \mathbf{PP}[\mathbf{s}, n]\}$. Here, θ_p denotes the straightforward extension of θ to parallel paths. Based on Σ_n we can now define the set C_n of all terms yielding a unique constant value at program point $n \in N$ at run-time:

$$C_n =_{df} \bigcup \{C_n^d \mid d \in \mathbf{D'}\}$$

with $C_n^d =_{df} \{t \in \mathbf{T} \mid \forall \sigma \in \Sigma_n.\ \mathcal{E}(t)(\sigma) = d\}$ for all $d \in \mathbf{D}'$. Unfortunately, even for sequential programs C_n is in general not decidable (cf. [12]). Simple constants recalled next are an efficiently decidable subset of C_n, which is computed by state-of-the-art optimizers of sequential programs.

3 Simple Constants

Intuitively, a term is a (sequential) *simple constant* (*SC*) (cf. [4]), if it is a program constant or all its operands are simple constants (in Figure 2, a, b, c and d are SCs, whereas e, though it is a constant of value 5, and f, which in fact is not constant, are not). Data flow analysis provides the means for computing SCs.

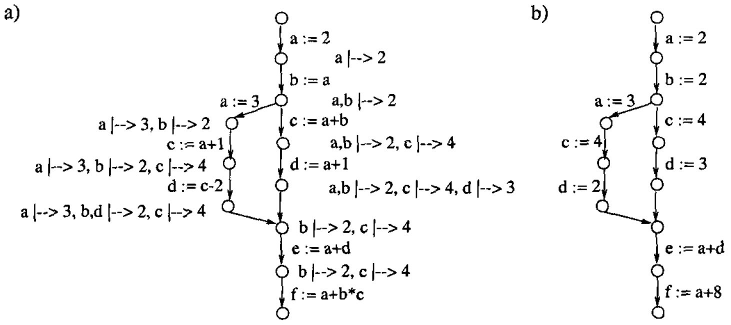

Fig. 2. Simple constants.

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. [1]). Usually, the abstract semantics, which is tailored for a specific problem, 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.

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

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

Unfortunately, this is in general not effective, which leads us to the *maximal fixed point* (*MFP*) approach. 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 a 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, denoted 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. The solution of the *MFP*-approach is then defined as follows:

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

Central is now the following theorem relating both solutions. It gives sufficient conditions for the correctness and even precision of the *MFP*-solution (cf. [3, 4]):

Theorem 1 (Safety and Coincidence).

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

Computing Simple Constants. Considering $\mathbf{D}$ a flat lattice as illustrated in Figure 3(a), the set of states Σ together with the pointwise ordering is a complete lattice, too. The computation of SCs relies then on the local semantic functional $[\![\ ]\!]_{sc} : E \to (\Sigma \to \Sigma)$ defined by $[\![e]\!]_{sc} =_{df} \theta_e$ for all $e \in E$, with respect to the start state $\sigma_\perp$. Because the number of variables occurring in a program is finite, we have (cf. [3]):

Lemma 1. *Σ is of finite height, and all functions $[\![e]\!]_{sc}$, $e \in E$, are monotonic.*

Hence, the *MFP*-solution $MFP^{sc}_{\sigma_\perp}$ of the SC-problem can effectively be computed inducing 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 $\mathcal{C}^{sc,mop}_n =_{df} \{ t \in \mathbf{T} \mid \exists\, d \in \mathbf{D}'.\ \mathcal{E}(t)(MOP^{sc}_{\sigma_\perp}(n)) = d \}$ induced by the $MOP^{sc}_{\sigma_\perp}$-solution, we obtain by means of Theorem 1(1) (cf. [3]):

Theorem 2 (SC-Correctness). $\forall\, n \in N.\ \mathcal{C}_n \supseteq \mathcal{C}^{sc,mop}_n \supseteq \mathcal{C}^{sc,mfp}_n = \mathcal{C}^{sc}_n$.

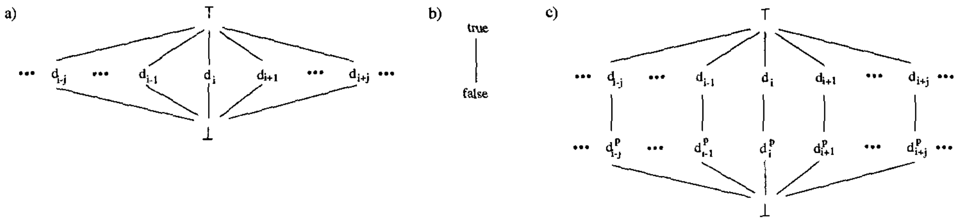
a)
⊤
... d_{i-j} ... d_{i-1} d_i d_{i+1} ... d_{i+j} ...
⊥
b)
true
false
c)
⊤
... d_{i-j} ... d_{i-1} d_i d_{i+1} ... d_{i+j} ...
... d^p_{i-j} ... d^p_{i-1} d^p_i d^p_{i+1} ... d^p_{i+j} ...
⊥

Fig. 3. a), b) Flat (data) lattices. c) Extended lattice.

4 Parallel Constant Propagation

In a parallel program, the validity of the SC-property for terms occurring 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 $[\![\ ]\!]_{stc} : E \rightarrow (\Sigma \rightarrow \Sigma)$ defined by $[\![e]\!]_{stc}(\sigma) =_{df} \theta_e^{stc}(\sigma)$ for all $e \in E$ and $\sigma \in \Sigma$, where θ_e^{stc} denotes the state transformation function of Section 2, where, however, the evaluation function $\mathcal{E}$ is replaced by the simpler $\mathcal{E}_{stc}$, where $\mathcal{E}_{stc}(t)(\sigma)$ is defined by $I_0(c)$, if $t = c \in \mathbf{C}$, and $\perp$ otherwise. The set of strong constants at n, denoted by $\mathcal{C}_n^{stc}$, is then analogously defined to the set of SCs at n.

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

Intuitively, Lemma 2 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 3(b), and the transformation functions relevant for a slot are the two constant functions and the identity on $\mathcal{B}$. However, there is also an important difference. $\mathcal{F}_{\mathbf{D}}$ 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 [9] focusing on the scenario of bitvector 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 program is specified by a local semantic functional $[\![\ ]\!] : E \rightarrow (\mathcal{L} \rightarrow \mathcal{L})$ giving meaning to its statements in terms of functions on an (arbitrary) complete lattice $\mathcal{L}$. In analogy to their sequential counterparts, they can 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:** $\forall\, l_0 \in \mathcal{L}.\; PMOP_{l_0}(n) = \bigsqcap \{ [\![p]\!](l_0) \mid p \in \mathbf{PP}[\mathbf{s}, n] \}$

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. StC-like problems are characterized as UBv-problems by (the simplicity of) their local semantic functional $[\![\ ]\!] : E \to (\mathbb{D} \to \mathbb{D})$. 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 $\mathbb{D}$ is a flat lattice of some underlying (finite) set $\mathbb{D}'$ enlarged by two new elements $\bot$ and $\top$ (cf. Figure 3(a)), and where every function $[\![e]\!]$, $e \in E$, is an element of $\mathcal{F}_{\mathbb{D}} =_{df} \{ Cst_d \mid d \in \mathbb{D}\} \cup \{Id_{\mathbb{D}}\}$. As shown before, $\mathcal{F}_{\mathbb{D}}$ is not closed under pointwise meet. The possibly resulting peak-functions, however, are here essential in order to always be 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 $\mathbb{D}$ to $\mathbb{D}_X$ by inserting a second layer as shown in Figure 3(c), considering then the set of functions $\mathcal{F}_{\mathbb{D}_X} =_{df} (\mathcal{F}_{\mathbb{D}} \backslash \{Id_{\mathbb{D}}\}) \cup \{Id_{\mathbb{D}_X}\} \cup \{ Cst_{d^p} \mid d \in \mathbb{D}\}$. The functions Cst_{d^p}, $d \in \mathbb{D}'$, 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.

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 for some paths only). 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 $\mathcal{F}_{\mathbb{D}_X}$ are distributive. Moreover, together with the orderings $\sqsubseteq_{seq}$ and $\sqsubseteq_{par}$ displayed in Figure 4, they form two complete lattices with least element $Cst_\bot$ and greatest element $Cst_\top$, and least element $Cst_\bot$ and greatest element $Id_{\mathbb{D}_X}$, respectively. Moreover, $\mathcal{F}_{\mathbb{D}_X}$ is closed under function composition (for both orderings). Intuitively, the "meet" operation with respect to $\sqsubseteq_{par}$ models 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

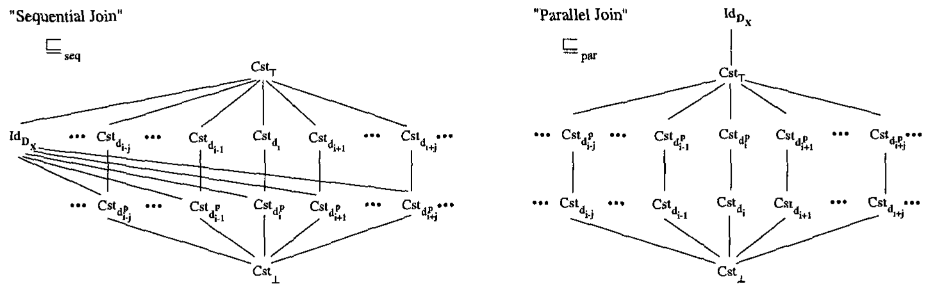

Fig. 4. The function lattices for "sequential" and "parallel join".

functions of $\mathcal{F}_{\mathbb{D}}$, 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 2 and identifying the functions Cst_{d^P}, $d \in \mathbb{D}'$, with the peak-functions p_d, we obtain for every sequential flow graph G and start information out of $\mathcal{F}$:

Lemma 3. $\forall\, n \in N.\ MOP_{pw}(n) = MOP(n) = MFP(n)$

Lemma 3 together with Lemma 4, of which in [9] its variant for bitvector problems was considered, are the key for the efficient computation of the effects of "interference" and "synchronization" for StC-like problems.

Lemma 4 (Main-Lemma). *Let* $f_i : \mathcal{F}_{\mathbb{D}_X} \to \mathcal{F}_{\mathbb{D}_X}$, $1 \leq i \leq q$, $q \in \mathbb{N}$, *be functions of* $\mathcal{F}_{\mathbb{D}_X}$. *Then:* $\exists\, k \in \{1, ..., q\}.\ f_q \circ ... \circ f_2 \circ f_1 = f_k \ \wedge\ \forall\, j \in \{k+1, ..., q\}.\ f_j = Id_{\mathbb{D}_X}$.

Interference. As for UBv-problems, also for StC-like problems the relevance of Main Lemma 4 is that each possible interference at a program point n is due to a single statement in a parallel component, i.e., due to a statement whose execution can be interleaved with the statement of n's incoming edge(s), denoted as n's interleaving predecessors $IntPred(n) \subseteq E$. This is implied by the fact that for each $e \in IntPred(n)$, there is a parallel path leading to n whose last step requires the execution of e. Together with the obvious existence of a path to n that does not require the execution of any statement of $IntPred(n)$, this implies that the only effect of interference is "destruction". In the bitvector situation considered in [9], 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

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

These (precomputable) constant functions $InterferenceEff(n)$, $n \in N$, suffice for modelling interference.

Synchronization. In order to leave a parallel statement, all parallel components are required to terminate. As for UBv-problems, the information required to model this effect can be hierarchically computed by an algorithm, which only considers purely sequential programs. The central idea coincides with that of interprocedural DFA (cf. [14]): 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. In essence, this three-step procedure $\mathcal{A}$ is a hierarchical adaptation of the functional version of the *MFP*-approach to the parallel setting. Here we only consider the second step realizing the synchronization at end nodes of parallel statements in more detail. This step can essentially be reduced to the case of parallel statements G with purely sequential components $G_1, \ldots, G_k$. Thus, the global semantics $[\![G_i]\!]^*$ of the component graphs G_i can be computed as in the sequential case. Afterwards, the global semantics $[\![G]\!]^*$ of G is given by:

$$[\![G]\!]^* = \sqcap_{par}\{ [\![G_i]\!]^* \mid i \in \{1, \ldots, k\} \} \qquad (\textit{Synchronization})$$

Central for proving the correctness of this step, i.e., $[\![G]\!]^*$ coincides with the *PMOP*-solution, is again the fact that a single statement is responsible for the entire effect of a path. Thus, it 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, it can be proved by Main Lemma 4 together with Lemma 2 and Lemma 3, and the identification of the functions Cst_{d^p} with the peak-function p_d. The correctness of the complete hierarchical process then follows by a hierarchical coincidence theorem generalizing the one of [9]. 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 [9], however, the process is 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

We now present our algorithm for PCP. It is based on the semantic functional $[\![\ ]\!]_{pcp} : E \to (\Sigma \to \Sigma)$, where $[\![e]\!]$ is defined by $[\![e]\!]_{stc}$, if e belongs to a parallel statement, and by $[\![e]\!]_{sc}$ otherwise. Thus, for sequential program parts, $[\![\ ]\!]_{pcp}$ 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. (1) Hierarchically computing (cf. procedure $\mathcal{A}$) the semantics of parallel statements according to Section 4.1. (2) Computing the data flow information valid at program points of sequential program parts. (3) 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 [9]. In fact, the complete three-step procedure evolves as an adaption of this algorithm. Denoting the final annotation computed by the PCP-algorithm by $Ann^{pcp} : N \to \Sigma$, the set of constants detected, called *parallel simple constants* (*PSCs*), is given by

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

The correctness of this approach, whose power has been demonstrated in the example of Figure 1, is a consequence of Theorem 3:

Theorem 3 (PSC-Correctness). $\forall n \in N.\ C_n \supseteq C_n^{psc}$.

Aggressive PCP. PCP can detect the constancy of complex terms inside parallel statements as e.g. of $y + b$ at edge **45** in Figure 1. In general, though not in this example, this is the source of *second-order effects* (cf. [13]). They can be captured by incrementally reanalyzing affected program parts starting with the enclosing parallel statement. In effect, this leads to an even more powerful, aggressive variant of PCP. Similar this holds for extensions to subscripted variables. We here concentrated 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 [9], we developed a PCP-algorithm, 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 to behave differently on sequential and parallel program parts. On sequential parts this allows us to be as precise as the algorithm for SCs; on parallel parts it allows us to capture the phenomena of interference and synchronization without having to consider any interleaving. Together with the successful earlier transfer of bitvector-based optimizations to the parallel setting, complemented here by the successful transfer of CP, a problem beyond this class, we hope that compiler writers are encouraged to integrate classical sequential optimizations into parallel compilers.

References

1. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conf. Rec. 4th Symp. Principles of Prog. Lang.* (*POPL'77*), pages 238 - 252. ACM, NY, 1977.
2. M. S. Hecht. *Flow Analysis of Computer Programs.* Elsevier, North-Holland, 1977.
3. J. B. Kam and J. D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7:309 - 317, 1977.
4. G. A. Kildall. A unified approach to global program optimization. In *Conf. Rec. 1st Symp. Principles of Prog. Lang.* (*POPL'73*), pages 194 - 206. ACM, NY, 1973.
5. J. Knoop. Constant propagation in explicitly parallel programs. Technical report, Fak. f. Math. u. Inf., Univ. Passau, Germany, 1998.
6. J. Knoop. Eliminating partially dead code in explicitly parallel programs. *TCS*, 196(1-2):365 - 393, 1998. (Special issue devoted to *Euro-Par'96*).

7. J. Knoop, O. Rüthing, and B. Steffen. Partial dead code elimination. In *Proc. ACM SIGPLAN Conf. on Prog. Lang. Design and Impl. (PLDI'94)*, volume *29*,6 of *ACM SIGPLAN Not.*, pages 147 - 158, 1994.
8. J. Knoop, B. Steffen, and J. Vollmer. Code motion for parallel programs. In *Proc. of the Poster Session of the 6th Int. Conf. on Compiler Construction (CC'96)*, pages 81 - 88. TR LiTH-IDA-R-96-12, 1996.
9. J. Knoop, B. Steffen, and J. Vollmer. Parallelism for free: Efficient and optimal bitvector analyses for parallel programs. *ACM Trans. Prog. Lang. Syst.*, 18(3):268 - 299, 1996.
10. S. P. Midkiff and D. A. Padua. Issues in the optimization of parallel programs. In *Proc. Int. Conf. on Parallel Processing, Volume II*, pages 105 - 113, 1990.
11. E. Morel and C. Renvoise. Global optimization by suppression of partial redundancies. *Comm. ACM*, 22(2):96 - 103, 1979.
12. J. H. Reif and R. Lewis. Symbolic evaluation and the global value graph. In *Conf. Rec. 4th Symp. Principles of Prog. Lang. (POPL'77)*, pages 104 - 118. ACM, NY, 1977.
13. B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *Conf. Rec. 15th Symp. Principles of Prog. Lang. (POPL'88)*, pages 2 - 27. ACM, NY, 1988.
14. 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, NJ, 1981.
15. M. Wolfe. *High performance compilers for parallel computing.* Addison-Wesley, NY, 1996.
16. H. Zima and B. Chapman. *Supercompilers for parallel and vector computers.* Addison-Wesley, NY, 1991.