UNIVERSITÄT PASSAU
Fakultät für Mathematik und Informatik

Eliminating Partially Dead Code in Explicitly Parallel Programs[1]

Jens Knoop[2]
Fakultät für Mathematik und Informatik
Universität Passau
Innstraße 33
D–94032 Passau

MIP-9618
November 1996

[1]A preliminary version of this article appeared in Proceedings of the 2^{nd} European Conference on Parallel Processing (Euro-Par'96) (cf. [Kn]).

[2]Author's current address: Fachbereich 10 — Informatik, Carl von Ossietzky Universität Oldenburg, Uhlhornsweg 49 - 55, D-26111 Oldenburg, Germany. E-mail: jens.knoop@informatik.uni-oldenburg.de. Most of the work was done while the author was at the University of Passau. The author's work is partially funded by the Leibniz Programme of the German Research Council (DFG) under grant Ol 98/1-1.

Abstract

Eliminating *partially dead code* has proved to be a powerful technique for the runtime optimization of sequential programs. In this article, we show how this technique can be adapted to explicitly *parallel* programs with shared memory and interleaving semantics on the basis of a recently presented framework for efficient and precise bitvector analyses for this program setting. Whereas the framework underlying our approach allows a straightforward adaptation of the required data flow analyses to the parallel case, the transformation part of the optimization requires special care in order to preserve parallelism. This preservation is an absolute must in order to guarantee that the optimization does never impair efficiency. The introduction of an appropiate natural side condition suffices to lift even the optimality result known from the sequential setting to the parallel setting.

Keywords

Parallelism, interleaving semantics, shared memory, synchronization, compiler, program optimization, data flow analysis, bitvector problems, code motion, assignment motion, dead code elimination.

Contents

1 Motivation

Partial dead code elimination (*PDE*) improves the runtime efficiency of a program by avoiding the execution of unnecessary statements at runtime. This is illustrated in Figure 1, where the assignment of node **1** of Figure 1(a) is *partially dead* because it is dead on the left branch (i.e., its left hand side variable is not used afterwards in the program), but alive on the right one. This assignment can be eliminated by moving it to the entries of node **3** and node **4** as shown in Figure 1(b). There, the occurrence of node **3** is (totally) dead, and can be eliminated as shown in Figure 1(c).

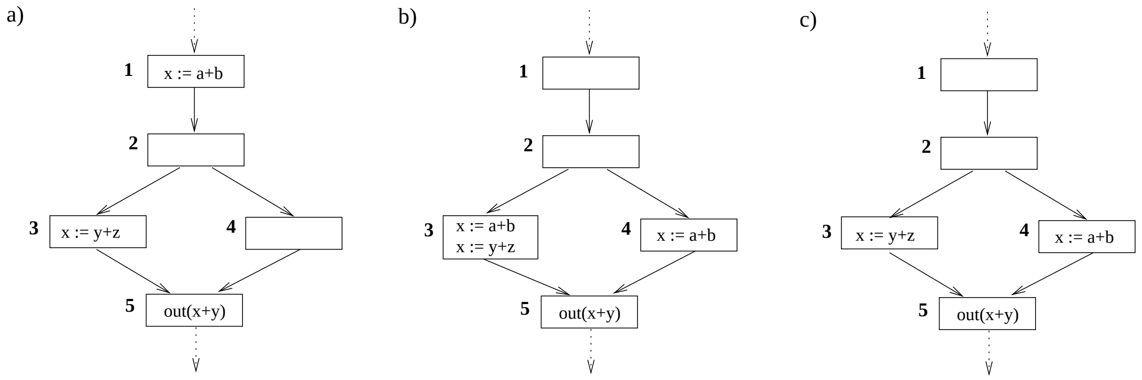

Figure 1: Partial Dead Code Elimination

As illustrated in this example, the effect of PDE depends in essence on the combined effects of two separate transformations: (1) *sinking of assignments*, and (2) *eliminating dead assignments*.

In the *sequential setting*, it is an important observation that assignment sinkings *never* affect the execution time. Intuitively, this fact is responsible for the optimality of the following iterative two-step procedure, which is repeated until the program stabilizes: (1) Moving assignments *as far as possible* in the direction of the control flow (which places them in a context as specific as possible, and thus maximizes the potential of dead code), and (2), eliminating all assignments being dead after the moving step. Actually, this captures all *second order effects* between assignment sinkings and eliminations, and guarantees the following optimality result [KRS3]: Partially dead code remaining in the final program cannot be eliminated without changing the branching structure or the semantics of the program, or without impairing some program executions.

At first sight, the very same iterative process seems to work for parallel programs as well. However, in the *parallel setting* the execution time is extremely sensitive to assignment sinkings. Intuitively, if assignments are moved from a parallel program part into a sequential program part, sinking can dramatically *impair* the execution time. In the worst case, for example, if the assignments of a parallel statement are all data independent and alive, a "naive" realization of the "as-far-as-possible" sinking strategy results in a completely sequentialized program without profiting from any elimination; hence drastically impairing the execution time. This is illustrated in the example of Figure 2(a) and (b), where for clarity the components of the parallel statement are separated by two parallels. In this example the assignments of the two parallel components are data independent and thus can be sunk to node **8** as shown in Figure 2(b). However, as a side-effect of this semantically correct assignment sinking the benefits of executing the assignments in parallel are completely lost. The point demonstrated in this example is that in contrast to the sequential setting, in the parallel setting a subset of semantically correct assignment sinkings violates the basic *requirement* (R) that program optimization should never impair the execution time.

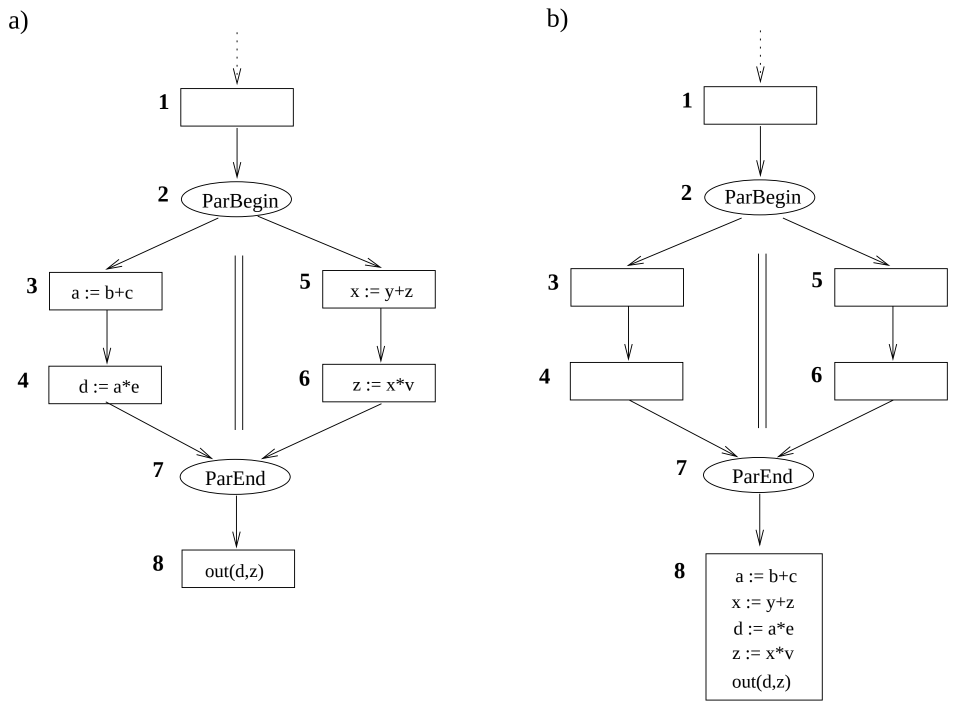

Figure 2: Illustrating the Pitfall of “naive” Assignment Sinking in the Parallel Setting

In this article we show how to cure this defect and the possibly desastrous impact on the execution time of a “naive” transfer of the sequential transformation technique for PDE to the parallel setting by introducing a *natural* and *sufficient* constraint (C). Intuitively, this constraint restricts assignment sinkings to semantically correct sinkings which additionally satisfy requirement (R):

(C) An assignment pattern $\alpha \equiv v := t$ occurring in a component of some parallel statement can only be sunk out of it if occurrences of α can be sunk out of all of its components.

An immediate side-effect of imposing constraint (C) is that assignment sinkings preserve the parallelism of the argument program. Moreover, assignment sinkings moving assignments from sequential to parallel program parts are further supported. In fact, under constraint (C) we can lift the optimality result known from the sequential setting to the parallel setting (cf. Theorem 3.2).

The power of the complete algorithm, which works for reducible as well as for irreducible control flow, is demonstrated in the example of Figure 3. It is discussed in more detail in Section 3. In this example our algorithm is unique to achieve the optimization result of Figure 4. It removes the (totally) dead assignment of node **4**, and eliminates the partially dead assignments of node **1**, **2**, **3**, **7**, **12**, and **16** by moving them to the nodes **8**, **17**, **23**, and **24**, respectively. Moreover, this example illustrates the following three features, which are central for our approach:

1. *Parallelism is preserved:* This is a consequence of imposing constraint (C). Note the different treatment of the assignments to x and z occurring in the parallel statement, which are all partially dead.

2. *Parallelism is transparent:* Assignments can be moved across parallel statements which are "transparent" for them. See the assignment of node **3**, which is moved to node **23**.

3. *Parallelism can be enhanced:* Assignments can be moved from sequential to parallel program parts. See the assignments of node **1** and **2**, which are moved to node **8** and **17**, respectively.

Related Work

For sequential programs there are essentially two algorithms addressing the problem of eliminating partially dead code. First, the "revival transformation" of Feigen et al. [FKCX], and second, the PDE algorithm of Knoop et al. [KRS3], which is underlying the approach in this article. Both algorithms, the latter of which is implemented in the current release of the Sun SPARCompiler[1] language systems, are incomparable as the one of [FKCX] considers complex program statements as sinking candidates, but is not capable of moving statements out of or across loops. For parallel programs, however, PDE has not yet been addressed to the knowledge of the author. In fact, in contrast to the large number of approaches that have been been proposed for the *automatic parallelization* of sequential programs (cf. [BENP, GC, Jo, Le, LY] to cite a few), there is currently very little work on classical *optimizations* for parallel programs (cf. [KY, SW, WS]). This may be due to the fact that naive adaptations of the sequential optimization methods fail [MP], and their straightforward correct adaptations have unacceptable costs caused by the interleavings manifesting the possible executions of a parallel program. In fact, the situation is currently characterized by approaches investigating problems specific to parallel programs like dead lock detection, mutual exclusion or data races (cf.

[1]SPARCompiler is a registered trademark of SPARC International, Inc., and is licensed exclusively to Sun Microsystems, Inc.

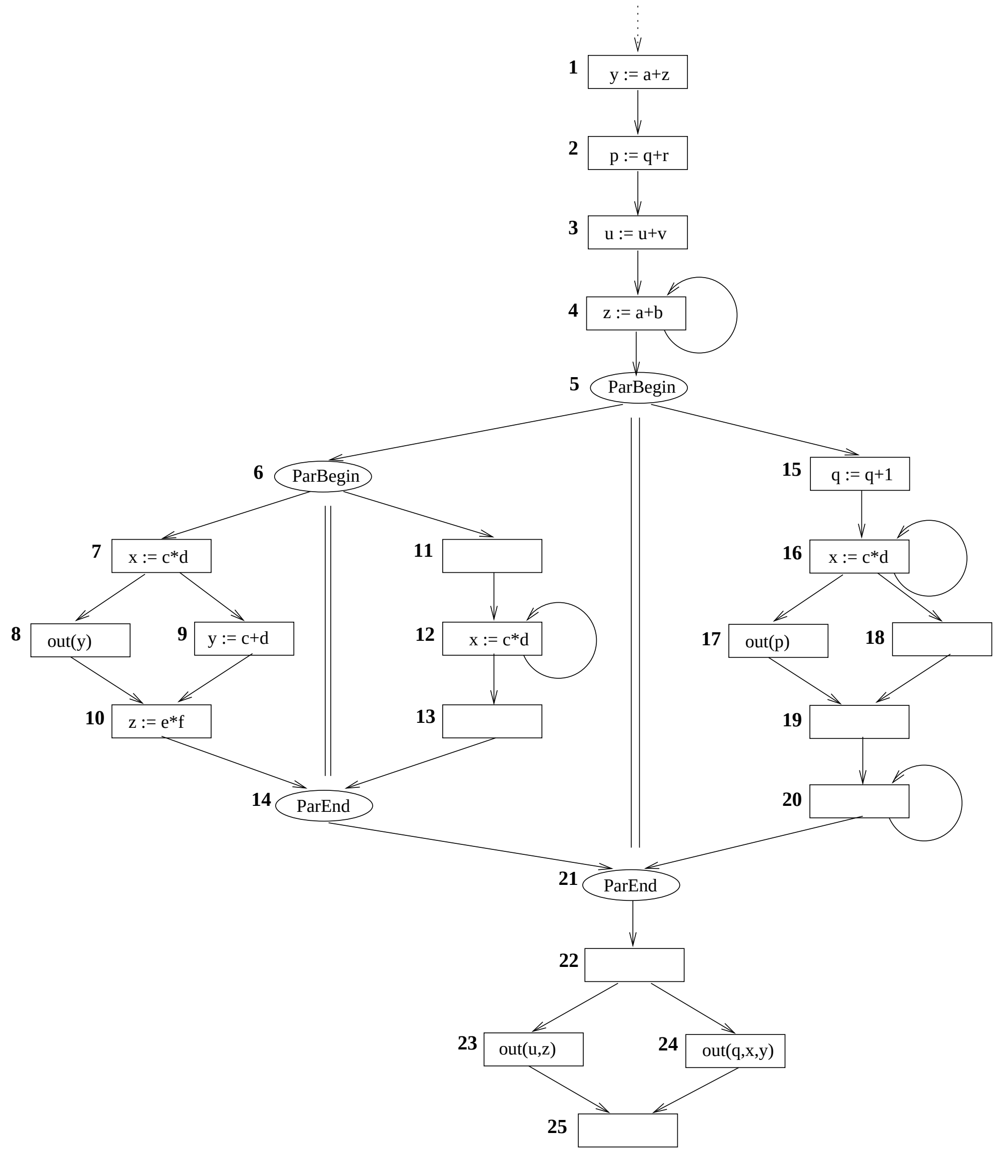

Figure 3: Partial Dead Code Elimination for *Parallel* Programs

[DC]), or by approaches considering restricted situations without shared variables (cf. [LC]) or that require data independence of parallel components (cf. [GS]), which implies that the result of a computation does not depend on the particular choice of interleaving.

Structure of the Paper

In Section 2 we recall the framework of [KSV1] for efficient and precise bitvector analyses of parallel programs, which is underlying the construction of our algorithm for eliminating partially dead code in parallel programs. Central is the Parallel Bitvector Coincidence Theorem 2.6, which states the coincidence of the parallel versions of the *meet over all paths* (*PMOP*) solution and the *maximal fixed point*

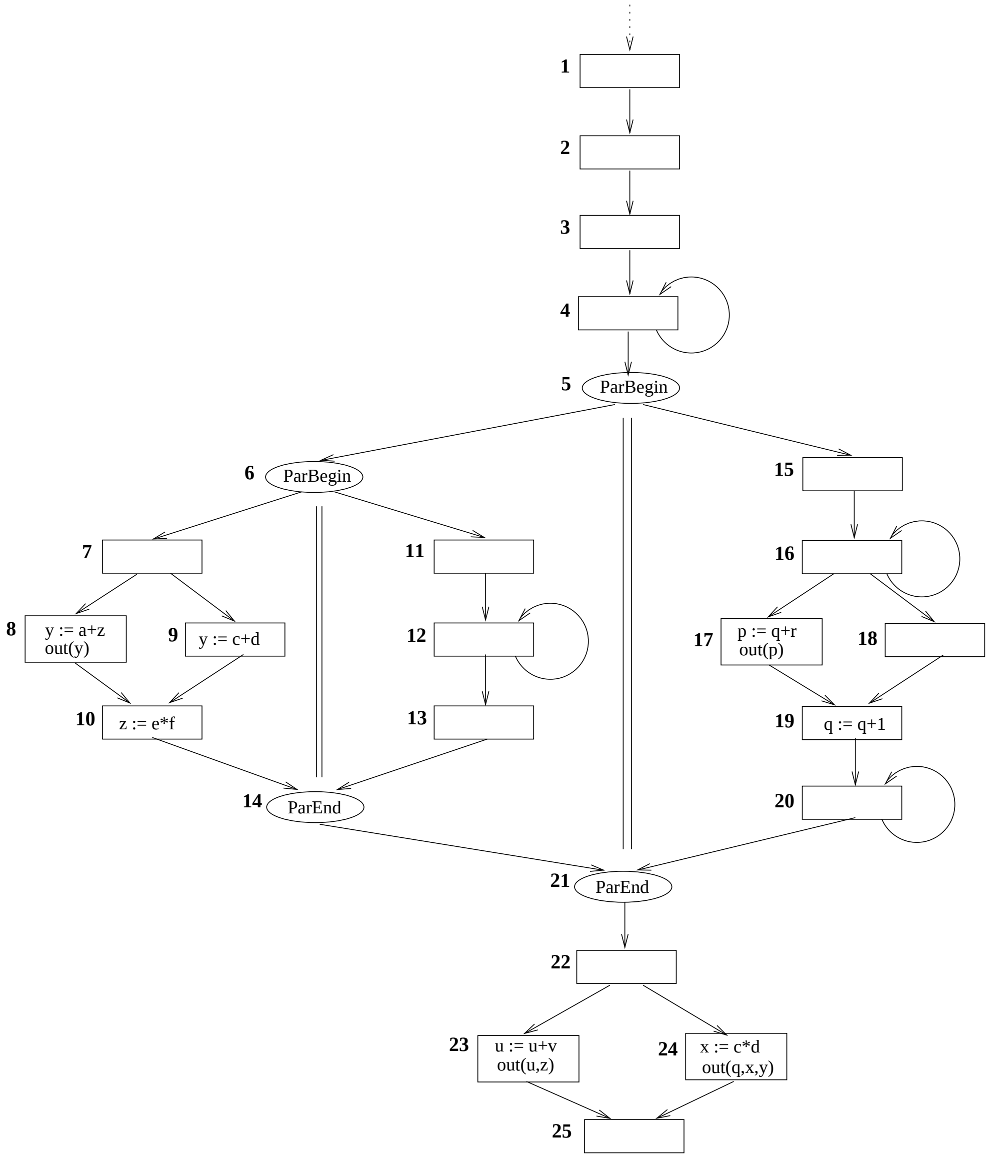

Figure 4: After Partial Dead Code Elimination

(*PMFP*) solution of a unidirectional bitvector problem. Based on this theorem, we subsequently develop our PDE algorithm for parallel programs in Section 3 by systematically enhancing its counterpart for sequential programs of [KRS3]. Afterwards, we draw our conclusions in Section 4. The Appendix, finally, contains the generic algorithm for computing the *PMFP*-solution.

2 The Parallel Setting

This section sketches our setup, which has been presented in detail in [KSV1]. We consider a parallel imperative programming language with interleaving semantics. Parallelism is syntactically expressed by means of a `par` statement whose components are executed in parallel on a shared memory. As usual, we assume that there are neither jumps entering a component of a parallel statement from outside nor vice versa.

2.1 Parallel Flow Graphs

Similar to [SHW] and [GS], we represent a parallel program by a nondeterministic *parallel flow graph* $G^* = (N^*, E^*, \mathbf{s}^*, \mathbf{e}^*)$ with node set N^* and edge set E^* as illustrated in Figure 3. As in a sequential flow graph, nodes $n \in N^*$ represent the statements, and edges $(m, n) \in E^*$ the nondeterministic branching structure of the progam under consideration, while $\mathbf{s}^*$ and $\mathbf{e}^*$ denote the distinct *start node* and *end node* of the graph, which are assumed to represent the empty statement `skip`, and to have no incoming and outgoing edges, respectively. A parallel statement is represented by a subgraph, which is encapsulated by a `ParBegin` node and a `ParEnd` node representing both `skip`. For clarity, `ParBegin` nodes and `ParEnd` nodes are represented by ellipses, and additionally the component graphs of a parallel statement are separated by two parallels as shown in Figure 3.

Interleaving Predecessors

For a sequential flow graph G, the set of nodes that might precede a node n at runtime is precisely given by the set $pred_G(n)$ of its predecessors in the graph. For a parallel flow graph, however, the interleaving of parallel components must also be taken into account. Here, a node n occurring in a component of some `par` statement can at runtime also be preceded by any node of another component of this `par` statement. As in [KSV1] we denote these "potentially parallel" nodes of a node n as its *interleaving predecessors*, denoted by $Pred_{G^*}^{Itlvg}(n)$. In the example of Figure 5 node **8** and node **9** are the "ordinary" predecessors of node **10**, while the nodes **11**, **12**, and **13**, and **15** up to **20** are its interleaving predecessors.

Program Paths of Parallel Programs

The interleaving semantics of parallel imperative programs can be defined via a translation that reduces them to (much larger) nondeterministic programs, which explicitly represent all the possible interleavings. These "product" programs directly induce the notion of a (finite) feasible path of a parallel program, or for short, of a *parallel path*: A node sequence of a parallel program is a parallel path iff it is a path in the corresponding product program. We denote the set of all parallel paths from m to n or to a predecessor of n by $\mathbf{PP}_{G^*}[m, n]$ and $\mathbf{PP}_{G^*}[m, n[$, respectively.

2.2 Data Flow Analysis

In essence, for imperative programming languages *data flow analysis* (*DFA*) provides information about the program states that may occur at some given program points during execution (cf. [He, MJ]). Theoretically well-founded are DFAs that are based

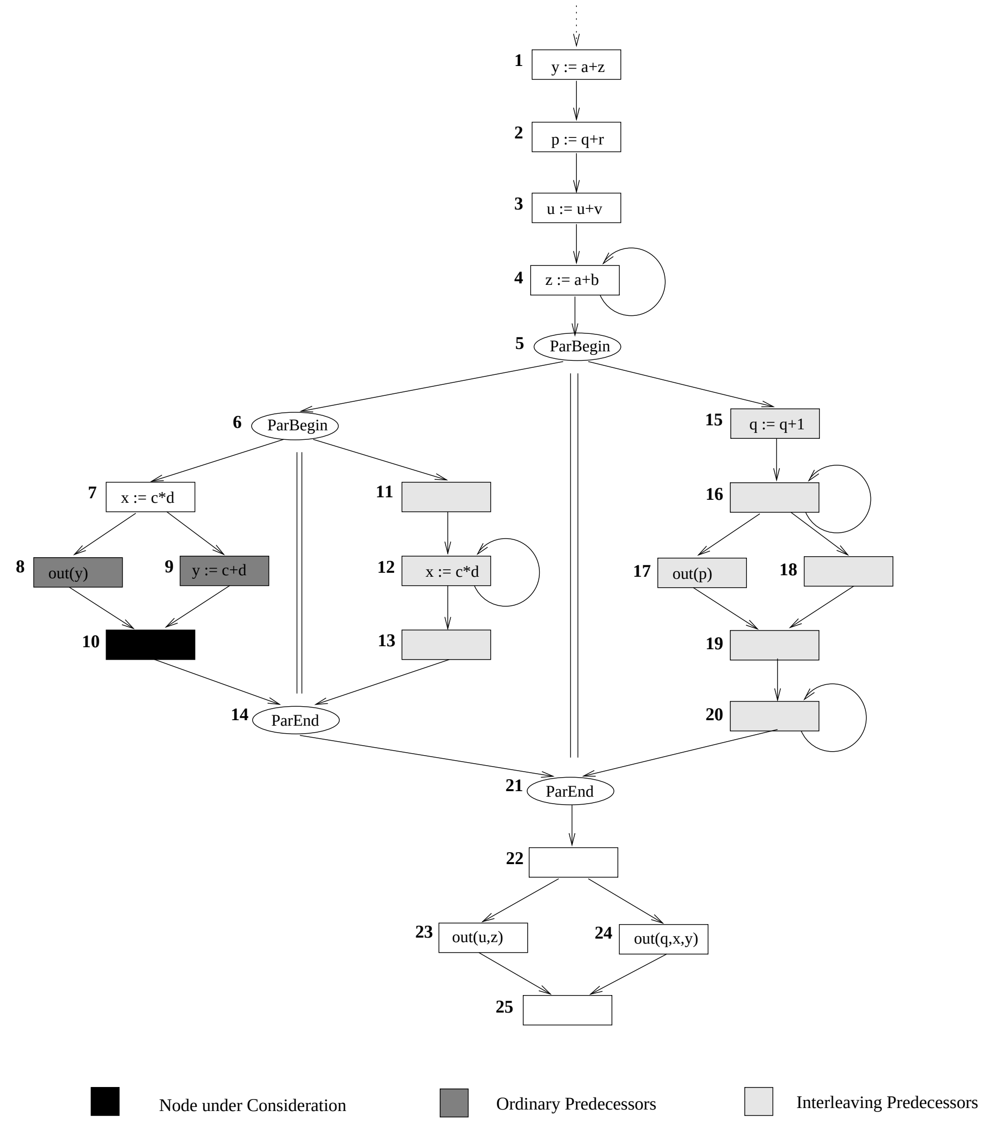

Figure 5: Ordinary and Interleaving Predecessors

on *abstract interpretation* (cf. [CC, Ma]). The point of this approach is to replace the "full" semantics by a simpler more abstract version tailored to deal with a specific problem. In the sequential setting, the abstract semantics is usually specified by a *local semantic functional*, which gives abstract meaning to each program statement in terms of a transformation function on a complete lattice $(\mathcal{C}, \sqcap, \sqsubseteq, \bot, \top)$ with least element $\bot$ and greatest element $\top$, whose elements express the DFA-information of interest. In our framework this carries over to the parallel setting, i.e., as for a sequential program, a DFA for a parallel program is completely specified by means of a local semantic functional

$$[\![\ ]\!] : N^* \to (\mathcal{C} \to \mathcal{C})$$

which gives abstract meaning to every node n of a parallel flow graph G^* in terms of a function on a complette lattice $\mathcal{C}$. In order to define the solution of a DFA problem, it is important that a local semantic functional can easily be extended to cover also parallel paths. Denoting the identity on $\mathcal{C}$ by $Id_{\mathcal{C}}$, we define for each path $p=(n_1,\ldots,n_q)\in \mathbf{PP}_{G^*}[m,n]$:

$$[\![\, p \,]\!] =_{df} \begin{cases} Id_{\mathcal{C}} & \text{if } q<1 \\ [\![\, (n_2,\ldots,n_q) \,]\!] \circ [\![\, n_1 \,]\!] & \text{otherwise} \end{cases}$$

This extension is the key for defining the solution of the parallel version of the *meet over all paths* (*MOP*) approach in the sense of Kam and Ullman [KU], which specifies the intuitively desired solution of a DFA-problem. The *MOP*-approach (in the parallel setting the *PMOP*-approach) directly mimics possible program executions in that it "meets" (intersects) all informations belonging to a program path reaching the program point $n\in N^*$ under consideration.

The *PMOP*-Solution:

$$\forall\, n\in N^*\ \forall\, c_0\in\mathcal{C}.\ PMOP_{(G^*,[\![\ ]\!])}(n)(c_0) = \bigsqcap \{\, [\![\, p \,]\!](c_0) \mid p\in \mathbf{PP}_{G^*}[\mathbf{s}^*,n[\,\}$$

This directly reflects our desires, but is in general not effective. However, as we will recall in the following section, for unidirectional bitvector problems there exists an elegant and efficient way for computing the *PMOP*-solution by means of a fixpoint computation.

Bitvector Analyses

Unidirectional bitvector problems are characterized by the simplicity of their local semantic functional

$$[\![\ ]\!] : N^* \to (\mathcal{B}\to\mathcal{B})$$

which specifies the effect of a node n on a particular component of the bitvector, where $\mathcal{B}$ is the lattice $(\{ff, tt\}, \sqcap, \sqsubseteq)$ of Boolean truth values with $ff \sqsubseteq tt$ and the logical "and" as meet operation $\sqcap$ (or its dual counterpart with $tt \sqsubseteq ff$ and the logical "or" as meet operation $\sqcap$). Central for our efficient fixpoint approach are the following obvious facts on the semantic domain $\mathcal{F}_{\mathcal{B}}$ of the monotonic Boolean functions $\mathcal{B}\to\mathcal{B}$ of bitvector analyses (cf. [KSV1]). We have:

Proposition 2.1 *1. $\mathcal{F}_{\mathcal{B}}$ consists of the constant functions $Const_{tt}$ and $Const_{ff}$, together with the identity $Id_{\mathcal{B}}$ on $\mathcal{B}$ only.*

2. All functions of $\mathcal{F}_{\mathcal{B}}$ are distributive.

3. $\mathcal{F}_{\mathcal{B}}$ together with the orderings $\sqsubseteq_{seq}$ and $\sqsubseteq_{par}$ defined by $Const_{ff} \sqsubseteq_{seq} Id_{\mathcal{B}} \sqsubseteq_{seq} Const_{tt}$ and $Const_{ff} \sqsubseteq_{par} Const_{tt} \sqsubseteq_{par} Id_{\mathcal{B}}$, respectively, forms two complete lattices with least element $Const_{ff}$ and greatest element $Const_{tt}$, and least element $Const_{ff}$ and greatest element $Id_{\mathcal{B}}$, respectively, which are closed under function composition.

Note that $\sqsubseteq_{seq}$ is just the pointwise ordering between functions of $\mathcal{F}_{\mathcal{B}}$. In the following we will usually drop the index "seq". Hence, $\sqcap$ and $\sqsubseteq$ expand to $\sqcap_{seq}$ and $\sqsubseteq_{seq}$, respectively. The following lemma will be the key to the efficient computation of the "interleaving effect". It pin-points the specific nature of a domain of functions that only consists of constant functions and the identity on an arbitrary set M.

Lemma 2.2 (Main-Lemma)
Let $f_i : \mathcal{F}_{\mathcal{B}} \to \mathcal{F}_{\mathcal{B}}$, $1 \leq i \leq q$, $q \in I\!N$, *be functions on* $\mathcal{F}_{\mathcal{B}}$. *Then:* $\exists\, k \in \{1, \ldots, q\}.\ f_q \circ \ldots \circ f_2 \circ f_1 = f_k \ \wedge\ \forall\, j \in \{k+1, \ldots, q\}.\ f_j = Id_{\mathcal{B}}$.

Interference: The relevance of Main Lemma 2.2 for bitvector problems is that 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 n. Combining this observation with the fact that for $m \in Pred^{Itlvg}_{G^*}(n)$, there exists a parallel path leading to n whose last step requires the execution of m, we obtain that the potential of interference, which in general would be given in terms of paths, is fully characterized by the set $Pred^{Itlvg}_{G^*}(n)$ of interleaving predecessors of n. In fact, considering the computation of universal properties that 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 $Pred^{Itlvg}_{G^*}(n)$ implies that the only effect of interference is "destruction". This motivates the introduction of the predicate *NonDestructible* defined for each node $n \in N^*$ by

$$NonDestructible(n) \Longleftrightarrow_{df} \forall\, m \in Pred^{Itlvg}_{G^*}(n).\ [\![\, m \,]\!] \in \{Const_{tt}, Id_{\mathcal{B}}\}.$$

Intuitively, *NonDestructible* indicates that no node of a parallel component destroys the property under consideration, i.e., for all $m \in Pred^{Itlvg}_{G^*}(n)$ holds $[\![\, m \,]\!] \neq Const_{ff}$. Note that only the constant function given by the precomputed value of this predicate is used in Definition 2.5 to model interference, and in fact, Theorem 2.6 guarantees that this modelling is sufficient. Algorithm A.1 computes this predicate as a side result.

Synchronization: Besides taking care of possible interference, we also need to take care of the synchronization required by nodes in N^*_X, i.e., by the end nodes of parallel statements: in order to leave a parallel statement, all its parallel components are required to terminate. The information which is necessary to model this effect can be computed by a hierarchical algorithm that only considers purely sequential programs. The central idea coincides with that of interprocedural DFA (cf. [KS]): we need 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 given below, where $\mathcal{G}_{\mathcal{P}}(G)$ denotes the set of all subgraphs of G representing a parallel statement, $\mathcal{G}_{\mathcal{C}}(\bar{G})$ the set of all component graphs of a parallel statement $\bar{G}$, and *start* a function mapping a graph to its start node:

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

$$[\![n]\!] = \begin{cases} Id_{\mathcal{C}} & \text{if } n = start(G') \\ \bigsqcap\{ [\![m]\!]' \circ [\![m]\!] \mid m \in pred_G(n)\} & \text{otherwise} \end{cases}$$

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

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

3. Transform G by replacing all innermost parallel statements $\bar{G}$ by a graph with two nodes $\bar{\mathbf{s}}$ and $\bar{\mathbf{e}}$ linked by an edge from $\bar{\mathbf{s}}$ to $\bar{\mathbf{e}}$ and attach with $\bar{\mathbf{s}}$ and $\bar{\mathbf{e}}$ the local semantics $Id_{\mathcal{B}} \sqcap \bigsqcap\{ [\![n]\!] \mid n \in \bar{N}\}$ and $[\![\bar{G}]\!]^*$, respectively. Continue with step 1.

In essence, this three-step procedure is a straightforward hierarchical adaptation of the functional version of the *maximal fixed point* (*MFP*) approach in the sense of Kam and Ullman [KU] to the parallel setting (cf. [SP]).[2] Here we only consider the second step realizing the synchronization at end nodes of parallel statements in more detail. In fact, adapting this step will be the key for overcoming the pitfalls of assignment sinkings in the parallel setting in Section 3. Central is the following lemma, which can be proved by means of Main Lemma 2.2. Here, *end* is a function mapping a graph to its end node:

Lemma 2.3 *The PMOP-solution of a parallel flow graph G that only consists of purely sequential parallel components is given by:*

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

The point of this lemma is that a single statement is responsible for the entire effect of a path. Thus, the effect of each complete path through a parallel statement is already given by the projection of this path onto the parallel component containing the vital statement. Thus, in order to model the effect (or *PMOP*-solution) of a parallel statement, it is sufficient to combine the effects of all paths local to the components, a fact, which is formalized in Lemma 2.3.

The correctness of the complete hierarchical preprocess is then a consequence of the Hierarchical Coincidence Theorem 2.4:

Theorem 2.4 (Hierarchical Coincidence Theorem)
Let $G \in \mathcal{G}_{\mathcal{P}}(G^*)$ be a parallel flow graph, and $[\![\ ]\!] : N^* \to \mathcal{F}_{\mathcal{B}}$ a local semantic functional. Then we have:

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

[2] See [KSV1] for details.

After the hierarchical preprocess, the functional $[\![\ ]\!] : N^* \to \mathcal{F}_{\mathcal{B}}$ introduced in Definition 2.5 is the key for characterizing the *PMOP*-solution of a unidirectional bitvector problem algorithmically:

Definition 2.5 The functional $[\![\ ]\!] : N^* \to \mathcal{F}_{\mathcal{B}}$ is defined as the greatest solution of the equation system given by:

$$[\![\, n \,]\!] = \begin{cases} Id_{\mathcal{B}} & \text{if } n = \mathbf{s}^* \\ [\![\, pfg(n) \,]\!]^* \circ [\![\, start(pfg(n)) \,]\!] \sqcap Const_{NonDestructible(n)} & \text{if } n \in N^*_X \\ \bigsqcap \{ [\![\, m \,]\!] \circ [\![\, m \,]\!] \mid m \in pred_{G^*}(n) \} \sqcap Const_{NonDestructible(n)} & \text{otherwise} \end{cases}$$

where $pred_{G^*}(n) =_{df} \{ m \mid (m,n) \in E^* \}$ denotes the set of all immediate predecessors of node n in G^*, and pfg denotes a function, which maps a node occurring in a flow graph of $\mathcal{G}_{\mathcal{P}}(G^*)$ to the smallest flow graph of $\mathcal{G}_{\mathcal{P}}(G^*)$ containing it, and to G^*, otherwise.

In analogy to the *MFP*-solution of Kam and Ullman [KU] for the sequential setting, we can now recall the $PMFP_{BV}$-solution of unidirectional bitvector problems for the parallel setting:

The $PMFP_{BV}$-Solution: $\forall\, n \in N^* \ \forall\, b \in \mathcal{B}.\ PMFP_{BV(G^*,[\![\ ]\!])}(n)(b) = [\![\, n \,]\!](b)$

As in the sequential case, the $PMFP_{BV}$-solution is practically relevant because it can efficiently be computed (see the generic Algorithm A.1 in Appendix A). Moreover, it coincides with the desired *PMOP*-solution (cf. [KSV1]):

Theorem 2.6 (The Parallel Bitvector Coincidence Theorem)
Given a parallel flow graph G^*, and a local semantic functional $[\![\ ]\!] : N^* \to \mathcal{F}_{\mathcal{B}}$, the PMOP-solution and the $PMFP_{BV}$-solution coincide, i.e.,

$$\forall\, n \in N^*.\ PMOP_{(G^*,[\![\ ]\!])}(n) = PMFP_{BV(G^*,[\![\ ]\!])}(n)$$

3 Partial Dead Code Elimination

Based on the framework of Section 2, we will now develop our algorithm for partial dead code elimination in parallel programs. Without loss of generality, we consider an arbitrary, but fixed parallel program G^*. Moreover, we assume that all edges starting in a node outside the set of start nodes of parallel statements with more than one successor have been split by inserting a synthetic node. Edge splitting is typical for code motion transformations (cf. [KRS1]) in order to avoid the blocking of the code motion process by *critical edges* as it is illustrated in Appendix B.

As mentioned in Section 1 the effect of PDE depends on the iterated applications of assignment sinkings and assignment eliminations. In order to be semantics preserving (and profitable), both sinkings and eliminations must satisfy certain *admissibility* constraints. Assignment sinkings, for example, must never move an assignment across a "blocking" instruction, i.e., an instruction which uses or modifies its left-hand side variable, or modifies an operand of its right-hand side term. Assignment eliminations may only remove *dead* assignments, i.e., assignments whose left-hand side variable is not used afterwards in the program without a preceding redefinition. These admissibility constraints can be expressed by means of appropriate predicates. In the following we will write $G \vdash_{(\mathcal{S},\mathcal{E})} G'$ if the parallel flow graph G' results from G by an assignment sinking or elimination to G satisfying the sinking and elimination constraints expressed by $\mathcal{S}$ and $\mathcal{E}$, respectively. Then,

$$\mathcal{G}_{(\mathcal{S},\mathcal{E})} =_{df} \{\, G' \mid G^* \vdash^*_{(\mathcal{S},\mathcal{E})} G' \,\}$$

denotes the universe of all programs being reachable from G^* by sinking and elimination steps respecting $\mathcal{S}$ and $\mathcal{E}$.

In order to compare different programs of $\mathcal{G}_{(\mathcal{S},\mathcal{E})}$, we introduce the notion of the *execution time* of a (parallel) program path as a standard of comparison. For simplicity, we assume that all assignments have unit costs. The *execution time* of a parallel program path is then (structurally) given as follows: for a parallel statement it is the maximum of the execution times of its components for the considered execution, and for a parallel program path, i.e., the sequential composition of elementary and parallel statements, it is the sum of the execution times of its components.

Based on this definition, we say that a program $G' \in \mathcal{G}_{(\mathcal{S},\mathcal{E})}$ is *better* than a program $G'' \in \mathcal{G}_{(\mathcal{S},\mathcal{E})}$, in signs $G'' \precsim G'$,[3] if and only if for all paths p from the start node to the end node in G' the execution time is less or equal to that of the corresponding path in G''.[4] Moreover, a program $\bar{G} \in \mathcal{G}_{(\mathcal{S},\mathcal{E})}$ is *optimal* if and only if $\bar{G}$ is better than any other program in $\mathcal{G}_{(\mathcal{S},\mathcal{E})}$.

Now we can present our algorithm for the optimal elimination of partially dead assignments in a parallel program. This algorithm evolves from stepwise refining the straightforward parallel extension of the underlying sequential algorithm, which we call the "naive" parallel PDE algorithm. Like their sequential counterpart, both the naive and the refined parallel algorithm are composed of two procedures for assignment sinking, controlled by a sinkability analysis, and dead assignment elimination, controlled by a dead variable analysis, which are repeated until the program stabilizes. As in the sequential case this is necessary in order to capture all second order effects of PDE. It is worth noting that both algorithms differ only in their sinking procedure, whereas they share the elimination procedure. Thus, we present the elimination procedure first. It directly evolves from its sequential counterpart.

3.1 The Elimination Procedure

Intuitively, an assignment is dead at a program point, if there is no program continuation on which its left-hand side variable is used without a preceding redefinition.

[3] Note that this relation is reflexive. In fact, *at least as good* would be the more precise but uglier notion.

[4] Remember that PDE preserves the branching structure. Hence, starting from a path in G^*, we can easily identify corresponding paths in G' and G''.

This is a classical *backward* bitvector problem (cf. [He]).[5] Table 1 shows its specification in a form which directly fits into the framework of Section 2. Here, $Used_x$ and Mod_x are two local predicates of nodes, which are true for a node n and a variable pattern x, if x is a right-hand side variable of n, and if x is the left-hand side variable of n, respectively.

$$\forall\, n \in N^*.\ [\![\, n \,]\!]_x^{dd} =_{df} \begin{cases} Const_{tt} & \text{if } \neg\, Used_x(n) \wedge Mod_x(n) \\ Id_{\mathcal{B}} & \text{if } \neg(Used_x(n) \vee Mod_x(n)) \\ Const_{ff} & \text{otherwise} \end{cases}$$

Table 1: Dead Variable Analysis for Variable Pattern x

Note that $[\![\ ]\!]_x^{dd}$ is exactly the same functional as in the sequential case. Indeed, the effect of interference is completely taken care of by the instantiated version of the predicate *NonDestructible* induced by $[\![\ ]\!]_x^{dd}$. "Destruction" by interference here means that the variable under consideration is forced to be alive by some statement of a parallel "relative". However, one does not have to bother about these details when applying the framework. In fact, the functional $[\![\ ]\!]_x^{dd}$ can directly be fed into the generic Algorithm A.1 of the framework of Section 2 computing the *PMFP*-solution, which, as a consequence of the Parallel Bitvector Coincidence Theorem 2.6 coincides with the desired *PMOP*-solution. Hence, the algorithm comes up with the desired set of dead variables. The corresponding program transformation is then very simple and coincides with that of the sequential setting: Eliminate every assignment occurrence, whose left-hand side variable is dead immediately after it.

In spite of the similarity of the elimination procedures for the sequential and the parallel setting there is a difference which is worth to be noted: Immediate profitability. In a sequential program part every elimination of a dead assignment is immediately profitable for each program execution passing this point. This obvious fact does not simply carry over to the parallel setting. We still have that eliminating a dead assignment never impairs the execution time, but immediate profitability of eliminating an assignment in a parallel program part is only given for program executions passing the elimination point, for which the component containing the eliminated assignment is the bottleneck of the program execution under consideration. This is illustrated in the example of Figure 6(a) and (b). Note that the two assignments of node **1** and node **4** are both dead as their left-hand side variable m is never used in the program. While, however, eliminating the assignment of node **1** immediately improves the execution time of each run of the program fragment of Figure 6, eliminating the dead occurrence of node **4** immediately improves only runs, for which the left component of the parallel statement is the bottleneck. Note, however, that there may be second order effects. In the example of Figure 6, eliminating the dead occurrence of node **4**, which may not directly be profitable for some

[5] In contrast, detecting "faint" variables required by the similar problem of "partial faint code elimination" is not a bitvector problem (cf. [KRS3]).

program executions, is the prerequisite for eliminating the assignment of node **2**, which is always profitable.

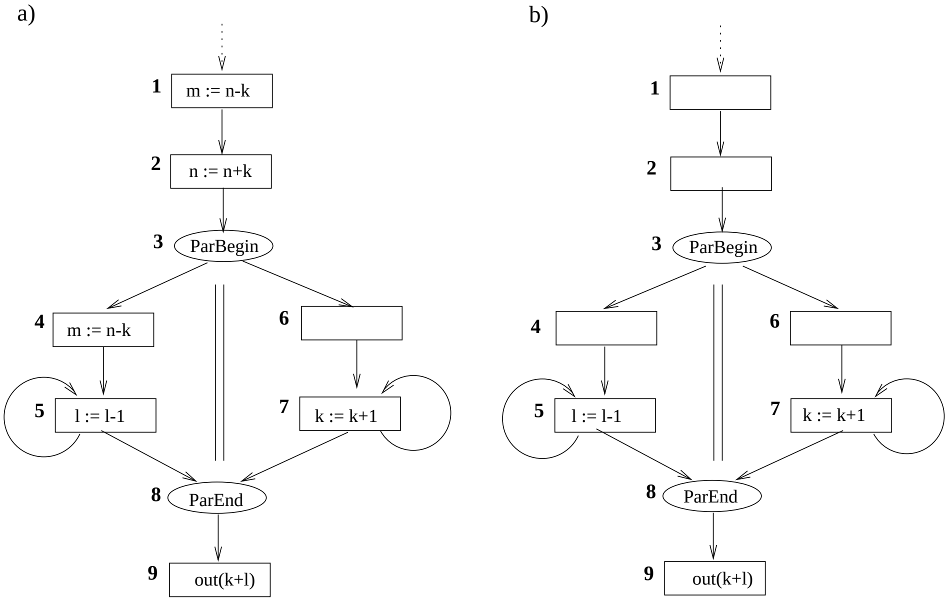

Figure 6: Profitability of Eliminations in Sequential and Parallel Program Parts

3.2 The Sinking Procedure of the "Naive" PDE Algorithm

Central for the sinking procedure of the "naive" PDE algorithm is the admissibility constraint imposed on it. For the "naive" algorithm this is the natural extension $\mathcal{S}_{nv}$ of its sequential counterpart demanding for semantical correctness only (cf. [KRS3]). According to this demand, an assignment must never be moved across an instruction "blocking" it, nor moved to a program point which is not reached by an occurrence of the same assignment pattern from all its predecessors. The local semantic functional of Table 2 then specifies the "naive" sinking procedure completely. Here, $LocSinkable_\alpha$ and $LocBlocked_\alpha$ are two local predicates of nodes, which are true for a node n and an assignment pattern α, if n represents an occurrence of α, and if the statement of n blocks the sinking of α, respectively.

Remark 3.1 Note that occurrences of α satisfy both the predicate $LocSinkable_\alpha$ and $LocBlocked_\alpha$, a fact, which cannot directly be modelled in our abstract domain. While this distinction is unnecessary in the sequential setting, it is vital in the parallel one because of interference. It is taken care of by associating all occurrences of α (in parallel components) with two semantic functions. This does not introduce any subtleties in the framework of Section 2, but just makes the twofold character of an occurrence of α explicit.

For the sinkability analysis "destruction" by interference means that an assignment is blocked by the instruction of a parallel relative. Expressing the result of

$$\forall\, n \in N^*.\ [\![\, n \,]\!]_\alpha^{sk} =_{df} \begin{cases} Const_{tt} & \text{if } LocSinkable_\alpha(n) \\ Id_{\mathcal{B}} & \text{if } \neg(LocSinkable_\alpha(n) \vee LocBlocked_\alpha(n)) \\ Const_{ff} & \text{otherwise} \end{cases}$$

Table 2: Sinkability Analysis for Assignment Pattern $\alpha \equiv v := t$

the sinkability analysis for an assignment pattern α by the predicate $Sink_\alpha$, the corresponding program transformation is as follows: Remove all original occurrences of α satisfying $\bigwedge\{\, Sink_\alpha(m) \,|\, m \in succ(n) \,\}$ and insert new instances of α at the entry of each node satisfying $Sink_\alpha(n) \ \wedge\ (LocBlocked_\alpha(n) \ \vee \bigvee\{\, \neg Sink_\alpha(m) \,|\, m \in succ(n) \,\})$, where $succ(n)$ denotes the set of all successor nodes of n.

Discussing the "Naive" Algorithm: As its sequential counterpart the sinking procedure of the "naive" algorithm moves assignments as far as possible in the direction of the control flow, i.e., as far as it is semantics preserving and does not impair any program execution. Whereas in the sequential case this strategy leads to optimal results because it maximizes the potential of dead code without impairing any program execution, it can be desastrous in the parallel setting as the parallelism of the argument program can be destroyed without profiting from any elimination as it was illustrated in the example of Figure 2. In fact, when applied to the program of Figure 2(a), the "naive" algorithm comes up with the undesired result of Figure 2(b). In the following section we will show how to cure this defect of the "naive" algorithm.

3.3 The Sinking Procedure of the Optimal PDE Algorithm

The point of refining the "naive" sinking procedure is to exclude assignment sinkings which possibly impair the execution time of a program execution. Conceptually, this is achieved by strengthening the admissibility constraint $\mathcal{S}_{nv}$ of the "naive" sinking procedure by additionally imposing constraint (C) introduced in Section 1, i.e., an assignment pattern $\alpha \equiv v := t$ occurring in a component of some parallel statement can only be sunk out of it if occurrences of α can be sunk out of all of its components. In essence, this means to handle synchronization on leaving a parallel statement like an ordinary join of the control flow in the sequential setting.

3.3.1 The First Refinement: Synchronization

In order to take care of constraint (C) during synchronization, it is sufficient to replace the synchronization step of the three-step procedure $\mathcal{A}$ of Section 2 computing the effects of parallel statements by:

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

$$[\![\, \bar{G} \,]\!]^* = \bigsqcap_{seq}\{\, [\![\, G' \,]\!]^* \,|\, G' \in \mathcal{G}_{\mathcal{C}}(\bar{G}) \,\}$$

In contrast to the "real" synchronization mechanism, which, instantiated with the sinkability functional $[\![\ ]\!]^{sk}_{\alpha}$, allows us to move an assignment out of a parallel statement, if it can be moved out of *some* of its components, the modified one requires that this is possible for *all* components. Clearly, this requires that there is at least one occurrence in each component of the parallel statement under consideration. Note, however, that in this case the occurrences interfere and therefore block each other. Thus, as a consequence of modifying the synchronization mechanism as shown above no assignment at all could be sunk out of a parallel statement. Of course, this is too restrictive, and is overcome by the second refinement dealing with interference.

3.3.2 The Second Refinement: Interference

The point of this refinement is to exclude that occurrences of the same assignment pattern prevent each other from sinking due to interference in a parallel statement. In fact, this restriction can be weakened without affecting the program semantics. In the framework of Section 2, it suffices to replace the predicate *NonDestructible* by the weaker version

$$NonDestructible'(n) =_{df}$$

$$\forall\, m \in Pred^{Itlvg}_{G^*}(n).\ Patt(m) \neq Patt(n) \Rightarrow [\![\ m\ ]\!] \in \{Const_{tt}, Id_{\mathcal{B}}\}$$

where *Patt* is a function returning the statement pattern of its argument node. The new version is used for all nodes except for the start nodes of parallel components. This is mandatory in order to prevent that assignments sink from a sequential program part into a parallel statement and get stuck in more than one of its components. In fact, correctness, even in the sense of *sequential consistency* (cf. [La, SS]), gets lost in general. This is particularly obvious for "recursive" assignments, i.e., assignments whose left-hand side variables occur in their right-hand side terms. Conversely, sinking of occurrences of assignment patterns out of each component of a parallel statement as it is intended by the second refinement is always correct for non-recursive patterns. For recursive patterns, however, this depends on the underlying interleaving model. In case of non-atomic assignments, sequential consistency would be preserved, and thus it would be correct. Otherwise, this cannot be guaranteed and hence, the second refinement has to be restricted to "non-recursive" assignment patterns in this case.

3.3.3 Discussing the Motivating Example

The example of Figure 3 illustrates the power of the complete algorithm. It is unique to achieve the optimization displayed in Figure 4. Consider first the assignment to u at node **3**. It is partially dead as its value is only used in the output operation of node **23**. Moreover, note that the parallel statement is transparent for this assignment. Thus, it can directly be moved to the entries of node **23** and **24**, where the occurrence of node **24** is dead, and can thus be eliminated. Consider next the assignment to z in node **4**. It is (totally) dead with respect to the assignment to z in node

10 because the synchronization required by the parallel statement guarantees that every program execution reaching node **21** passes node **10**. Thus, it can also be eliminated. Simultaneously, this removes the blockade of the assignment to y at node **1**, which is partially dead with respect to the assignment to y in node **9**. As the other parallel components are free of statements interfering with that of node **1**, it sinks into the left-most component (but not into the remaining ones because of the interfering statement of node **9**!) to the entries of node **8** and **9**. As it is dead at node **9**, it is eliminated there. Similarly, the partially dead assignment to p at node **2** can be eliminated. After sinking the blocking assignment to q of node **15** to node **19**, it can be moved to the entries of node **17** and **18**, where it is dead at node **18**, and can be eliminated there. Note that the assignment of node **15** is moved across the loop of node **16**, however, it is not moved inside the loop of node **20**. Finally, the second refinement removes the mutual blockade of the three assignments to x of node **7**, **12**, and **16** and allows them to simultaneously be moved to the entries of node **23** and **24**. Here, the occurrence of node **23** is dead, and eliminating it results in the promised program of Figure 4.

3.4 Optimality

In this section we present the main result of Section 3 stating the optimality of our PDE algorithm for parallel programs. To this end let $\mathcal{S}_R$ and $\mathcal{E}$ denote the admissibility constraints applying to assignment sinkings and eliminations in the second refinement, and let G_{pde} denote the program, which finally results from applying the refined algorithm to the argument program G^*. Then we have:

Theorem 3.2 (Optimality) *G_{pde} is optimal in $\mathcal{G}_{(\mathcal{S}_R,\mathcal{E})}$.*

In essence, the proof of Theorem 3.2 follows the lines of [GKLRS] proving the optimality of the sequential counterpart of the PDE algorithm presented here. Intuitively, the proof boils down to proving that every "fair" sequence of applications of sinking and elimination steps terminates with a program of $\mathcal{G}_{(\mathcal{S}_R,\mathcal{E})}$, which is minimal with respect to the order "better" $\precsim$. The central gap in this proof is bridged by an application of the Chaotic Fixpoint Theorem of [GKLRS], a generalization of the classical fixpoint theorem of Knaster and Tarski to families of functions. Intuitively, this theorem can be considered a functional characterization of the well-known *Church-Rosser* property. It guarantees the existence and the finite computability of a least common fixpoint of a family of *increasing* and *delay-monotonic* functions on a *well-founded partial order*. While increasingness means that for every argument d of the partial order under consideration the image $f(d)$ is greater or equal to d, delay-monotonicity is a property being weaker than the usual monotonicity. A family $\mathcal{F} =_{df} (f_k)_{k \in \mathbb{N}}$ of functions on a partial order $\langle D; \sqsubseteq \rangle$ is called *delay-monotonic*, if for each $k \in \mathbb{N}$ and each pair of elements $d, d' \in D$ holds:

$$d \sqsubseteq d' \text{ implies } \exists\, s \in \mathbb{N}^*.\, f_k(d) \sqsubseteq f_s(d')$$

Note that monotonicity always implies delay-monotonicity.

In our application the family of functions $\mathcal{F}$ is given by the set of admissible assignment sinkings and eliminations $\mathcal{F}_{appl}$, and the order by the derivation relation $\vdash_{(\mathcal{S},\mathcal{E})}$ on the program universe $\mathcal{G}_{(\mathcal{S}_R,\mathcal{E})}$, i.e., by the pair $\langle \mathcal{G}_{(\mathcal{S}_R,\mathcal{E})}; \vdash_{(\mathcal{S},\mathcal{E})} \rangle$. Obviously, $\langle \mathcal{G}_{(\mathcal{S}_R,\mathcal{E})}; \vdash_{(\mathcal{S},\mathcal{E})} \rangle$ is a well-founded partial order with least element G^*. Moreover, all functions of $\mathcal{F}_{appl}$ are apparently increasing with respect to the derivation order $\vdash_{(\mathcal{S},\mathcal{E})}$, and can additionally be proven to be delay-monotonic. In essence, the latter proof resembles the classical Newman Lemma [Ne], stating that local confluence on a well-founded partial order implies confluence. The example of Figure 7 illustrates the difference between monotonicity and delay-monotonicity in our application context, and demonstrates that even in the sequential setting the functions of $\mathcal{F}_{appl}$ fail the usual monotonicity property.

Having established the preconditions of the Chaotic Fixpoint Theorem, the desired optimality result follows essentially from combining it with the fact that the transitive closure $\vdash^*_{(\mathcal{S},\mathcal{E})}$ of the derivation relation $\vdash_{(\mathcal{S},\mathcal{E})}$ is a subset of the relation "better" $\sqsubseteq$, and that fixpoints in $\mathcal{G}_{(\mathcal{S}_R,\mathcal{E})}$ are maximal with respect to the transitive closure $\vdash^*_{(\mathcal{S},\mathcal{E})}$. In fact, we have: $\mathcal{G}_{(\mathcal{S}_R,\mathcal{E})}$ has (up to local reorderings in basic blocks) a unique best element (with respect to $\sqsubseteq$), which can be computed in a finite number of steps by any "fair" sequence of function applications from $\mathcal{F}_{appl}$.

4 Conclusions

Optimizing explicitly parallel programs is desirable and possible. In this article we demonstrated this by developing an algorithm for the optimal elimination of partially dead code in a parallel program. An important observation during this development was that in contrast to the DFAs required, which, based on the framework of [KSV1], could straightforward be adapted from their sequential counterparts, the transformation itself required special care. In fact, the naive adaptation of the optimal sequential algorithm could destroy the parallelism of its argument programs, and thus dramatically impair their execution time. By introducing a natural and sufficient side condition this defect could be cured and the optimality result known from the sequential setting be lifted to the parallel setting. Currently, we are investigating how to adapt our approach to other classical optimizations like *code motion* [KRS2] and *assignment motion* [KRS4].

References

[BENP] Banerjee, U., Eigenmann, R., Nicola, A., and Padua, D. A. Automatic program parallelization. In Proc. IEEE 81(2), 1993, 211- 243.

[CC] Cousot, P., and Cousot, R. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the 4^{th} International Symposium on*

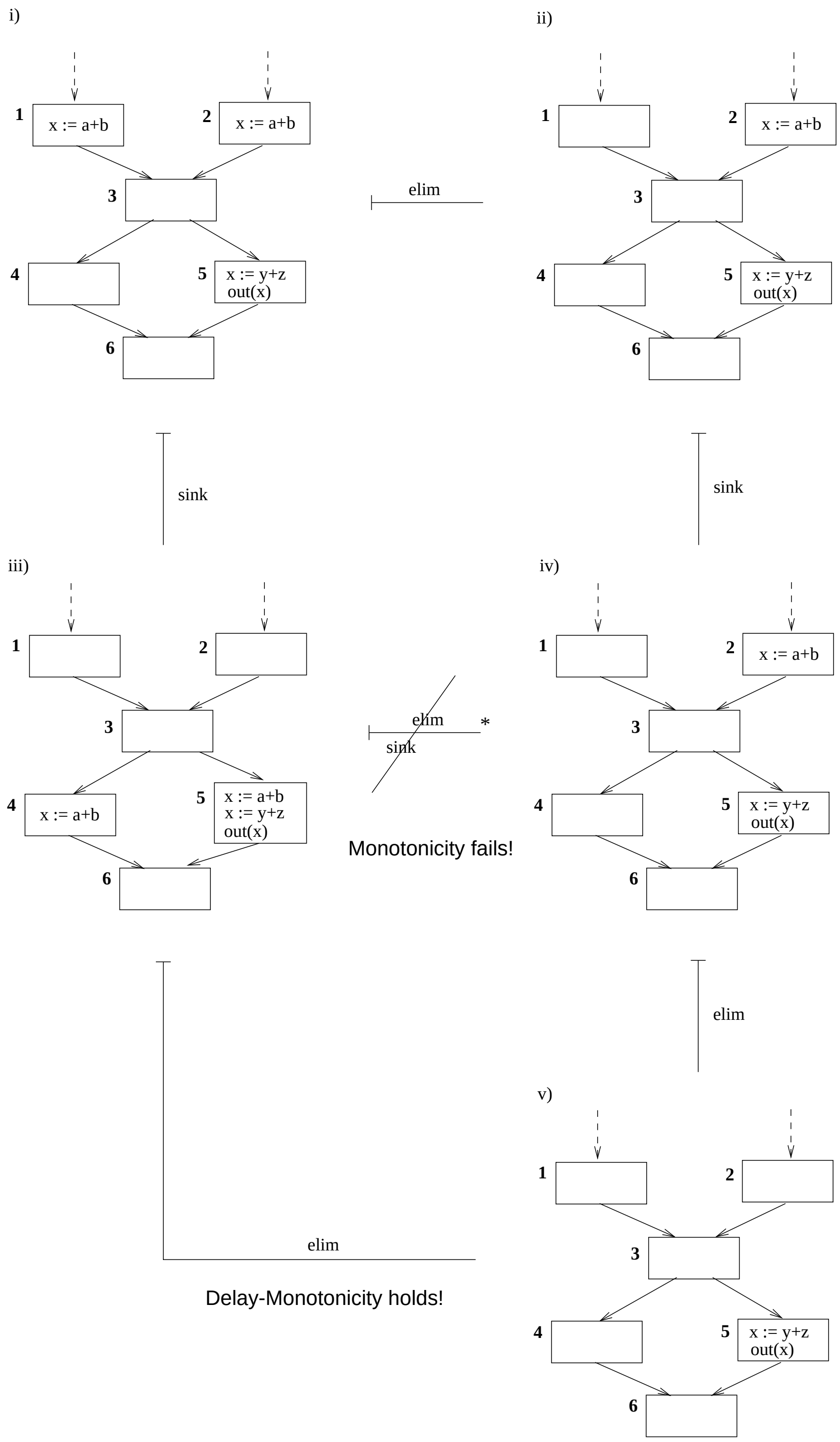

Figure 7: Monotonicity vs. Delay-Monotonicity

Principles of Programming Languages (*POPL'77*) (*Los Angeles, California*), 1977, 238 - 252.

[FKCX] Feigen, L., Klappholz, D., Casazza, R., and Xue, X. The revival transformation. In *Conference Record of the 21st Symposium on Principles of Programming Languages* (*POPL'94*) (*Portland, Oregon*), 1994, 421 - 434.

[DC] Dwyer, M. B., and Clarke, L. A. Data flow analysis for verifying properties of concurrent programs. In *Proceedings of the 2nd ACM SIGSOFT'94 Symposium on Foundations of Software Engineering* (*SIGSOFT'94*) (*New Orleans, Lousiana*), *Software Engineering Notes 19*, 5 (1994), 62 - 75.

[GC] Griebl, M., and Collard, J.-F. Generation of synchronous code for automatic parallelization. In *Proceedings of the 1st European Conference on Parallel Processing* (*Euro-Par'95*) (*Stockholm, Sweden*), Springer-Verlag, Heidelberg, LNCS 966 (1995), 315 - 326.

[GKLRS] Geser, A., Knoop, J., Lüttgen, G., Rüthing, O., and Steffen, B. Non-monotone fixpoint iterations to resolve second order effects. In *Proceedings of the 6th International Conference on Compiler Construction* (*CC'96*) (*Linköping, Sweden*), Springer-Verlag, Heidelberg, LNCS 1060 (1996), 106 - 120.

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

[He] Hecht, M. S. Flow analysis of computer programs. Elsevier, North-Holland, 1977.

[Jo] Jouvelot, P. Semantic parallelization: A practical exercise in abstract interpretation. In *Conference Record of the 14th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL'87)* (*Munich, Germany*), 1987, 39 - 48.

[Kn] Knoop, J. Partial dead code elimination for parallel programs. In *Proceedings of the 2nd European Conference on Parallel Processing* (*EURO-PAR'96*) (*Lyon, France*), Springer-Verlag, Heidelberg, LNCS 1123 (1996), 441 - 450.

[KRS1] Knoop, J., Rüthing, O., and Steffen, B. Lazy code motion. In *Proceedings of the ACM SIGPLAN'92 Conference on Programming Language Design and Implementation* (*PLDI'92*) (*San Francisco, California*), *ACM SIGPLAN Notices 27*, 7 (1992), 224 - 234.

[KRS2] Knoop, J., Rüthing, O., and Steffen, B. Optimal code motion: Theory and practice. *ACM Transactions of Programming Languages and Systems 16*, 4 (1994), 1117 - 1155.

[KRS3] Knoop, J., Rüthing, O., and Steffen, B. Partial dead code elimination. In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation* (*PLDI'94*) (*Orlando, Florida*), *ACM SIGPLAN Notices 29*, 6 (1994), 147 - 158.

[KRS4] Knoop, J., Rüthing, O., and Steffen, B. The power of assignment motion. In *Proceedings of the ACM SIGPLAN'95 Conference of Programming Language Design and Implementation* (*PLDI'95*) (*La Jolla, California*), *ACM SIGPLAN Notices 30*, 6 (1995), 233 - 245.

[KS] Knoop, J., and Steffen, B. The interprocedural coincidence theorem. In *Proceedings of the 4^{th} International Conference on Compiler Construction* (*CC'92*) (*Paderborn, Germany*), Springer-Verlag, Heidelberg, LNCS 641 (1992), 125 - 140.

[KSV1] Knoop, J., Steffen, B., and Vollmer, J. Parallelism for free: Efficient and optimal bitvector analyses for parallel programs. In *ACM Transactions on Programming Languages and Systems 18*, 3 (1996), 268 - 299.

[KSV2] Knoop, J., Steffen, B., and Vollmer, J. Parallelism for free: Bitvector Analyses $\Rightarrow$ No state explosion! In *Proceedings of the 1^{st} International Workshop on Tools and Algorithms for the Construction and Analysis of Systems* (*TACAS'95*) (*Aarhus, Denmark*), Springer-Verlag, Heidelberg, LNCS 1019 (1995), 264 - 289.

[KU] Kam, J. B., and Ullman, J. D. Monotone data flow analysis frameworks. *Acta Informatica 7*, (1977), 309 - 317.

[KY] Krishnamurthy, A., and Yelick, K. Optimizing parallel programs with explicit synchronization. In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation* (*PLDI'95*) (*La Jolla, California*), *ACM SIGPLAN Notices 30*, 6 (1995), 196 - 204.

[La] Lamport, L. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput. C-28*, 9 (1979), 690 - 691.

[Le] Lengauer, Ch. Loop parallelization in the polytope model. In *Proceedings of the 4^{th} International Conference on Concurrency Theory* (*CONCUR'93*) (*Hildesheim, Germany*), Springer-Verlag, Heidelberg, LNCS 715 (1993), 398 - 416.

[LC] Long, D., and Clarke, L. Data flow analysis of concurrent systems that use the rendezvous model of synchronization. In *Proceedings of the ACM SIGSOFT Symposium on Testing, Analysis, and Verification* (*TAV'91*), Software Eng. Notes 16, (1991), 21 - 35.

[LY] Li, Z., and Yew, P.-C., Efficient interprocedural analysis for program parallelization and restructuring. In *Proceedings of the ACM SIGPLAN Conference on Parallel Programming: Experience with Applications, Languages, and Systems* (*PEALS'88*), *ACM SIGPLAN Notices 23*, 9 (1988), 85 - 99.

[Ma] Marriot, K. Frameworks for abstract interpretation. *Acta Informatica 30*, (1993), 103 - 129.

[MJ] Muchnick, S. S., and Jones, N. D. (Eds.). Program flow analysis: Theory and applications. Prentice Hall, Englewood Cliffs, New Jersey, 1981.

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

[Ne] Newman, M. H. A. On theories with a combinatorial definition of equivalence. *Annals of Math. 43*, 2 (1942), 223 - 243.

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

[SP] Sharir, M., and Pnueli, A. Two approaches to interprocedural data flow analysis. In [MJ], 1981, 189 - 233.

[SS] Shasha, D. and Snir, M. Efficient and correct execution of parallel programs that share memory. *Transactions on Programming Languages and Systems 10*, 2 (1988), 282 - 312.

[SW] Srinivasan, H., and Wolfe, M. Analyzing programs with explicit parallelism. In *Proceedings of the 4th International Conference on Languages and Compilers for Parallel Computing* (*LCPC'91*) (*Santa Clara, California*), Springer-Verlag, Heidelberg, LNCS 589 (1991), 405 - 419.

[WS] Wolfe, M, and Srinivasan, H. Data structures for optimizing programs with explicit parallelism. In *Proceedings of the 1st International Conference of the Austrian Center for Parallel Computation*, Springer-Verlag, Heidelberg, LNCS 591 (1991), 139 - 156.

A The Generic Algorithm

Algorithm A.1 (Computing the $PMFP_{BV}$**-Solution)**

Input: *A parallel flow graph* $G^* = (N^*, E^*, \mathbf{s}^*, \mathbf{e}^*)$, *a local semantic functional* $[\![\ ]\!] : N^* \rightarrow \mathcal{F}_\mathcal{B}$, *a function* $f_{init} \in \mathcal{F}_\mathcal{B}$ *and a Boolean value* $b_{init} \in \mathcal{B}$, *where* f_{init} *and* b_{init}

reflect the assumptions on the context in which the procedure under consideration is called. Usually, f_{init} *and* b_{init} *are given by* $Id_{\mathcal{B}}$ *and* $f\!f$*, respectively.*

Output: *An annotation of* G^* *with functions* $[\![\, G \,]\!]^* \in \mathcal{F}_{\mathcal{B}}$, $G \in \mathcal{G}_{\mathcal{P}}(G^*)$*, representing the semantic functions computed in the second step of the three-step procedure of Section 2.2, and with functions* $[\![\, n \,]\!] \in \mathcal{F}_{\mathcal{B}}$, $n \in N^*$*, representing the greatest solution of the equation system of Definition 2.5. After the termination of the algorithm the functional* $[\![\ ]\!]$ *satisfies:*

$$\forall\, n \in N^*.\ [\![\, n \,]\!] = PMFP_{BV\,(G^*,[\![\ ]\!])}(n) = PMOP_{(G^*,[\![\ ]\!])}(n)$$

Remark: *For each flow graph* $G \in \mathcal{G}_{\mathcal{P}}(G^*)$ *we assume a* rank*, which is recursively defined by:*

$$rank(G) =_{df} \begin{cases} 0 & \text{if } G \in \mathcal{G}_{\mathcal{P}}^{min}(G^*) \\ max\{\, rank(G') \mid G' \in \mathcal{G}_{\mathcal{P}}(G^*) \wedge G' \subset G \,\} + 1 & \text{otherwise} \end{cases}$$

where $\mathcal{G}_{\mathcal{P}}^{min}(G^*) =_{df} \{\, G \in \mathcal{G}_{\mathcal{P}}(G^*) \mid \forall\, G' \in \mathcal{G}_{\mathcal{P}}(G^*).\ G' \subseteq G \Rightarrow G' = G \,\}$ *denotes the set of* minimal *graphs of* $\mathcal{G}_{\mathcal{P}}(G^*)$*. Additionally, for every parallel flow graph* G *we define an associated "sequentialized" flow graph* G_{seq}*, which results from* G *by replacing all component flow graphs of "maximal" graphs* $G' \in \mathcal{G}_{\mathcal{P}}^{max}(G)$*, the dual counterpart of* $\mathcal{G}_{\mathcal{P}}^{min}(G)$*, together with all edges starting or ending in such a graph by an edge leading from* $start(G')$ *to* $end(G')$*. Note that* G_{seq} *is free of nested parallel statements: all components of parallel statements are usual nondeterministic sequential flow graphs (cf. [He]).*

Using these definitions, the global variables $[\![\, G_{seq} \,]\!]^* \in \mathcal{F}_{\mathcal{B}}$, $G \in \mathcal{G}_{\mathcal{C}}(G^*)$*, store the global effect of* G *during the hierarchical computation of the* $PMFP_{BV}$*-solution. The global variables* $harmful(G_{seq})$, $G \in \mathcal{G}_{\mathcal{C}}(G^*)$*, store whether* G *contains a node* n *with* $[\![\, n \,]\!] = Const_{f\!f}$*. These variables are used to compute the value of the predicate NonDestructible of Section 2.2.*

BEGIN
  (*Synchronization: Computing* $[\![\, G \,]\!]^*$ *for all* $G \in \mathcal{G}_{\mathcal{P}}(G^*)$)
  $GLOBEFF(G^*, [\![\ ]\!])$;

  (*Interleaving: Computing the* $PMFP_{BV}$*-Solution* $[\![\, n \,]\!]$ *for all* $n \in N^*$)
  $PMFP_{BV}(G^*, [\![\ ]\!], f_{init}, b_{init})$
END.

where

PROCEDURE $GLOBEFF\ (G = (N, E, \mathbf{s}, \mathbf{e}) : ParallelFlowGraph;$
$\overline{[\![\ ]\!]} : N \to \mathcal{F}_{\mathcal{B}}\ : LocalSemanticFunctional);$
VAR $i : integer$;
BEGIN
  FOR $i := 0$ **TO** $rank(G)$ **DO**

FORALL $G' \in \{G'' \mid G'' \in \mathcal{G}_{\mathcal{P}}(G) \wedge rank(G'') = i\}$ **DO**
FORALL $G'' \in \{G'''_{seq} \mid G''' \in \mathcal{G}_{\mathcal{C}}(G')\}$ *where* $G'' = (N'', E'', \mathbf{s}'', \mathbf{e}'')$ **DO**
LET $\forall\, n \in N''.\ [\![\, n \,]\!]'' = \begin{cases} Id_{\mathcal{B}} \sqcap Const_{\forall\, \bar{G} \in \mathcal{G}_{\mathcal{C}}(pfg(n)).\ \neg harmful(\bar{G})} & \text{if } n \in N_N^* \\ [\![\, pfg(n) \,]\!]^* & \text{if } n \in N_X^* \\ \overline{[\![\, n \,]\!]} & \text{otherwise} \end{cases}$
BEGIN
$harmful(G'') := (\ |\{ n \in N'' \mid [\![\, n \,]\!]'' = Const_{ff} \}| \geq 1\ );$
$MFP(G'', [\![\ ]\!]'', Id_{\mathcal{B}});$
$[\![\, G'' \,]\!]^* := [\![\, end(G'') \,]\!]$
END
OD;
$[\![\, G' \,]\!]^* := \bigsqcap_{par} \{ [\![\, G'' \,]\!]^* \mid G'' \in \mathcal{G}_{\mathcal{C}}(G') \}$
OD
OD
END.

PROCEDURE $PMFP_{BV}$ ($G = (N, E, \mathbf{s}, \mathbf{e}) : ParallelFlowGraph;$
$\overline{[\![\ ]\!]} : N \to \mathcal{F}_{\mathcal{B}}$ $: LocalSemanticFunctional;$
$f_{start} : \mathcal{F}_{\mathcal{B}};$
$harmful : \mathcal{B}$);
VAR $f : \mathcal{F}_{\mathcal{B}};$
BEGIN
IF *harmful* **THEN FORALL** $n \in N$ **DO** $[\![\, n \,]\!] := Const_{ff}$ **OD**
ELSE
(*Initialization of the annotation arrays* $[\![\ ]\!]$ *and the variable workset*)
FORALL $n \in Nodes(G_{seq}) \backslash \{\mathbf{s}\}$ **DO** $[\![\, n \,]\!] := Const_{tt}$ **OD**;
$[\![\, \mathbf{s} \,]\!] := f_{start};$
$workset := \{ n \in Nodes(G_{seq}) \mid n \in N_N^* \cup \{\mathbf{s}\} \vee \overline{[\![\, n \,]\!]} = Const_{ff} \};$

(*Iterative fixed point computation*)
WHILE $workset \neq \emptyset$ **DO**
LET $n \in workset$
BEGIN
$workset := workset \backslash \{ n \};$
IF $n \in N \backslash N_N^*$
THEN
$f := \overline{[\![\, n \,]\!]} \circ [\![\, n \,]\!];$
FORALL $m \in succ_G(n)$ **DO**
IF $[\![\, m \,]\!] \sqsupset f$ **THEN** $[\![\, m \,]\!] := f; workset := workset \cup \{ m \}$ **FI**
OD
ELSE
FORALL $G' \in \mathcal{G}_{\mathcal{C}}(pfg(n))$ **DO**
$PMFP_{BV}(G', [\![\ ]\!], [\![\, n \,]\!], \sum_{G'' \in \mathcal{G}_{\mathcal{C}}(pfg(n)) \backslash \{G'\}} harmful(G''))$ **OD**;

```
                    f := [[ pfg(n) ]]* ∘ [[ n ]];
                    IF [[ end(pfg(n)) ]] ⊐ f
                      THEN
                        [[ end(pfg(n)) ]] := f;
                        workset := workset ∪ { end(pfg(n)) } FI
              FI
            END
        OD
    FI
END.
```

```
PROCEDURE MFP (G = (N, E, s, e) : SequentialFlowGraph;
                        [[ ]] : N → F_B  : LocalSemanticFunctional;
                        f_start : F_B);
VAR f : F_B;
BEGIN
    ( Initialization of the annotation array gtr and the variable workset )
    FORALL n ∈ N\{s} DO [[ n ]] := Const_tt OD;
    [[ s ]] := f_start;
    workset := { n | n = s ∨ [[ n ]] = Const_ff };

    ( Iterative fixed point computation )
    WHILE workset ≠ ∅ DO
      LET n ∈ workset
        BEGIN
          workset := workset\{ n };
          f := [[ n ]] ∘ [[ n ]];
          FORALL m ∈ succ_G(n) DO
            IF [[ m ]] ⊐ f THEN [[ m ]] := f; workset := workset ∪ { m } FI
          OD
        END
    OD
END.
```

Let $[\![n]\!]_{alg}$, $n \in N^*$, denote the final values of the corresponding variables after the termination of Algorithm A.1, and $[\![n]\!]$, $n \in N^*$, the greatest solution of the equation system of Definition 2.5, then we have:

Theorem A.2 $\forall\, n \in N^*.\ [\![n]\!]_{alg} = [\![n]\!]$

B Critical Edges

In order to exploit the full power of code motion it is well known that *critical edges*, i.e., edges leading from nodes with more than one successor to nodes with more than one predecessor must be removed in the argument flow graph as illustrated by the simple example below.

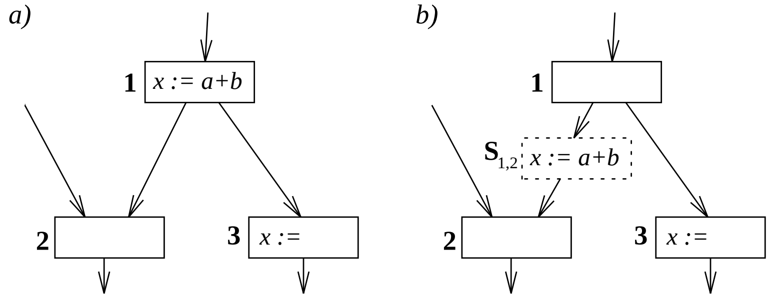

Figure 8: Critical Edges

Note that in Figure 8(a) the assignment $x := a + b$ at node **1** is partially dead with respect to the assignment at node **3**. However, this partially dead assignment cannot safely be eliminated by moving it to its successors, because this may introduce a new assignment on a path entering node **2** on the left branch. On the other hand, it can safely be eliminated after inserting a synthetic node $\mathbf{S}_{1,2}$ in the critical edge $(\mathbf{1}, \mathbf{2})$, as illustrated in Figure 8(b).

In this article we thus assume that all edges in G^* starting in a node with more than one successor, except for edges starting in an entry node of a parallel statement, have been split by inserting a synthetic node. Obviously, this simple transformation guarantees that all critical edges are eliminated. Moreover, it simplifies the process of assignment sinkings as "maximal" sinkings can be obtained by moving all assignments to node entries.