UNIVERSITÄT PASSAU
Fakultät für Mathematik und Informatik

Optimal Code Motion for Parallel Programs

Jens Knoop, Bernhard Steffen
Fakultät für Mathematik und Informatik
Universität Passau
Innstraße 33
D–94032 Passau

Jürgen Vollmer
Institut für Programmstrukturen und Datenorganisation (IPD)
Fakultät für Informatik
Universität Karlsruhe
Vincenz-Prießnitz-Straße 3
D–76128 Karlsruhe

MIP-9511
September 1995

Abstract

Code motion is well-known as a powerful technique for the optimization of sequential programs. It improves the run-time efficiency by avoiding unnecessary recomputations of values, and it is even possible to obtain *computationally optimal* results, i.e., results where no program path can be improved any further by means of semantics preserving code motion. In this paper we present a code motion algorithm that for the first time achieves this optimality result for *parallel* programs. Fundamental is the framework of [KSV1] showing how to perform optimal bitvector analyses for parallel programs as easily and as efficiently as for sequential ones. Moreover, the analyses can easily be adapted from their sequential counterparts. This is demonstrated here by constructing a computationally optimal code motion algorithm for parallel programs by systematically extending its counterpart for sequential programs, the *busy code motion* transformation of [KRS1, KRS2].

Keywords

Parallelism, interleaving semantics, synchronization, compiler, program optimization, data flow analysis, bitvector problems, code motion, partial redundancy elimination, computational optimality.

Contents

1 Motivation

Parallel languages are of growing interest, as they are more and more supported by modern hardware environments. However, despite their importance [SHW, SW, WS], there is currently very little work on classical *analyses* and *optimizations* for parallel programs. In fact, classical optimization and parallelization are often considered to exclude each other because naive adaptations of the sequential optimization methods fail [MP], and their straightforward correct adaptations have unacceptable costs caused by the interleavings which manifest the possible executions of a parallel program.

Thus, either heuristics are proposed to avoid the consideration of all the interleavings [McD], or restricted situations are considered, which do not require to consider the interleavings at all [GS].[1] Completely different are approaches that are based on state space reductions as proposed in [DBDS, CH1, CH2, GW, Va]. This allows general synchronization mechanisms, but still requires the investigation of an appropriately reduced version of the global state space which is often still unmanageable.

In [KSV1, KSV2], however, we have recently shown that for the large class of *bitvector* problems, which are most relevant in practice, there is an elegant way out of this dilemma. We have shown how to construct for unidirectional *bitvector* problems analysis algorithms for parallel programs with shared memory and interleaving semantics that

1. optimally cover the phenomenon of *interference*
2. are as *efficient* as their sequential counterparts and
3. *easy* to implement.

The key for this result was the observation that during unidirectional bitvector analyses the different interleavings of the executions of parallel components need not be considered, although they are semantically different. As a consequence, all the well-known bitvector algorithms for liveness, availability, very business, reaching definitions, definition-use chains (cf. [He]) can easily be adapted for parallel programs at almost no cost on the runtime and the implementation side.

In this paper we exploit this for the construction of a *code motion* algorithm for parallel programs which for the first time achieves computationally optimal placements of computations for this setting. Intuitively, this means that in the program resulting from our algorithm there is no program path, on which the number of computations can be reduced any more by means of semantics preserving code motion. Moreover, this algorithm is as efficient as its sequential counterpart, the *busy code motion* transformation of [KRS1, KRS2]. The power of the new algorithm is illustrated in Figure 1, where the components of parallel statements are separated by parallels. In this example, which is discussed in full detail in Section 3.6, our algorithm is unique to obtain the optimization result of Figure 2. It eliminates the partially redundant computations of $a + b$ at the nodes **3**, **13**, **15**, **18**, **19**, **28**, **29**, **39** by moving them to the nodes **2**, **14** and **26**, but it does not touch the partially redundant computations of $a + b$ at the nodes **8** and **10**, which cannot safely be eliminated.

[1]In [GS] e.g., this is achieved by requiring data independence of the parallel components according to the PCF FORTRAN standard.

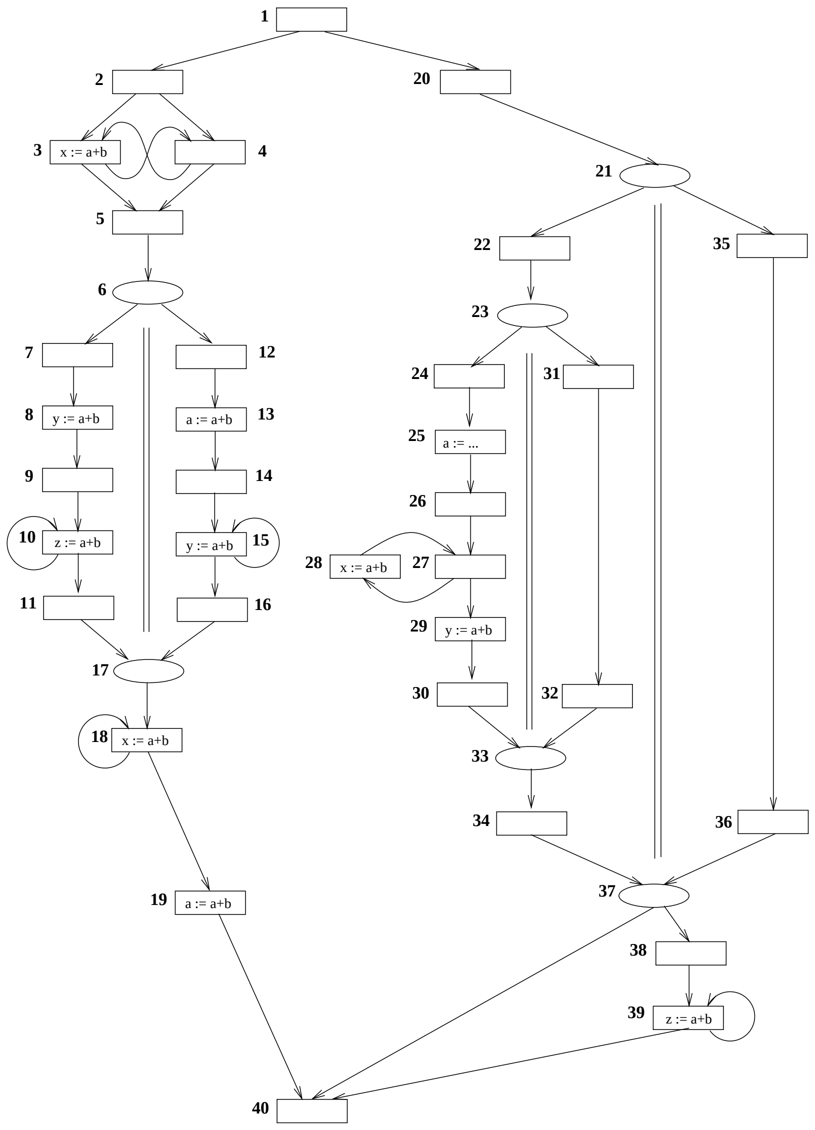

Figure 1: The Motivating Example: The Parallel Argument Program G^*

Structure of the Paper

In Section 2 we recall the framework of [KSV1] for unidirectional bitvector analyses of parallel programs. Based on this framework, we subsequently develop our algorithm for the computationally optimal placement of computations in parallel programs in Section 3. Section 4, finally, contains our conclusions, and the Appendix presents the detailed generic algorithm of [KSV1] for unidirectional bitvector analyses of parallel programs.

2 The Parallel Setting

We consider a parallel imperative programming language with interleaving semantics. Parallelism is syntactically expressed by means of a `par` statement whose components are assumed to be executed in parallel on a shared memory. As usual, we assume that there are neither jumps leading into a component of a `par` statement from outside nor vice versa. This setup already introduces the phenomena of interference and synchronization,

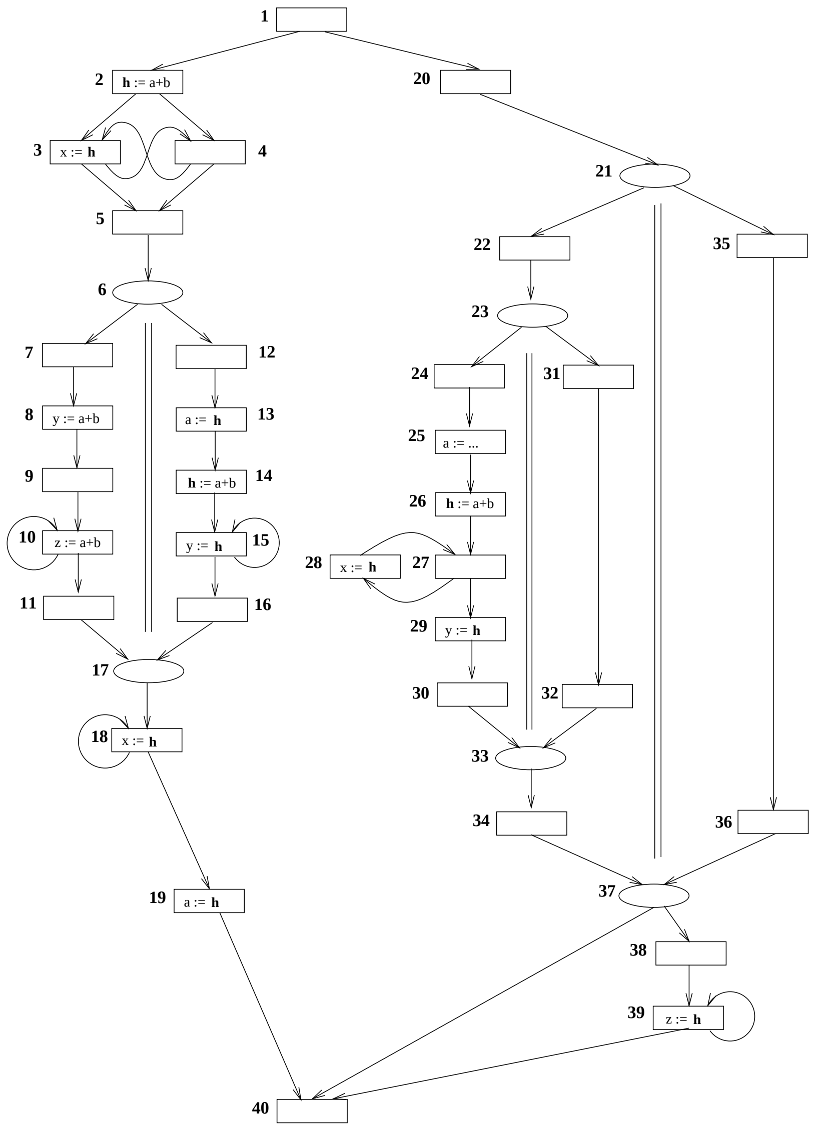

Figure 2: The Computationally Optimal Result of the BCM_{PP}-Transformation

and allows us to concentrate on the central features of our approach, which, however, is not limited to this setting. For example, a replicator statement allowing a dynamical process creation can be integrated along the lines of [CH2, Vo1, Vo2].

2.1 Parallel Flow Graphs

Similarly 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 1. Except for subgraphs representing `par` statements a parallel flow graph is a nondeterministic flow graph as for the representation of a sequential program (cf. [He]). Thus, nodes $n \in N^*$ represent the statements, edges $(m, n) \in E^*$ the nondeterministic branching structure of the procedure under consideration, and $\mathbf{s}^*$ and $\mathbf{e}^*$ denote the distinct *start node* and *end node*, which are assumed to represent the empty statement `skip` and to possess no predecessors and successors, respectively.

A `par` statement and each of its components are also considered parallel flow graphs.

The graph G_{par} representing a complete par statement arises from linking its component graphs by means of a ParBegin and a ParEnd node which have the start nodes and the end nodes of the component graphs as their only successors and predecessors, respectively. The ParBegin node and the ParEnd node are the unique start node and end node of G_{par}, and are assumed to represent the empty statement skip. They form the entry and the exit to program regions whose subgraph components are assumed to be executed in parallel, and thus make the synchronization points in the program explicit. For clarity we represent ParBegin and ParEnd nodes by ellipses and additionally separate the corresponding component graphs by two parallels as shown in Figure 1.

Moreover, $pred_{G^*}(n) =_{df} \{ m \mid (m,n) \in E^* \}$ and $succ_{G^*}(n) =_{df} \{ m \mid (n,m) \in E^* \}$ denote the set of all immediate predecessors and successors of a node $n \in N^*$, respectively. A sequence $(n_1, \ldots, n_q)$ of nodes such that $(n_j, n_{j+1}) \in E^*$ for $j \in \{1, \ldots, q-1\}$ is called a *finite path* of G^*. Given a finite path p, λ_p denotes the *length* of p. Moreover, $\mathbf{P}_{G^*}[m,n]$ denotes the set of all finite paths from m to n, and $\mathbf{P}_{G^*}[m,n[$ the set of all finite paths from m to a predecessor of n. As usual, we assume that every node $n \in N^*$ lies on a finite path from $\mathbf{s}^*$ to $\mathbf{e}^*$. It is worth noting that not all finite paths of G^* represent a proper program execution. This is taken into account by restricting to *parallel paths*, which are introduced in Definition 2.2 below.

Additionally, $\mathcal{G}_{\mathcal{P}}(G^*)$ denotes the set of all subgraphs of G^* representing a par statement. In particular,

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

and

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

denote the set of *maximal* and *minimal* graphs of $\mathcal{G}_{\mathcal{P}}(G^*)$.[2] Moreover, every flow graph $G \in \mathcal{G}_{\mathcal{P}}(G^*)$ is given a *rank* that is recursively defined by:

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

For illustration see Figure 3 and Figure 4, which display the set of parallel subgraphs of rank 1 and of rank 0 of the parallel flow graph of Figure 1.

Furthermore, for $G' \in \mathcal{G}_{\mathcal{P}}(G^*)$, $\mathcal{G}_{\mathcal{C}}(G')$ denotes the set of component flow graphs of G', which we also consider parallel flow graphs for notational convenience. Thus, all subgraphs of G^* referred to in the paper are considered parallel flow graphs, but only the subgraphs $G \in \mathcal{G}_{\mathcal{P}}(G^*)$ represent parallel statements. It is worth noting that for $G \in \mathcal{G}_{\mathcal{P}}(G^*)$ every component flow graph $G' \in \mathcal{G}_{\mathcal{C}}(G)$ and also G itself is a single-entry/single-exit region of G^*.

Moreover, for $G' \in \mathcal{G}_{\mathcal{P}}(G^*)$, $CpNodes(G') =_{df} N' \backslash \{\mathbf{s}', \mathbf{e}'\}$ denotes the set of nodes of its component flow graphs.[3] Addititionally, we introduce the following abbreviations for the sets of start nodes (i.e., ParBegin nodes) and end nodes (i.e., ParEnd nodes) of graphs of $\mathcal{G}_{\mathcal{P}}(G^*)$:

[2]For parallel flow graphs G and G' we define: $G \subseteq G'$ if and only if $N \subseteq N'$ and $E \subseteq E'$.

[3]We use the convention that the node set and the edge set, and the start node and the end node of a flow graph carry the same marking as the flow graph itself. Hence, G and G' stand for the expanded versions $G = (N, E, \mathbf{s}, \mathbf{e})$ and $G' = (N', E', \mathbf{s}', \mathbf{e}')$, respectively.

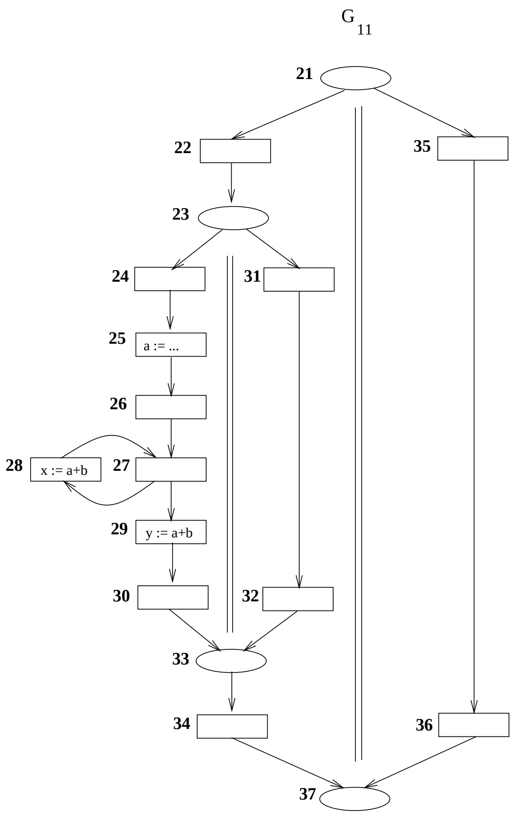

Figure 3: $\{G \mid G \in \mathcal{G}_{\mathcal{P}}(G^*) \wedge \mathit{rank}(G)=1\} \;=\; \{G_{11}\}$

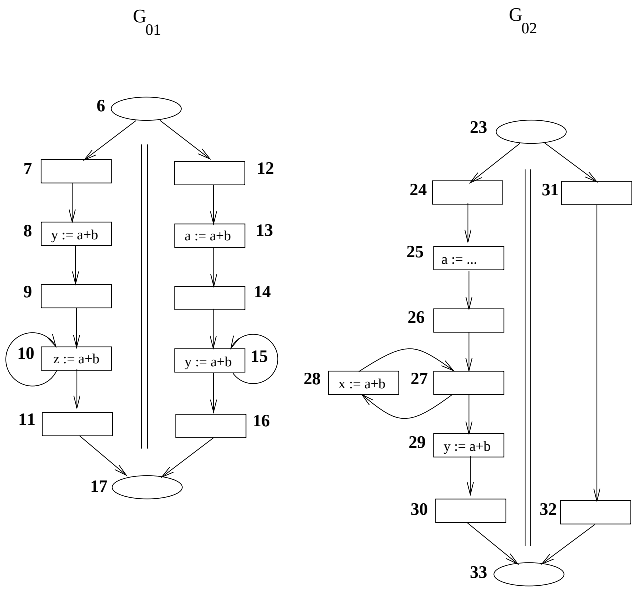

Figure 4: $\{G \mid G \in \mathcal{G}_{\mathcal{P}}(G^*) \wedge \mathit{rank}(G)=0\} \;=\; \{G_{01}, G_{02}\}$

$$N^*_N =_{df} \{\, \mathit{start}(G) \mid G \in \mathcal{G}_{\mathcal{P}}(G^*)\,\} \quad \text{and} \quad N^*_X =_{df} \{\, \mathit{end}(G) \mid G \in \mathcal{G}_{\mathcal{P}}(G^*)\,\}$$

where *start* and *end* are functions, which map a flow graph to its start node and end node, respectively. Additionally, we need the function *Nodes*, which maps a flow graph to its node set, and two functions *pfg* and *cfg*. The first function, *pfg*, maps a node n occurring in some flow graph $G' \in \mathcal{G}_{\mathcal{P}}(G^*)$ to the smallest flow graph of $\mathcal{G}_{\mathcal{P}}(G^*)$

containing n; and it maps the remaining nodes n of N^* to G^*, i.e.,

$$pfg(n) =_{df} \begin{cases} \bigcap\{\, G' \in \mathcal{G}_{\mathcal{P}}(G^*) \mid n \in Nodes(G') \,\} & \text{if } n \in Nodes(\mathcal{G}_{\mathcal{P}}(G^*)) \\ G^* & \text{otherwise} \end{cases}$$

Similarly, cfg maps a node n occurring in a component flow graph of some graph $G \in \mathcal{G}_{\mathcal{P}}(G^*)$ to the smallest component flow graph containing n; and it maps the remaining nodes n of N^* to G^*, i.e.,

$$cfg(n) =_{df} \begin{cases} \bigcap\{\, G' \in \mathcal{G}_{\mathcal{C}}(\mathcal{G}_{\mathcal{P}}(G^*)) \mid n \in Nodes(G') \,\} & \text{if } n \in CpNodes(\mathcal{G}_{\mathcal{P}}(G^*)) \\ G^* & \text{otherwise} \end{cases}$$

Both pfg and cfg are well-defined, since `par` statements in a program are either unrelated or properly nested.

Finally, for each parallel flow graph G we define an associated 'sequentialized' flow graph G_{seq}, which results from G by replacing all nodes belonging to a component flow graph of a graph $G' \in \mathcal{G}_{\mathcal{P}}^{max}(G)$ together with all edges starting or ending in such a node 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 standard nondeterministic sequential flow graphs (cf. [He]). This is illustrated in Figure 5 and Figure 6, which show the sequentialized versions of the parallel flow graphs of Figure 1 and Figure 3, respectively.

Interleaving Predecessors

For a sequential flow graph G, the set of nodes that might dynamically precede a node n is precisely given by the set of its static predecessors $pred_G(n)$. For a parallel flow graph, however, the interleaving of statements of parallel components must be taken into account. Here, nodes n occurring in a component of some `par` statement can dynamically be preceded also by nodes, whose execution may be interleaved with that of n. For example, in the program of Figure 1 the execution of node **26**, whose only static predecessor is node **25**, may be interleaved with the execution of the nodes **31**, **32**, **35**, and **36**. We denote these 'potentially parallel' nodes as *interleaving predecessors*. The set of all interleaving predecessors of a node $n \in N^*$ is recursively defined by means of the function $Pred_{G^*}^{Itlvg} : N^* \to \mathcal{P}(N^*)$, where $\mathcal{P}$ denotes the power set operator and $mpe\text{-}pfg$ a function, which maps a node $n \in N^*$ to its *minimal properly enclosing* graph of $\mathcal{G}_{\mathcal{P}}(G^*) \cup \{G^*\}$:

$$Pred_{G^*}^{Itlvg}(n) =_{df} \begin{cases} \emptyset & \text{if } n \in N^* \backslash CpNodes(\mathcal{G}_{\mathcal{P}}(G^*)) \\ \\ CpNodes(mpe\text{-}pfg(n)) \backslash Nodes(cfg(n)) \cup \\ Pred_{G^*}^{Itlvg}(start(cfg(start(mpe\text{-}pfg(n))))) & \text{otherwise} \end{cases}$$

where $mpe\text{-}pfg$ is defined by:

$$mpe\text{-}pfg(n) =_{df} \begin{cases} pfg(start(cfg(n))) & \text{if } n \in N_N^* \cup N_X^* \\ \\ pfg(n) & \text{otherwise} \end{cases}$$

This is illustrated in Figure 7, which shows the sets of static and interleaving predecessors of the nodes **14** and **32** of Figure 1. We have:

$$pred_{G^*}(\mathbf{14}) = \{\mathbf{13}\} \quad \text{and} \quad Pred_{G^*}^{Itlvg}(\mathbf{14}) = \{\mathbf{7}, \ldots, \mathbf{11}\}$$

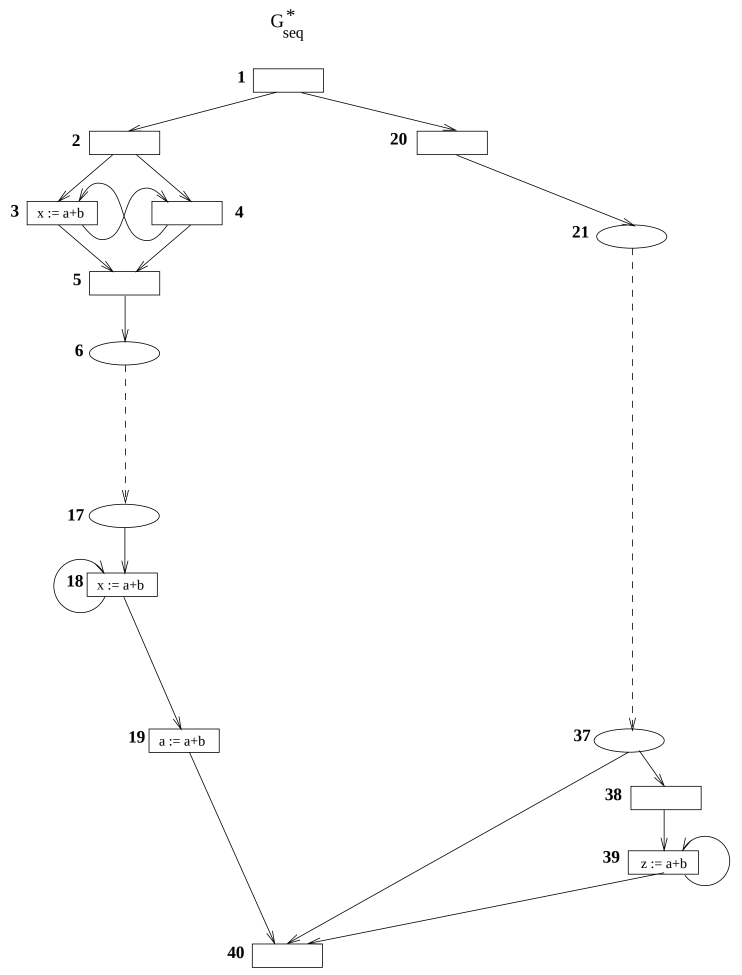

Figure 5: The Sequentialized Version G^*_{seq} of G^*

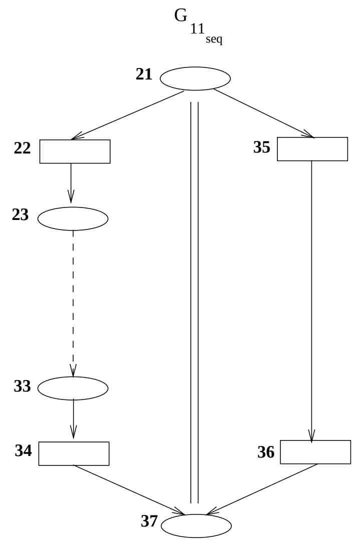

Figure 6: $\{G_{seq} \,|\, G \in \mathcal{G}_{\mathcal{P}}(G^*) \wedge rank(G) = 1\} \;=\; \{G_{11_{seq}}\}$

and

$$pred_{G^*}(\mathbf{32}) = \{\mathbf{31}\} \quad \text{and} \quad Pred^{Itlvg}_{G^*}(\mathbf{32}) = \{\mathbf{24}, \ldots, \mathbf{30}, \mathbf{35}, \mathbf{36}\}$$

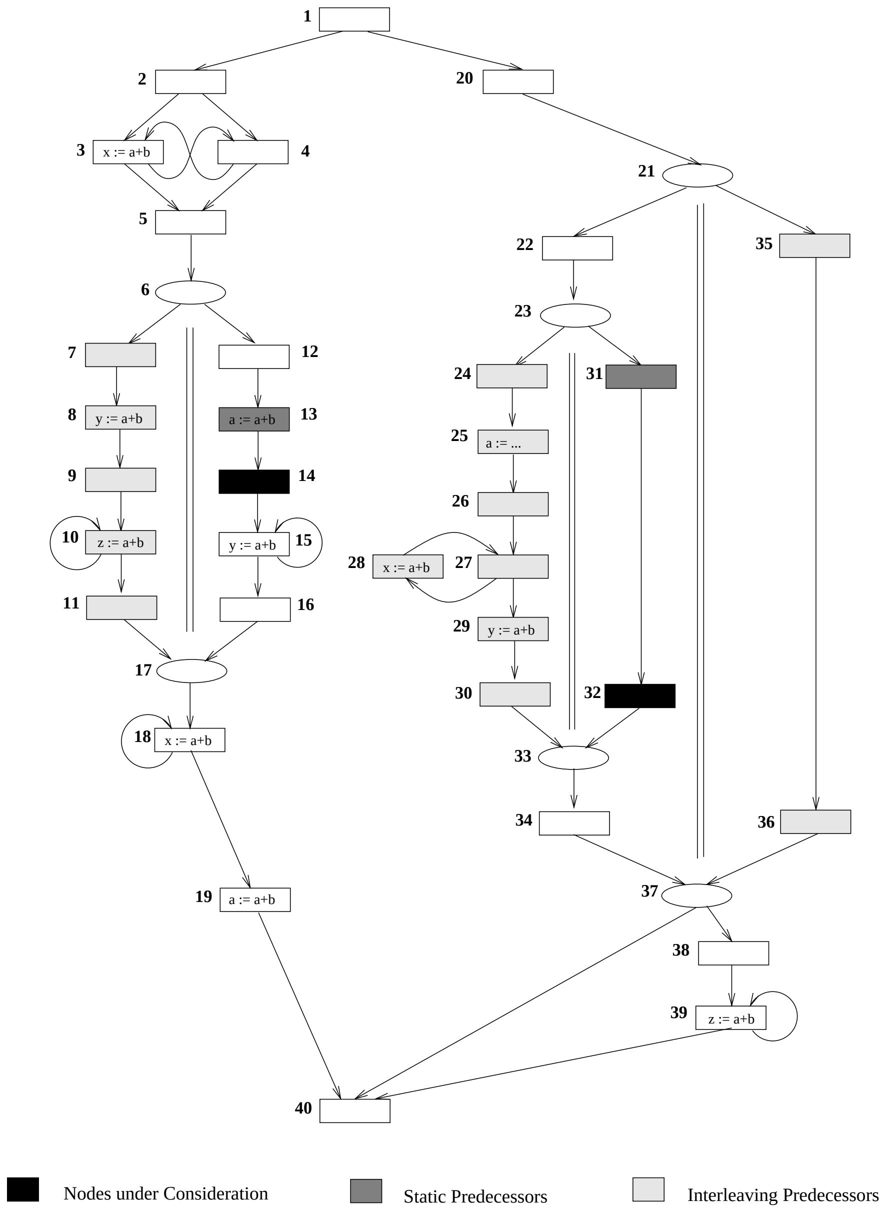

Figure 7: Static and Interleaving Predecessors

Program Paths of Parallel Programs

It is well-known that the interleaving semantics of a parallel imperative programming language can be defined via a translation that reduces parallel programs to (much larger) nondeterministic programs. In this section, we recall the alternative view of [KSV1] to characterize the node sequences constituting a parallel (program) path, which in spirit follows the definition of an interprocedural program path as proposed by Sharir and Pnueli [SP]. They start by interpreting every branch statement purely nondeterministically, which allows to simply use the definition of *finite path* as introduced in Section 2.1. This results in a superset of the set of all interprocedurally valid paths, which they then refine by means of an additional consistency condition. In our case, we are forced to define our consistency condition on arbitrary node sequences, as the consideration of interleavings invalidates the first step. Here, the following notion of well-formedness is important.

Definition 2.1 (G-Well-Formedness)
*Let G be a (parallel) flow graph, and $p =_{df} (n_1, \ldots, n_q)$ be a sequence of nodes. Then p is G-*well-formed *if and only if*

1. *the projection $p{\downarrow}_{G_{seq}}$ of p onto G_{seq} lies in $\mathbf{P}_{G_{seq}}[start(G_{seq}), end(G_{seq})]$*
2. *for all node occurrences $n_i \in N_N^*$ of the sequence p there exists a $j \in \{i+1, \ldots, q\}$ such that*
   (a) *$n_j \in N_X^*$,*
   (b) *n_j is the successor of n_i on $p{\downarrow}_{G_{seq}}$ and*
   (c) *the sequence $(n_{i+1}, \ldots, n_{j-1})$ is G'-well-formed for all $G' \in \mathcal{G}_{\mathcal{C}}(pfg(n_i))$.*

Now the set of parallel paths is defined as follows.

Definition 2.2 (Parallel Path)
Let $G^* = (N^*, E^*, \mathbf{s}^*, \mathbf{e}^*)$ be a parallel flow graph, and $p =_{df} (n_1, \ldots, n_q)$ be a sequence of nodes of N^*. Then:

1. *p is a* parallel path from $\mathbf{s}^*$ to $\mathbf{e}^*$ *if and only if p is G^*-well-formed.*
2. *p is a* parallel path from n_1 to n_q *if it is a subpath of some parallel path from $\mathbf{s}^*$ to $\mathbf{e}^*$.*

$\mathbf{PP}_{G^*}[m, n]$ *denotes the set of all parallel paths from m to n, and* $\mathbf{PP}_{G^*}[m, n[$ *the set of all parallel paths from m to a (static or interleaving) predecessor of n, defined by*

$$\mathbf{PP}_{G^*}[m, n[=_{df} \{(n_1, \ldots, n_q) \mid (n_1, \ldots, n_q, n_{q+1}) \in \mathbf{PP}_{G^*}[m, n]\}$$

2.2 Data Flow Analysis

Data flow analysis (DFA) is the prerequisite of almost any performance improving program transformation used by 'optimizing' compilers to generate efficient object code (cf. [He, MJ]). For imperative languages, DFA provides information about the program states that may occur at some given program points during execution. Theoretically well-founded are DFAs that are based on *abstract interpretation* (cf. [CC, Ma]). The point of this approach is to replace the "full" semantics by a simpler more abstract version, which is 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 every program statement in terms of a transformation function on a complete lattice $(\mathcal{C}, \sqcap, \sqsubseteq, \perp, \top)$ with least element $\perp$ and greatest element $\top$, whose elements express the DFA-information of interest.[4] 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 $\mathcal{C}$.

[4]In the following $\mathcal{C}$ will always denote a complete lattice.

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. For every path $p = (n_1, \ldots, n_q) \in \mathbf{PP}_{G^*}[m, n]$, we define:

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

where $Id_{\mathcal{C}}$ denotes the identity on $\mathcal{C}$. 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 under consideration.

The *PMOP*-Solution:

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

Obviously, this directly reflects our desires, but is in general not effective. However, as we will see in the next section, for bitvector problems there exists an elegant and efficient way for computing the *PMOP*-solution by means of a fixed point computation. We therefore recall the essentials of bitvector analyses allowing our efficient fixed point approach.

2.3 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 (see Section 3.5 for illustration). Here $\mathcal{B}$ 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 $\sqcap$.

Despite their simplicity, unidirectional bitvector problems are highly relevant in practice because of their broad scope of applications ranging from simple analyses like liveness, availability, very business, reaching definitions, and definition-use chains to more sophisticated and powerful program optimizations like code motion, strength reduction, partial dead code elimination, and assignment motion.

Next we are going to show, how to optimize the effort for computing the *PMOP*-solution of bitvector problems. This requires the consideration of the semantic domain $\mathcal{F}_{\mathcal{B}}$ consisting of the monotonic Boolean functions $\mathcal{B} \to \mathcal{B}$. Obviously, we have:

Proposition 2.3 *1. $\mathcal{F}_{\mathcal{B}}$ simply consists of the constant functions $Const_{tt}$ and $Const_{ff}$, together with the identity $Id_{\mathcal{B}}$ on $\mathcal{B}$.*

2. $\mathcal{F}_{\mathcal{B}}$, together with the pointwise ordering between functions, forms a complete lattice with least element $Const_{ff}$ and greatest element $Const_{tt}$, which is closed under function composition.

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

The key to the efficient computation of the 'interleaving effect' is based on the following simple observation, which pinpoints the specific nature of a domain of functions that only consists of constant functions and the identity on an arbitrary set M.

Lemma 2.4 (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 from $\mathcal{F}_{\mathcal{B}}$ to $\mathcal{F}_{\mathcal{B}}$. Then we have:

$$\exists\, k \in \{1, \ldots, q\}.\ f_q \circ \ldots \circ f_2 \circ f_1 = f_k \ \wedge\ \forall\, j \in \{k+1, \ldots, q\}.\ f_j = Id_{\mathcal{B}}$$

Interference

The relevance of Main Lemma 2.4 for bitvector problems is that it restricts the way of possible interference within a parallel program: each possible interference is due to a single statement within a parallel component. 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 $Pred^{Itlvg}_{G^*}(n)$. 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 is reflected in the definition of the following predicate:

$NonDestructed : N^* \to \mathcal{B}$ defined by

$$\forall\, n \in N^*.\ NonDestructed(n) =_{df} \bigwedge \{ [\![\, m \,]\!](tt) \mid m \in Pred^{Itlvg}_{G^*}(n) \}$$

Intuitively, $NonDestructed(n) = tt$ indicates that no node of a parallel component destroys the property under consideration, i.e. $[\![\, m \,]\!] \neq Const_{ff}$ for all $m \in Pred^{Itlvg}_{G^*}(n)$. Note that only the constant function given by the precomputed value of this predicate is used in Definition 2.7 to model interference, and in fact, Theorem 2.8 guarantees that this modelling is sufficient. Obviously, this predicate is easily and efficiently computable. Algorithm A.1 computes it 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: in order to leave a parallel statement, all parallel components are required to terminate. The information that 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 analysis [KS]: we need to compute the effect of complete subgraphs, or 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 is given below:

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

any parallel statement, and determine the effect $[\![G']\!]$ of this (purely sequential) graph according to the equational system[5]

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

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

$$[\![G'']\!]^* = \begin{cases} Const_{ff} & \text{if } \exists\, G' \in \mathcal{G}_{\mathcal{C}}(G'').\ [\![end(G')]\!] = Const_{ff} \\ Id_{\mathcal{B}} & \text{if } \forall\, G' \in \mathcal{G}_{\mathcal{C}}(G'').\ [\![end(G')]\!] = Id_{\mathcal{B}} \\ Const_{tt} & \text{otherwise} \end{cases}$$

3. Transform G by replacing all innermost parallel statements $G'' = (N'', E'', \mathbf{s}'', \mathbf{e}'')$ by $(\{\mathbf{s}'', \mathbf{e}''\}, \{(\mathbf{s}'', \mathbf{e}'')\}, \mathbf{s}'', \mathbf{e}'')$, and replace the local semantics of $\mathbf{s}''$ and $\mathbf{e}''$ by $Id_{\mathcal{B}} \sqcap \bigsqcap\{ [\![n]\!] \mid n \in N''\}$ and $[\![G'']\!]^*$, respectively. Continue with step 1.

In essence, this three-step algorithm 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]). Details can be found in [KSV1]. Here we only consider the second step realizing the synchronization at nodes in N_X^* in more detail. Central is the following lemma, which can be proved by means of Main Lemma 2.4.

Lemma 2.5 *The PMOP-solution of a parallel flow graph G that only consists of purely sequential parallel components $G_1, \ldots, G_k$ is given by:*

$$PMOP_{(G,[\![\ ]\!])}(end(G)) = \begin{cases} Const_{ff} & \text{if } \exists\, 1 \le i \le k.\ [\![end(G_i)]\!] = Const_{ff} \\ Id_{\mathcal{B}} & \text{if } \forall\, 1 \le i \le k.\ [\![end(G_i)]\!] = Id_{\mathcal{B}} \\ Const_{tt} & \text{otherwise} \end{cases}$$

The point here 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 some path through one of the parallel components (the one 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. By means of this fact, which is formalized in Lemma 2.5, we can prove (cf. [KSV1]):

Theorem 2.6 (The 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:

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

After this hierarchical preprocess the following equation system is the key for characterizing the *PMOP*-solution of a unidirectional bitvector problem algorithmically:

[5]Note that the local semantic functional $[\![\ ]\!]'$ of G' is known, whenever this step is executed.

Definition 2.7 *The functional* $[\![\ ]\!] : N^* \to \mathcal{F}_{\mathcal{B}}$ *is defined as the greatest solution of the equation system given by:*

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

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

The $PMFP_{BV}$-Solution:

$PMFP_{BV(G^*,[\![\ ]\!])} : N^* \to \mathcal{F}_{\mathcal{B}}$ defined by $\forall\, n \in N^*\ \forall\, b \in \mathcal{B}.\ PMFP_{BV(G^*,[\![\ ]\!])}(n)(b) = [\![\, n \,]\!](b)$

As for the sequential setting, the practical relevance of the $PMFP_{BV}$-solution stems from the fact that it can efficiently be computed (a generic Algorithm A.1 is given in Appendix A). Moreover, it coincides with the desired *PMOP*-solution (cf. [KSV1]).

Theorem 2.8 (The Parallel Bitvector Coincidence Theorem)
Let $G^* = (N^*, E^*, \mathbf{s}^*, \mathbf{e}^*)$ *be a parallel flow graph, and* $[\![\ ]\!] : N^* \to \mathcal{F}_{\mathcal{B}}$ *a local semantic functional. Then we have that the PMOP-solution and the* $PMFP_{BV}$*-solution coincide, i.e.,*

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

3 Optimal Code Motion

In this section we demonstrate the practicality and elegance of the framework of Section 2 by constructing a *code motion* algorithm for parallel programs, which places the computations of a parallel program computationally optimally. This algorithm, which is unique in achieving this optimality result, evolves from extending the *busy code motion (BCM)* transformation of [KRS1, KRS2] to the parallel setting.

Intuitively, code motion improves the efficiency of a program by avoiding unnecessary recomputations of values at runtime. This is achieved by replacing the original computations of a program by temporaries that are initialized at suitable program points. In order to preserve the semantics of the argument program, code motion must be *admissible*, i.e., it must be *correct* and *safe*. Intuitively, 'correct' means that the temporaries are properly initialized, i.e., they always represent the same value as the computation they replace; 'safe' means that no computations of new values on paths are introduced. For sequential programs it is well-known that under these requirements computationally optimal results can be obtained, i.e., results where the number of computations on each program path cannot be reduced anymore by means of admissible code motion. The central idea to obtain computational optimality is to place computations *as early as possible* in a program, while maintaining admissibility (cf. [Dh1, Dh2, DS1, DS2, DRZ, KRS1, KRS2, MR]). As we are going to show here, the same strategy applies to parallel programs (cf. Theorem

3.5). Moreover, the transformation we develop has the same simple structure as its underlying algorithm for sequential programs: Like the BCM-transformation, it is composed of only two uni-directional bitvector data flow analyses.

Local Predicates

The definition of the BCM_{PP}-transformation is based on two local predicates *Transp* and *Comp* defined for every node $n \in N^*$. Intuitively, they indicate whether t is modified or computed by the assignment of node n, respectively.[6]

- *Transp*(n): n is *transparent* for t, i.e., n does not modify an operand of t.
- *Comp*(n): n is a *computation* of t, i.e., n contains an occurrence of t.

'Recursive' assignments: Assignments whose left hand side variable occurs in its right hand side term t use *and* modify t, a property, which cannot be distinguished from a simple use in our abstract domain. While this distinction is unnecessary in the sequential setting, it is vital in the parallel setting because of interference. We therefore consider recursive assignments $x := t$ in parallel statements implicitely as being decomposed into sequences $x_t := t$; $x := x_t$, where x_t is a new variable: rather than changing the argument program, this implicit decomposition is realized by associating two semantic functions with recursive assignments (in parallel components) (cf. Section 3.5).

Conventions

In order to obtain concise notations we introduce the following abbreviations. Given a parallel path $p = (n_1, \ldots, n_q)$ of G^* and an index $1 \leq i \leq \lambda_p$, p_i denotes the i-th component of p. Additionally, $p[i,j]$ and $p[i,j[$, i, $j \leq \lambda_p$, denote the subpaths $(n_i, \ldots, n_j)$ and $(n_i, \ldots, n_{j-1})$ of p, respectively. Moreover, if *Predicate* is a predicate defined on nodes and p is a path, we define:

- $\mathit{Predicate}^{\forall}(p) \iff_{df} \forall\, 1 \leq i \leq \lambda_p.\ \mathit{Predicate}(p_i)$
- $\mathit{Predicate}^{\exists}(p) \iff_{df} \exists\, 1 \leq i \leq \lambda_p.\ \mathit{Predicate}(p_i)$

Note that the formulas $\neg \mathit{Predicate}^{\forall}(p)$ and $\neg \mathit{Predicate}^{\exists}(p)$ are then abbreviations of the formulas $\exists\, 1 \leq i \leq \lambda_p.\ \neg \mathit{Predicate}(p_i)$ and $\forall\, 1 \leq i \leq \lambda_p.\ \neg \mathit{Predicate}(p_i)$, respectively.

3.1 Code Motion

Intuitively, a code motion transformation CM for a fixed program term t is characterized by the following three-step procedure: (1) Declare a new temporary $\mathbf{h}$ in the program G^* under consideration, (2) insert assignments of the form $\mathbf{h} := t$ at some nodes in G^*, and (3) replace some of the original computations of t in G^* by $\mathbf{h}$.

As the declaration of the temporary is common to all code motion transformations, a code motion transformation CM is completely specified by two predicates Insert_{CM} and $\mathit{Replace}_{CM}$, which denote the set of program points where an initialization must be

[6]In [MR] the predicate *Comp* is called *Antloc*.

inserted and an original computation must be replaced. Without loss of generality, we assume that $Replace_{CM}$ implies $Comp$. In the following we denote the set of all code motion transformations with respect to t by $\mathcal{CM}$. Moreover, as in [KRS1] we assume that all edges leading to a node outside N^*_X with more than one predecessor has been split by inserting a synthetic node. Edge splitting is typical for code motion transformations (cf. [Dh1, Dh2, DS1, KRS1, KRS2]) in order avoid the blocking of the code motion process by *critical edges* as illustrated in Appendix B.

3.2 Admissible Code Motion

As mentioned already, an *admissible* code motion transformation CM preserves the semantics of the argument program, which requires that CM is *safe* and *correct*. 'Safe' means that on no program path the computation of a new value is introduced by inserting a computation of t; 'correct' means that every replacement of an original computation of t by $\mathbf{h}$ is *proper*, i.e., that $\mathbf{h}$ always represents the same value as t. Formally, two computations of t represent the *same value* on a path if and only if no operand of t is modified between them. This is reflected in the following definition, which defines when inserting and replacing a computation of t is *safe* and *correct* in a node $n \in N^*$, respectively.

Definition 3.1 (Safety and Correctness)
For all nodes $n \in N^*$ we define:

1. $Safe(n) \iff_{df} Up\text{-}Safe(n) \vee Down\text{-}Safe(n)$, *where*
   (a) $Up\text{-}Safe(n) \iff_{df}$
   $\forall\, p \in \mathbf{PP}_{G^*}[\mathbf{s}^*, n]\ \exists\, i < \lambda_p.\ Comp\,(p_i) \wedge Transp^{\forall}(\,p[i, \lambda_p[\,)$
   (b) $Down\text{-}Safe(n) \iff_{df}$
   $\forall\, m \in \{n\} \cup Pred^{Itlvg}_{G^*}(n)\ \forall\, p \in \mathbf{PP}_{G^*}[m, \mathbf{e}^*]\ \exists\, i \leq \lambda_p.\ Comp\,(p_i) \wedge Transp^{\forall}(\,p[1, i[\,)$
2. *Let $CM \in \mathcal{CM}$. Then:*
   $Correct_{CM}(n) \iff_{df} \forall\, p \in \mathbf{PP}_{G^*}[\mathbf{s}^*, n]\ \exists\, i \leq \lambda_p.\ Insert_{CM}(p_i) \wedge Transp^{\forall}(\,p[i, \lambda_p[\,)$

The predicate for safety is the disjunction of the predicates for up-safety and down-safety. Intuitively, a node n is *up-safe* at its entry, if on every program path starting in the start node $\mathbf{s}^*$ reaching n there is a computation of t which is not followed by a modification of t. Analogously, a node n is *down-safe* at its entry, if on every program path starting in n or in a node whose execution may be interleaved with that of n, reaching the end node $\mathbf{e}^*$ there is a computation of t, which is not preceded by a modification of t.[7] Intuitively, the replacement of a computation of t by $\mathbf{h}$ is *correct* at a node n, if $\mathbf{h}$ and t represent the same value at n, i.e., if every path from $\mathbf{s}^*$ to n goes through an initialization site of $\mathbf{h}$ which is not followed by a modification of t.

The predicates for safety and correctness are important because they directly induce the class of *admissible* code motion transformations which are guaranteed to preserve the semantics of the argument program.

[7]Up-safety and down-safety are often called *availability* and *anticipability*, respectively (cf. [MR]).

Definition 3.2 (Admissible Code Motion)
A code motion transformation $CM \in \mathcal{CM}$ is admissible *if and only if for each $n \in N^*$ the following two conditions hold:*

1. $Insert_{CM}(n) \Rightarrow Safe(n)$
2. $Replace_{CM}(n) \Rightarrow Correct_{CM}(n)$

We denote the set of all admissible code motion transformations by $\mathcal{CM}_{Adm}$.

3.3 Computationally Optimal Code Motion

A code motion transformation $CM \in \mathcal{CM}_{Adm}$ is *computationally better*[8] than a code motion transformation $CM' \in \mathcal{CM}_{Adm}$ if and only if

$$\forall\, p \in \mathbf{PP}_{G^*}[\mathbf{s}^*, \mathbf{e}^*].\ Comp\#(p_{CM}) \leq Comp\#(p_{CM'})$$

where $Comp\#(p_{CM})$ denotes the number of computations of t on path p after applying the code motion transformation CM, i.e.:

$$Comp\#(p_{CM}) =_{df} \mid \{i \mid Insert_{CM}(p_i)\} \mid + \mid \{i \mid Comp(p_i) \wedge \neg Replace_{CM}(p_i)\} \mid$$

Analogously to the sequential case, we can now define:

Definition 3.3 (Computationally Optimal Code Motion)
An admissible code motion transformation $CM \in \mathcal{CM}_{Adm}$ is computationally optimal *if and only if it is computationally better than any other admissible code motion transformation. We denote the set of all computationally optimal code motion transformations by $\mathcal{CM}_{CmpOpt}$.*

Next we are going to specify the BCM_{PP}-transformation, a computationally optimal code motion transformation for parallel programs. This transformation evolves directly from its sequential counterpart, the busy code motion transformation of [KRS1, KRS2]. Central is the notion of 'earliest' safe program points, which are required for the as-early-as-possible placing strategy realized by this transformation.

Definition 3.4 (Earliestness)
A node $n \in N^*$ is earliest, *if it satisfies the predicate Earliest defined by*

$$Earliest(n) =_{df} Down\text{-}Safe(n) \wedge \neg Up\text{-}Safe(n) \wedge \begin{cases} tt & \text{if } n = \mathbf{s}^* \\ \bigvee_{m \in pred_{G^*}(n)} \neg(Transp(m) \wedge Safe(m)) & \text{otherwise} \end{cases}$$

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

Intuitively, n is earliest (for t), if it is

- down-safe, i.e., if the value of t is computed on every continuation of a program execution leaving n and reaching the end node,
- not up-safe, i.e., if the value of t is not already available at n, and if it is
- either the start node $\mathbf{s}^*$, or if the placement of t in some of n's predecessors would not be safe (it would introduce a new value on some path) or would not be transparent due to a modification of one of t's operands (a computation there would not yield the same value as in n).

The BCM_{PP}-Transformation

Table 1 now shows the specification of the BCM_{PP}-transformation:

• $\forall\, n \in N^*.\ \mathit{Insert}_{BCM_{PP}}(n) =_{df} \mathit{Earliest}(n)$
• $\forall\, n \in N^*.\ \mathit{Replace}_{BCM_{PP}}(n) =_{df} \mathit{Comp}(n) \wedge \mathit{Safe}(n)$

Table 1: The BCM_{PP}-Transformation

Intuitively, the BCM_{PP}-transformation moves computations as far as possible in the opposite direction of the control flow while maintaining admissibility. Thus, like its sequential counterpart, also the BCM_{PP}-transformation realizes the *as-early-as-possible* strategy for placing the computations in a program.

Following the lines of [KRS1, KRS2] we can prove the main result of this section:

Theorem 3.5 (BCM_{PP}-Optimality Theorem)
The BCM_{PP}-transformation is computationally optimal, i.e., $BCM_{PP} \in \mathcal{CM}_{CmpOpt}$.

3.4 The Impact of Synchronization and Interference

In the sequential setting earliestness and the replacement condition for a node n are equivalently defined by (cf. [KRS2])

$$\mathit{Safe}(n) \ \wedge \ \begin{cases} tt & \text{if } n = \mathbf{s}^* \\ \bigvee_{m \in pred_{G^*}(n)} \neg(\mathit{Transp}(m) \wedge \mathit{Safe}(m)) & \text{otherwise} \end{cases} \tag{1}$$

and

$$\mathit{Comp}(n) \tag{2}$$

In the parallel setting, however, these equivalences do not hold. The synchronization on leaving parallel statements necessitates the refinement of the earliestness definition, and the interference between parallel components requires the strengthening of the replacement condition.

Earliestness: Condition (1) reflects two constraints that must intuitively be satisfied by an earliest admissible placement:

- A temporary should at most be initialized at program sites where (i) it does not introduce the computation of a new value, and where (ii) an 'earlier' placement is hindered by some predecessor.

In the sequential setting the formulation of the safety requirement and the disjunction over properties of the predecessors can be strengthened to down-safety and a conjunction over those properties without affecting the meaning of the overall property:

$$\mathit{Down\text{-}Safe}(n) \;\wedge \bigwedge_{m \in pred_{G^*}(n)} \neg(\mathit{Transp}\,(m) \wedge \mathit{Safe}(m)) \tag{3}$$

While the second strengthening is essentially a consequence of some edge splitting, which is typical for code motion transformations (cf. Appendix B), the first strengthening reflects an additional constraint that must intuitively be satisfied by a computationally optimal placement:

- A temporary should at most be initialized at program sites where (iii) the value under consideration is not available, but (iv) required on every program continuation.

Remembering that up-safety means availability of the considered value, earliest program points should never be up-safe. In fact, this requirement is an absolute must for a computationally optimal placement, and it is naturally implied in the sequential case.

Unfortunately, in the parallel flow graph setting these strengthenings have a semantical impact, because of the synchronization that takes place at the nodes of N_X^*. This can be illustrated by means of node **33** and node **17** in Figure 14.

Node **33** is safe (even up-safe), and has a predecessor that is not safe (node **32**). Thus it satisfies the weakest formulation of earliest. However, node **33** is not down-safe and it has a predecessor, which is safe and transparent (node **30**). Thus it does not satisfy any of the strengthened versions. Node **17** illustrates that the condition on up-safety for earliest program points must explicitly be stated for the parallel setting. Though this node is down-safe and one of its predecessors (node **11**) violates safety, it is up-safe. The point here is that a single component of a parallel statement is already sufficient to establish up-safety (availability) at its synchronization node. Thus the value of $a + b$ is available at node **17** and therefore, it cannot be the initialization site of a computationally optimal placement.

Replacement: Also the definition of the replacement predicate is more intricate in the parallel setting, where it does not suffice to simply use $\mathit{Comp}\,(n)$ (cf. condition (2)): In the sequential setting the predicate *Comp* implies the predicate *Down-Safe* and thus the predicate *Safe*. As a consequence all paths reaching a node satisfying *Comp* are guaranteed to go through an 'earliest' program point with an initialization of the corresponding temporary that is not followed by a modification of one of the operands of the computation under consideration. Unfortunately, this does not work for nodes of parallel statements, because of interference. See nodes **8** and **10** of Figure 2 for illustration: Though every program execution reaching node **8** or node **10** goes through the initialization site of **h** at node **2**, there are some program executions where this initialization is followed by the modification of a in node **13** before the execution under consideration reaches the use sites at node **8** and **10**. In other words, for nodes n occurring in a parallel statement the implication $\mathit{Comp}\,(n) \Rightarrow \mathit{Down\text{-}Safe}(n)$ is invalid, whenever an interleaving predecessor of n modifies the computation under consideration.

3.5 Computing Up-Safety and Down-Safety

In order to complete the presentation of the BCM_{PP}-transformation, it is sufficient to specify the local semantic functionals $[\![\ ]\!]_{us}$ and $[\![\ ]\!]_{ds}$ for up-safety and down-safety. The DFA-algorithms they induce compute the set of up-safe and down-safe program points of a given argument program, respectively. The functionality of $[\![\ ]\!]_{us}$ and $[\![\ ]\!]_{ds}$ is given by

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

and they are defined as follows:[9]

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

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

In the literature these definitions are usually given in the following equivalent form:

$$\forall\, n \in N^*\ \forall\, b \in \mathcal{B}.\ [\![\, n \,]\!]_{us}(b) = (b \vee Comp(n)) \wedge Transp(n)$$

and

$$\forall\, n \in N^*\ \forall\, b \in \mathcal{B}.\ [\![\, n \,]\!]_{ds}(b) = Comp(n) \vee (Transp(n) \wedge b)$$

Remember that our implicit decomposition associates recursive assignments in parallel statements with two local semantic functions, as illustrated in Figures 8 and 11.

3.6 Discussing the Motivating Example

In this section we discuss the BCM_{PP}-transformation by means of the motivating example of Figure 1. First, we illustrate the hierarchical preprocesses for computing the sets of up-safe and down-safe program points. Subsequently, we illustrate the induced sets of earliest program points and of safe program points containing an original computation, which fix the insertion and the replacement points of the BCM_{PP}-transformation, respectively.

The Hierarchical Preprocesses for Up-Safety and Down-Safety

Figures 8, 9, and 10 illustrate the hierarchical preprocess computing the semantics of the subgraphs $G \in \mathcal{G}_{\mathcal{P}}(G^*)$ for the predicate up-safe. Figure 8 shows the flow graph of Figure 1 enhanced with the local semantic functions for up-safety with respect to the computation $a+b$. The hierarchical preprocess computes in the first step the semantics of all `par` statements of rank 0 of G^* as illustrated in Figure 9. These results are then used for computing the semantics of the single `par` statement of rank 1 of G^*, which is shown in Figure 10.

Subsequently, Figures 11, 12, and 13 illustrate the hierarchical preprocess of the down-safety analysis.

[9]Note that up-safety requires a forward analysis of the argument program, whereas down-safety requires a backward analysis.

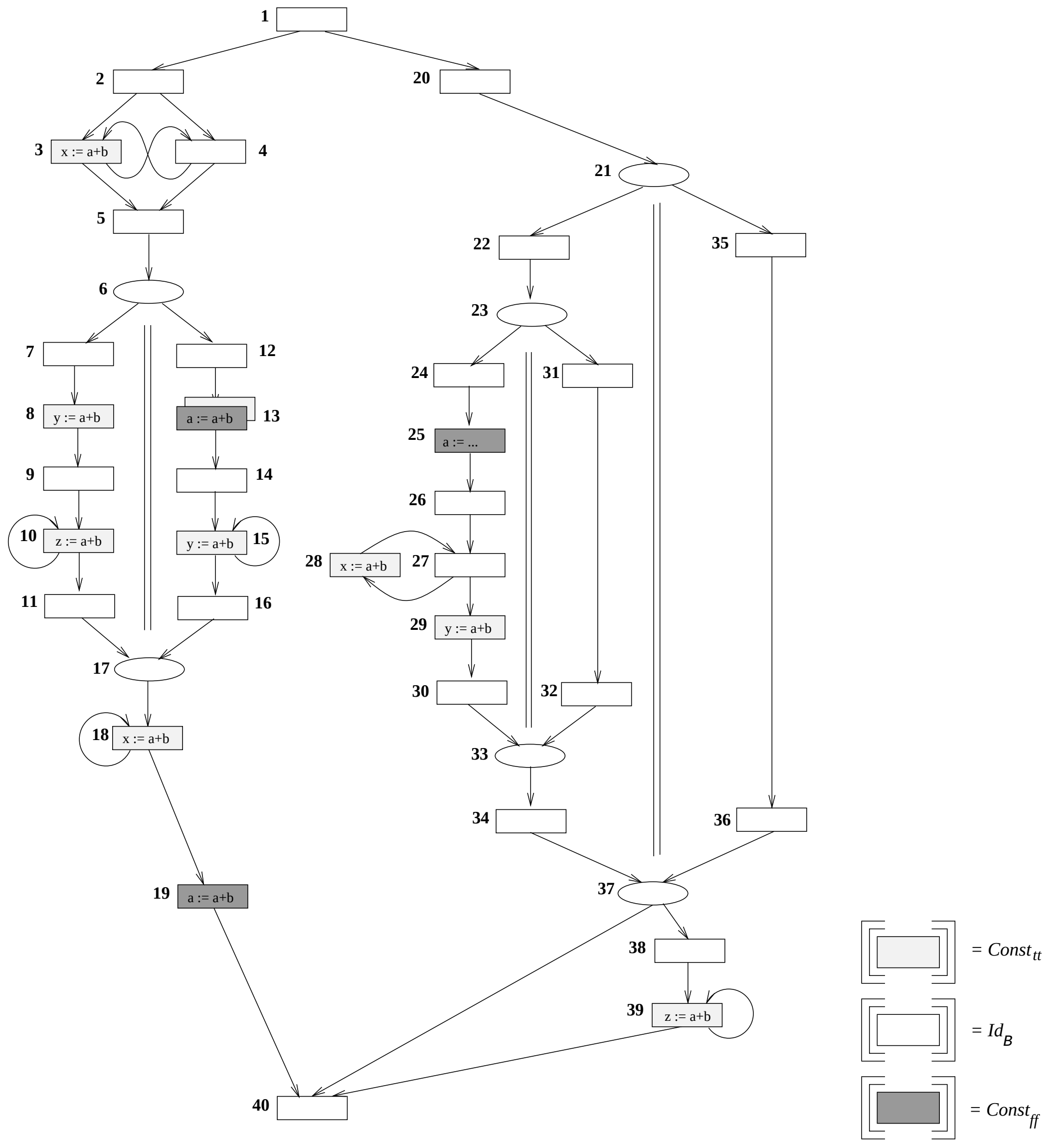

Figure 8: G^* with the Local Semantic Functional for Up-Safety $[\![\ ]\!]_{us}$ wrt $a + b$

The Effect of the BCM_{PP}-Transformation

After the hierarchical preprocesses for computing the semantics of all subgraphs $G \in \mathcal{G}_{\mathcal{P}}(G^*)$ with respect to up-safety and down-safety, the $PMFP_{BV}$-solution for both properties can be computed essentially as in the sequential case. It is worth noting here that both the preprocesses and the subsequent analyses for up-safety and down-safety are independent of each other. Thus, they can be computed in parallel in order to further speed up the complete analysis process. Figure 14 displays the result of computing the set of up-safe and down-safe program points based on the results of the preprocesses. Moreover, it shows the set of earliest program points, where a temporary **h** must be initialized with the value of $a + b$, and the set of program points, where an original computation of $a + b$ must be replaced. This results in the promised program of Figure 2, for which it is easy to check that it is indeed computationally optimal.

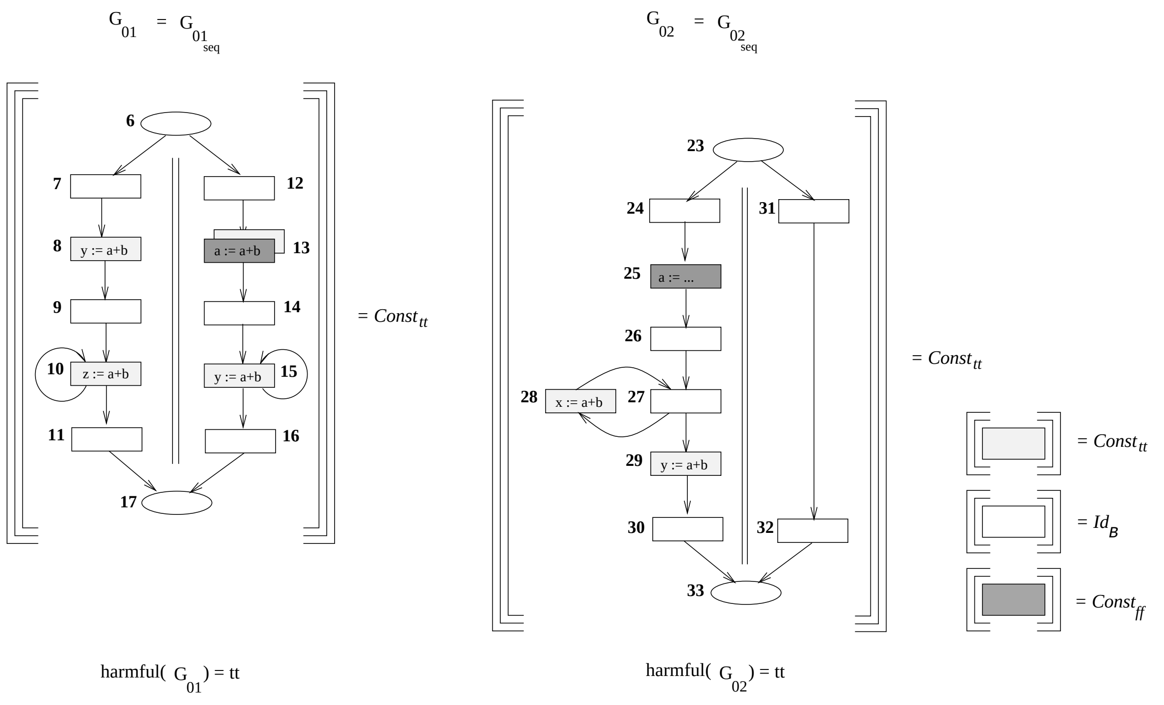

Figure 9: Up-Safety: After the 1st Iteration of the Outermost For-Loop of GLOBEFF

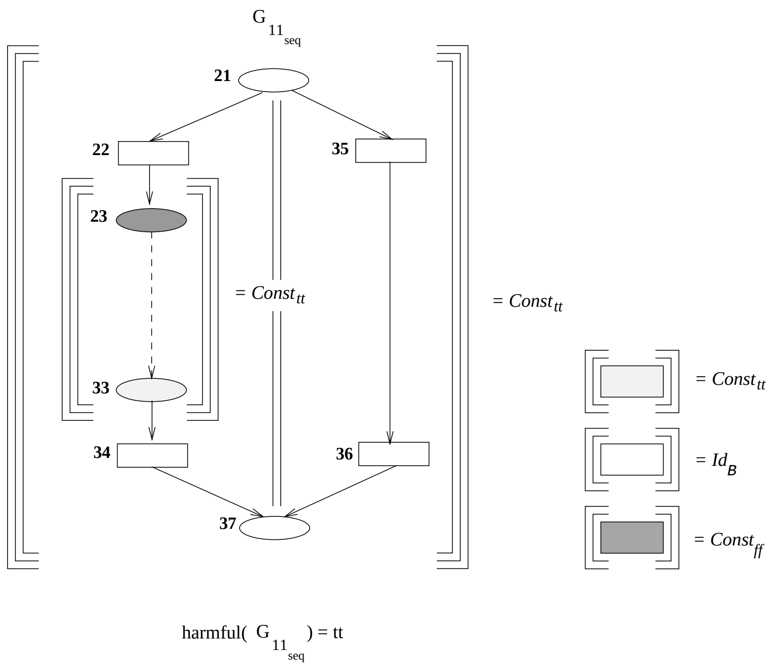

Figure 10: Up-Safety: After the 2nd Iteration of the Outermost For-Loop of GLOBEFF

4 Conclusions

In [KSV1] we have recently shown how to perform unidirectional bitvector analyses for parallel programs as efficiently as for sequential ones. Moreover, the analyses can easily be adapted from their sequential counterparts. This is highly relevant in practice because there is a broad variety of powerful classical program optimizations like code motion

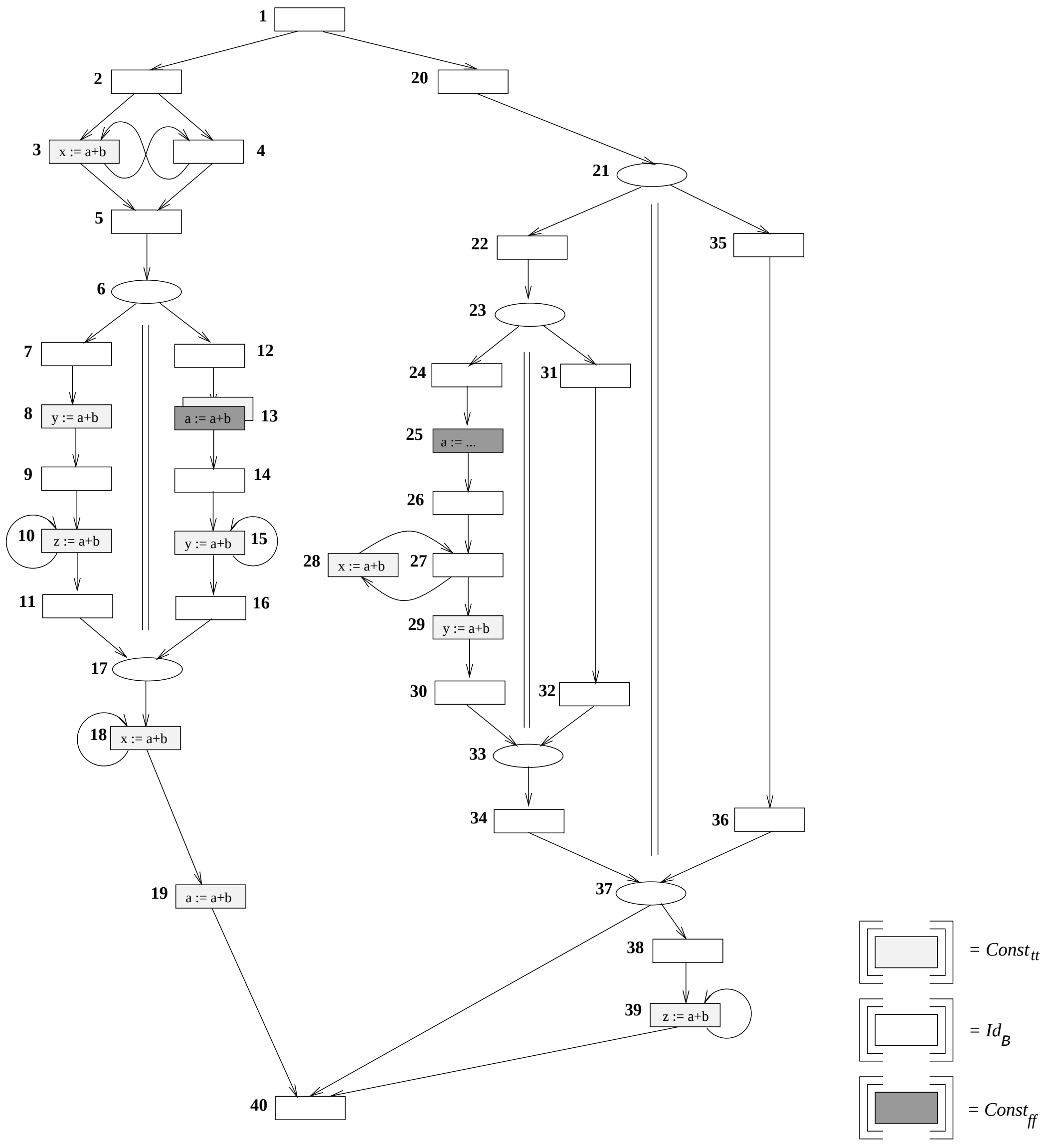

Figure 11: G^* with the Local Semantic Functional for Down-Safety $[\![\ ]\!]_{ds}$ wrt $a + b$

[DS2, DRZ, KRS1, KRS2], strength reduction [KRS3], partial dead code elimination [KRS4], and assignment motion [KRS5], which only require bitvector analyses of this type. All these techniques can now be adapted for parallel programs at almost no cost on the runtime and the implementation side. In this paper we demonstrated this by extending the *busy code motion* transformation of [KRS1, KRS2] to the parallel setting, which led to a computationally optimal code motion algorithm for parallel programs. The algorithm is implemented on the *Fixpoint Analysis Machine* of [SCKKM]. Moreover, the 'lazy' variant (cf. [KRS1, KRS2]) of this algorithm is implemented in the ESPRIT project COMPARE #5933 [Vo1, Vo2].

References

[CC] Cousot, P., and Cousot, R. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In

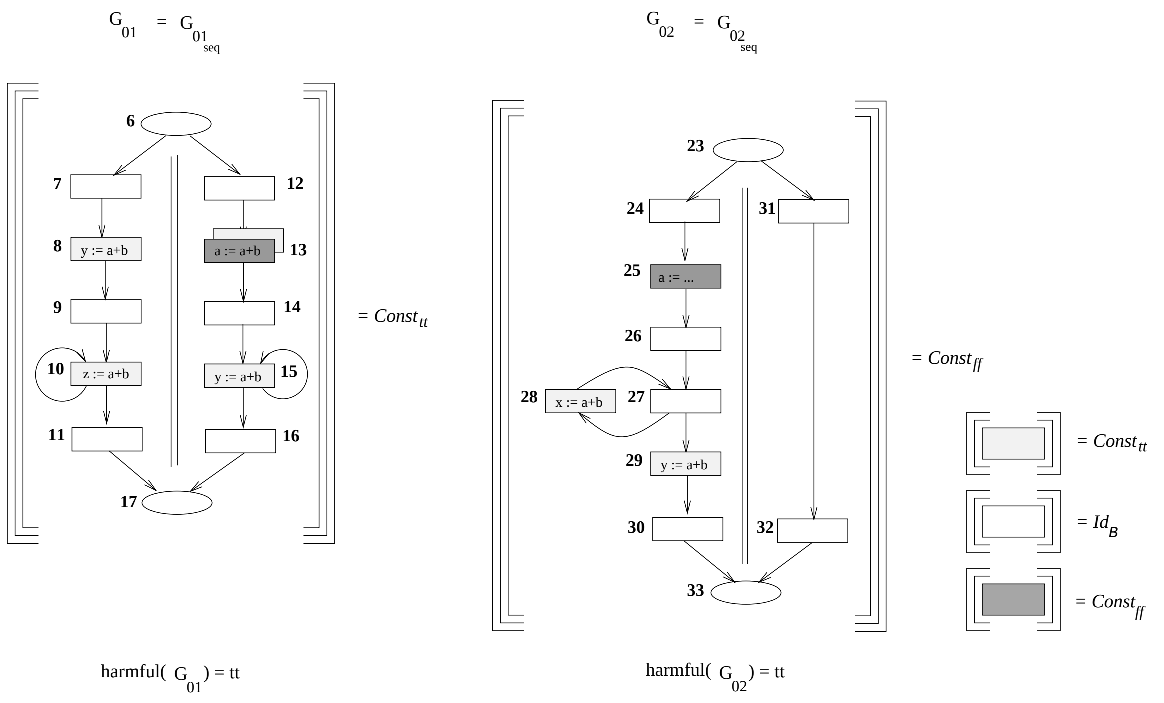

Figure 12: Down-Safety: After the 1st Iteration of the Outermost For-Loop of GLOBEFF

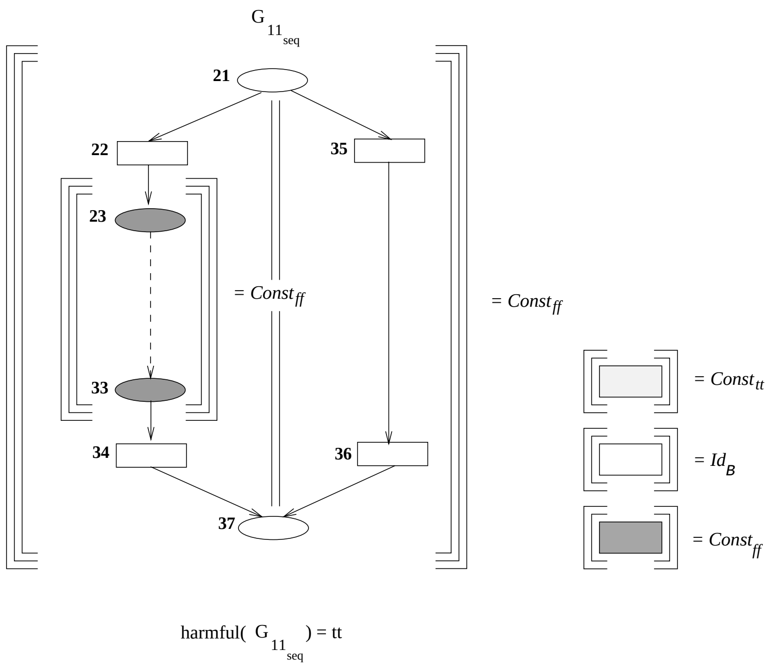

Figure 13: Down-Safety: After the 2nd Iteration of the Outermost For-Loop of GLOBEFF

Conference Record of the 4th International Symposium on Principles of Programming Languages (POPL'77), Los Angeles, California, 1977, 238 - 252.

[CH1] Chow, J.-H., and Harrison, W. L. Compile time analysis of parallel programs that share memory. In *Conference Record of the 19th International Symposium on Principles of Programming Languages (POPL'92)*, Albuquerque, New Mexico, 1992, 130 - 141.

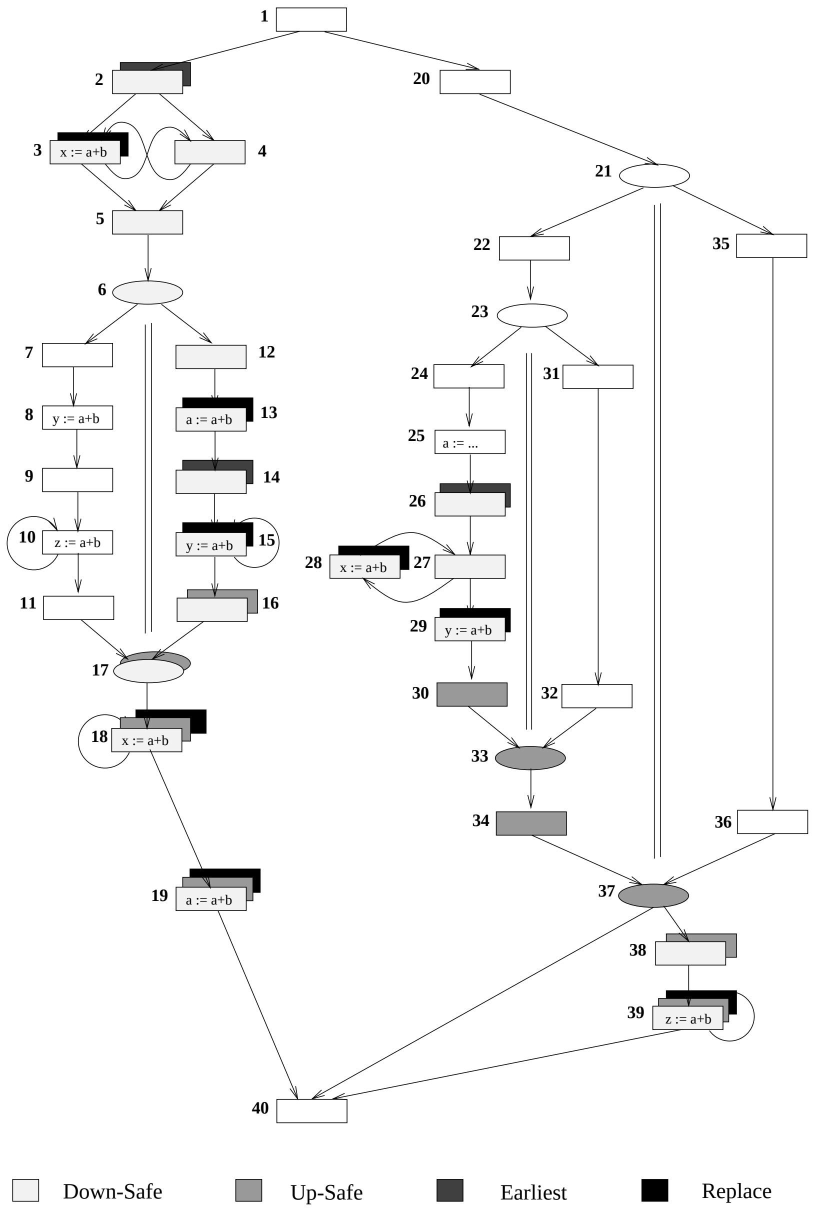

Figure 14: Down-Safe, Up-Safe, Earliest, and Replacement Program Points of $a + b$

[CH2] Chow, J.-H., and Harrison, W. L. State Space Reduction in Abstract Interpretation of Parallel Programs. In *Proceedings of the International Conference on Computer Languages, (ICCL'94)*, Toulouse, France, May 16-19, 1994, 277-288.

[Dh1] Dhamdhere, D. M. A fast algorithm for code movement optimization. *SIGPLAN Notices 23*, 10 (1988), 172 - 180.

[Dh2] Dhamdhere, D. M. A new algorithm for composite hoisting and strength reduction optimisation (+ Corrigendum). *Internat. J. Computer Math. 27*, (1989), 1 - 14 (+ 31 - 32).

[Dh3] Dhamdhere, D. M. Practical adaptation of the global optimization algorithm of Morel and Renvoise. *ACM Trans. Program. Lang. Syst. 13*, 2 (1991), 291 - 294.

[DBDS] Duri, S., Buy, U., Devarapalli, R., and Shatz, S. M. Using state space methods for deadlock analysis in Ada tasking. In *Proceedings of the ACM SIGSOFT'93 International Symposium on Software Testing and Analysis*, *Software Engineering Notes 18*, 3 (1993), 51 - 60.

[DP] Dhamdhere, D. M., and Patil, H. An elimination algorithm for bidirectional data flow problems using edge placement. *ACM Trans. Program. Lang. Syst. 15*, 2 (1993), 312 - 336.

[DRZ] Dhamdhere, D. M., Rosen, B. K., and Zadeck, F. K. How to analyze large programs efficiently and informatively. In *Proceedings of the ACM SIGPLAN'92 Conference on Programming Language Design and Implementation (PLDI'92)*, San Francisco, California, *SIGPLAN Notices 27*, 7 (1992), 212 - 223.

[DS1] Drechsler, K.-H., and Stadel, M. P. A solution to a problem with Morel and Renvoise's "Global optimization by suppression of partial redundancies". *ACM Trans. Program. Lang. Syst. 10*, 4 (1988), 635 - 640.

[DS2] Drechsler, K.-H., and Stadel, M. P. A variation of Knoop, Rüthing and Steffen's LAZY CODE MOTION. *SIGPLAN Notices 28*, 5 (1993), 29 - 38.

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

[GW] Godefroid, P., and Wolper, P. Using partial orders for the efficient verification of deadlock freedom and safety properties. In *Proceedings of the 3^{rd} International Workshop on Computer Aided Verification (CAV'91)*, Aalborg, Denmark, Springer-Verlag, LNCS 575 (1991), 332 - 342.

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

[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, *SIGPLAN Notices 27*, 7 (1992), 224 - 234.

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

[KRS3] Knoop, J., Rüthing, O., and Steffen, B. Lazy strength reduction. *Journal of Programming Languages 1*, 1 (1993), 71 - 91.

[KRS4] 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, *SIGPLAN Notices 29*, 6 (1994), 147 - 158.

[KRS5] Knoop, J., Rüthing, O., and Steffen, B. The power of assignment motion. In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation (PLDI'95)*, La Jolla, California, *SIGPLAN Notices 30*, 6 (1995), 233 - 245.

[KS] Knoop, J., and Steffen, B. The interprocedural coincidence theorem. In *Proceedings of the 4th International Conference on Compiler Construction (CC'92)*, Paderborn, Germany, Springer-Verlag, 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 *Preliminary Proceedings of the International Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'95)*, Aarhus, Denmark, BRICS Notes Series NS-95-2 (1995), 319 - 333.

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

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

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

[McD] McDowell, C. E. A practical algorithm for static analysis of parallel programs. *Journal of Parallel and Distributed Computing 6*, 3 (1989), 513 - 536.

[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.

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

[SCKKM] Steffen, B., Claßen, A., Klein, M., Knoop, J., and Margaria, T. The fixpoint-analysis machine. In *Proceedings of the 6th International Conference on Concurrency Theory (CONCUR'95)*, Philadelphia, Pennsylvania, Springer-Verlag, LNCS 962 (1995), 72 - 87.

[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.

[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*, Santa Clara, California, Springer-Verlag, LNCS 589 (1991), 405 - 419.

[Va] Valmari, A. A stubborn attack on state explosion. In *Proceedings of the 2nd International Conference on Computer Aided Verification*, New Brunswick, New Jersey, Springer-Verlag, LNCS 531 (1990), 156 - 165.

[Vo1] Vollmer, J. Data flow equations for parallel programs that share memory. Tech. Rep. 2.11.1 of the ESPRIT Project COMPARE #5933, Fakultät für Informatik, Universität Karlsruhe, Germany, (1994).

[Vo2] Vollmer, J. Data flow analysis of parallel programs. In *Proceedings of the IFIP WG 10.3 Working Conference on Parallel Architectures and Compilation Techniques (PACT'95)*, Limassol, Cyprus, 1995, 168 - 177.

[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*, Salzburg, Austria, Springer-Verlag, LNCS 591 (1991), 139 - 156.

A Computing the $PMFP_{BV}$-Solution

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

Input: *A parallel flow graph* $G^* = (N^*, E^*, \mathbf{s}^*, \mathbf{e}^*)$*, a local semantic functional* $[\![\ ]\!] : N^* \to \mathcal{F}_\mathcal{B}$*, 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 step 2 of the three-step procedure of Section 2.3, and with functions* $[\![\, n \,]\!] \in \mathcal{F}_\mathcal{B}$*,* $n \in N^*$*, representing the greatest solution of the equation system of Definition 2.7. In fact, after the termination of the algorithm the functional* $[\![\ ]\!]$ *satisfies:*

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

Remark: *The global variables* $[\![\, G \,]\!]^*$*,* $G \in \bigcup\{\mathcal{G}_\mathcal{C}(G') \,|\, G' \in \mathcal{G}_\mathcal{P}(G^*)\,\}$*, each of which is storing a function of* $\mathcal{F}_\mathcal{B}$*, are used for storing the global effects of component graphs of graphs* $G' \in \mathcal{G}_\mathcal{P}(G^*)$ *during the hierarchical computation of the* $PMFP_{BV}$*-solution. The global variables* $harmful(G_{seq})$*,* $G \in \bigcup\{\,\mathcal{G}_\mathcal{C}(G') \,|\, G' \in \mathcal{G}_\mathcal{P}(G^*)\,\}$*, store whether* G *contains a node* n *with* $[\![\, n \,]\!] = Const_{f\!f}$*. These variables are used to compute the value of the predicate* $NonDestructed$ *of Section 2.3.*

```
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;
            $[\![ G' ]\!]^* := \begin{cases} Const_{ff} & \text{if } \exists\, G'' \in \mathcal{G}_{\mathcal{C}}(G').\ [\![ end(G''_{seq}) ]\!] = Const_{ff} \\ Id_{\mathcal{B}} & \text{if } \forall\, G'' \in \mathcal{G}_{\mathcal{C}}(G').\ [\![ end(G''_{seq}) ]\!] = Id_{\mathcal{B}} \\ Const_{tt} & \text{otherwise} \end{cases}$
        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 $G' \in \underline{\mathcal{G}_\mathcal{C}}(pfg(n))$ DO
                    $PMFP_{BV}(G', \overline{[\![ \; ]\!]}, [\![ n ]\!], \sum_{G'' \in \mathcal{G}_\mathcal{C}(pfg(n)) \backslash \{G'\}} harmful(G''))$ OD;
                  $f := [\![ pfg(n) ]\!]^* \circ [\![ n ]\!]$;
                  IF $[\![ end(pfg(n)) ]\!] \sqsupset f$
                    THEN
                      $[\![ end(pfg(n)) ]\!] := f$;
                      $workset := workset \cup \{ end(pfg(n)) \}$ FI
              FI
            END
        OD
      FI
    END.

    PROCEDURE $MFP$ $(G = (N, E, \mathbf{s}, \mathbf{e}) : SequentialFlowGraph;$
                     $[\![ \; ]\!] : N \rightarrow \mathcal{F}_\mathcal{B}$ : $LocalSemanticFunctional$;
                     $f_{start} : \mathcal{F}_\mathcal{B})$;
    VAR $f : \mathcal{F}_\mathcal{B}$;
    BEGIN
      ( Initialization of the annotation array gtr and the variable workset )
      FORALL $n \in N \backslash \{\mathbf{s}\}$ DO $[\![ n ]\!] := Const_{tt}$ OD;
      $[\![ \mathbf{s} ]\!] := f_{start}$;
      $workset := \{ n \mid n = \mathbf{s} \vee [\![ n ]\!] = Const_{ff} \}$;

      ( Iterative fixed point computation )
      WHILE $workset \neq \emptyset$ DO
        LET $n \in workset$
          BEGIN
            $workset := workset \backslash \{ n \}$;
            $f := [\![ n ]\!] \circ [\![ n ]\!]$;
            FORALL $m \in succ_G(n)$ DO
              IF $[\![ m ]\!] \sqsupset f$ THEN $[\![ m ]\!] := f; workset := workset \cup \{ m \}$ FI OD
          END
      OD
    END.
```

Let $\llbracket\, n \,\rrbracket_{alg}$, $n \in N^*$, denote the final values of the corresponding variables after the termination of Algorithm A.1, and $\llbracket\, n \,\rrbracket$, $n \in N^*$, the greatest solution of the equation system of Definition 2.7, then we have:

Theorem A.2 $\forall\, n \in N^*.\ \llbracket\, n \,\rrbracket_{alg} = \llbracket\, n \,\rrbracket$

B Critical Edges

It is well-known that in order to exploit the full power of code motion, *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 it is illustrated by the simple example of Figure 15.

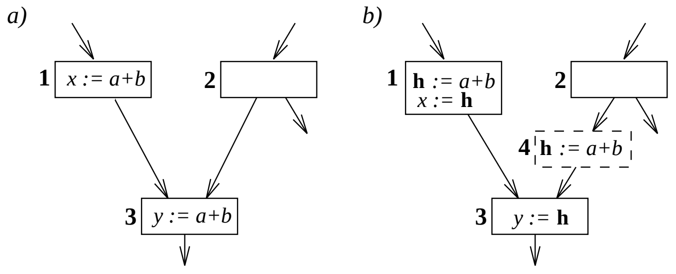

Figure 15: Critical Edges

Note that in Figure 15(a) the computation of "$a+b$" at node **3** is partially redundant with respect to the computation of "$a+b$" at node **1**. However, this partial redundancy cannot safely be eliminated by moving the computation of "$a+b$" to its preceding nodes, because this may introduce a new computation on a path leaving node **2** on the right branch. On the other hand, it can safely be eliminated after inserting a synthetic node **4** in the critical edge $(\mathbf{2},\mathbf{3})$ as illustrated in Figure 15(b).

In this paper we thus assume that all edges in G^* leading to a node with more than one predecessor, except for edges leading to a node of N^*_X, have been split by inserting a synthetic node. Obviously, this simple transformation guarantees that all critical edges are eliminated. Moreover, as in the sequential setting it simplifies the process of code motion as computationally optimal programs can be obtained by moving all computations to node entries (cf. [KRS1]).