Parallelism for Free: Efficient and Optimal Bitvector Analyses for Parallel Programs

JENS KNOOP and BERNHARD STEFFEN
Universität Passau
and
JÜRGEN VOLLMER
Universität Karlsruhe

We consider parallel programs with shared memory and interleaving semantics, for which we show how to construct for unidirectional bitvector problems optimal analysis algorithms that are as efficient as their purely sequential counterparts and that can easily be implemented. Whereas the complexity result is rather obvious, our optimality result is a consequence of a new Kam/Ullman-style Coincidence Theorem. Thus using our method, the standard algorithms for sequential programs computing liveness, availability, very busyness, reaching definitions, definition-use chains, or the analyses for performing code motion, assignment motion, partial dead-code elimination or strength reduction, can straightforward be transferred to the parallel setting at almost no cost.

Categories and Subject Descriptors: D.3.2 [**Programming Languages**]: Language Classifications—*concurrent, distributed, and parallel languages*; D.3.4 [**Programming Techniques**]: Processors—*code generation; compilers; optimization*

General Terms: Algorithms, Languages, Performance

Additional Key Words and Phrases: Assignment motion, bitvector problems, code motion, data flow analysis, definition-use chains, interleaving semantics, parallelism, partial dead-code elimination, program optimization, shared memory, strength reduction, synchronization

1. MOTIVATION

Parallel implementations are of growing interest, as they are more and more supported by modern hardware environments. However, despite its importance [Srinivasan and Wolfe 1991; Srinivasan et al. 1993; Wolfe and Srinivasan 1991], there is currently very little work on classical data flow analysis for parallel languages.

The third author has been supported by the ESPRIT Project COMPARE number 5933. A preliminary version of this article was presented at TACAS'95.
Authors' addresses: J. Knoop and B. Steffen, Fakultät für Mathematik und Informatik, Universität Passau, Innstrasse 33, D-94032 Passau, Germany; email: {knoop; steffen}@fmi.uni-passau.de; J. Vollmer, Fakultät für Informatik, Institut für Programmstrukturen und Datenorganisation (IPD), Universität Karlsruhe, Vincenz-Prießnitz-Straße 3, D-76128 Karlsruhe, Germany; email: vollmer@ipd.info.uni-karlsruhe.de.

Probably, the reason for this deficiency is that a naive adaptation fails [Midkiff and Padua 1990] and that the straightforward correct adaptation needs an unacceptable effort, which is caused by the interleavings that manifest the possible executions of a parallel program.

Thus, either heuristics are proposed to avoid the consideration of all the interleavings [McDowell 1989], or restricted situations are considered, which do not require to consider the interleavings at all. Grunwald and Srinivasan [1993a] for example require data independence of parallel components according to the PCF Fortran standard. Thus the result of a parallel execution does not depend on the particular choice of the interleaving, which is exploited for the construction of an optimal and efficient algorithm determining the reaching-definition information. However, it is not investigated how to systematically adapt their approach to other problems. A different setup is considered by Long and Clarke [1991], who propose a data flow analysis framework for Ada programs with task parallelism and rendezvous synchronization, but without shared variables. Though their Ada-specific setup is incomparable to ours, in particular as shared variables are excluded, both approaches have in common to borrow ideas from interprocedural data flow analysis. Dwyer and Clarke [1994] consider a setup similar to the one of Long and Clarke with explicit tasking and rendezvous communication in the sense of Ada tasking programs. They put emphasis on the trade-off between accuracy and efficiency with respect to explicitly stated correctness properties like mutual exclusion or data races. Completely different to heuristic approaches is the approach of abstract interpretation-based state space reduction proposed by Duri et al. [1993], Chow and Harrison [1992; 1994], Godefroid and Wolper [1991], and Valmari [1990], which 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 this article we consider parallel programs with explicit parallelism, shared memory, and interleaving semantics without special synchronization statements, for which we show how to construct for unidirectional bitvector problems analysis algorithms that

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

The first property is a consequence of a Kam/Ullman-style [Kam and Ullman 1977] Coincidence Theorem for bitvector analyses stating that the *parallel meet over all paths* (*PMOP*) solution, which specifies the desired properties, coincides with our *parallel bitvector maximal-fixed-point* ($PMFP_{BV}$) solution, which is the basis of our algorithm. This result is rather surprising, as it states that, though the various interleavings of the executions of parallel components are semantically different, they need not be considered during bitvector analysis. This is the key observation of this article.

The second property is a simple consequence of the fact that our algorithms behave like standard bitvector algorithms. In particular, they do *not* require the consideration of any kind of global state space. This is important, as even the corresponding reduced state spaces would usually still be exponential in size.

The third property is due to the fact that only a minor modification of the sequential bitvector algorithm needs to be applied after a preprocess consisting of a single fixed-point routine (cf. Section 3.4).

Thus, using our methods all the well-known algorithms for unidirectional bitvector problems can be adapted for parallel programs at almost no cost on the runtime and the implementation side. This is of high importance in practice, as this class of bitvector problems has a broad scope of applications ranging from simple analyses like liveness, availability, very busyness, reaching definitions, and definition-use chains [Hecht 1977] to sophisticated and powerful program optimizations like code motion [Dhamdhere et al. 1992; Drechsler and Stadel 1993; Knoop et al. 1992; 1994a], partial dead-code elimination [Knoop et al. 1994b], assignment motion [Knoop et al. 1995], and strength reduction [Knoop et al. 1993]. All these techniques can now be made available for parallel programs. In Section 4 we demonstrate this by two examples. We present the parallel extensions of the *busy-code-motion* transformation of Knoop et al. [1994a], and the partial dead-code elimination algorithm of Knoop et al. [1994b]. We conjecture that, like their sequential counterparts, these algorithms are unique in that they optimally eliminate the partially redundant computations and partially dead assignments in a parallel argument program, respectively.

We remark that the results of this article do not directly apply to algorithms which require bidirectional bitvector analyses like the pioneering code motion algorithm of Morel and Renvoise [1979] or the strength reduction algorithms of Dhamdhere [1989] and Joshi and Dhamdhere [1982a; 1982b]. This limitation has an analogy in interprocedural data flow analysis, where it turned out that unidirectionally based algorithms can systematically be extended to capture the effects of procedure calls precisely [Knoop and Steffen 1993; Knoop et al. 1994c], whereas extensions of bidirectionally based algorithms usually result in heuristics [Morel 1984; Morel and Renvoise 1981]. The best way to overcome the problems with bidirectional analyses is to decompose them into sequences of unidirectional analyses as it was done for example for the code motion algorithm of Morel and Renvoise [1979] and the strength reduction algorithms of Dhamdhere [1989] and Joshi and Dhamdhere [1982a; 1982b] by Knoop et al. [1992; 1994a] and Knoop et al. [1993], respectively. Actually, we do not know of any practically relevant bidirectional bitvector analysis that cannot be decomposed into unidirectional components.

Structure of the Article. After recalling the sequential situation in Section 2, we develop the corresponding notions for the parallel situation in Section 3. Subsequently, we present two applications of our algorithm in Section 4, and draw our conclusions in Section 5. The Appendix contains the detailed generic algorithm.

2. SEQUENTIAL PROGRAMS

In this section we summarize the sequential setting of data flow analysis.

2.1 Representation

In the sequential setting it is common to represent procedures as *directed flow graphs* $G = (N, E, \mathbf{s}, \mathbf{e})$ with node set N and edge set E. Nodes $n \in N$ represent the statements, edges $(n, m) \in E$ the *nondeterministic* branching structure of the

procedure under consideration, and $\mathbf{s}$ and $\mathbf{e}$ denote the unique *start node* and *end node* of G, which are assumed to possess no predecessors and successors, respectively, and to represent the empty statement `skip`. $pred_G(n) =_{df} \{ m \mid (m,n) \in E \}$ and $succ_G(n) =_{df} \{ m \mid (n,m) \in E \}$ denote the set of all immediate predecessors and successors of a node n, respectively. A *finite path* in G is a sequence $(n_1, \ldots, n_q)$ of nodes such that $(n_j, n_{j+1}) \in E$ for $j \in \{1, \ldots, q-1\}$. $\mathbf{P}_G[m,n]$ 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. Finally, every node $n \in N$ is assumed to lie on a path from $\mathbf{s}$ to $\mathbf{e}$.

2.2 Data Flow Analysis

Data flow analysis (*DFA*) is concerned with the static analysis of programs in order to support the generation of efficient object code by "optimizing" compilers [Hecht 1977; Muchnick and Jones 1981]. 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* [Cousot and Cousot 1977; Marriot 1993]. 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. Usually, the abstract semantics is specified by a *local semantic functional*

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

which gives abstract meaning to every program statement in terms of a transformation function on a complete lattice $(\mathcal{C}, \sqcap, \sqsubseteq, \bot, \top)$, whose elements express the DFA-information of interest. In the following $\mathcal{C}$ will always denote a complete lattice with least element $\bot$ and greatest element $\top$. As the start node $\mathbf{s}$ and the end node $\mathbf{e}$ are assumed to represent the empty statement `skip`, they are usually associated with the identity $Id_{\mathcal{C}}$ on $\mathcal{C}$.

A local semantic functional $[\![\ ]\!]$ can easily be extended to cover finite paths as well. For every path $p = (n_1, \ldots, n_q) \in \mathbf{P}_G[m,n]$, we define:

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

2.2.1 *The MOP-Solution of a DFA.* The solution of the *meet-over-all-paths* (*MOP*) approach in the sense of Kam and Ullman [1977], or for short, the *MOP*-solution, defines the intuitively desired solution of a DFA. This approach directly mimics possible program executions in that it "meets" (intersects) all informations belonging to a program path reaching the program point under consideration. This directly reflects our desires, but is in general not effective.

The *MOP*-Solution:

$$\forall\, n \in N\ \forall\, c_0 \in \mathcal{C}.\ MOP_{(G,[\![\ ]\!])}(n)(c_0) = \bigsqcap \{ [\![p]\!](c_0) \mid p \in \mathbf{P}_G[\mathbf{s}, n[\}.$$

2.2.2 *The MFP-Solution of a DFA.* The point of the *maximal-fixed-point* (*MFP*) approach in the sense of Kam and Ullman [1977] is to iteratively approximate the greatest solution of a system of equations which specifies consistency constraints between preconditions of the nodes of G expressed in terms of $\mathcal{C}$:

Equation System 2.2.2.1.

$$\mathbf{pre}(n) \;=\; \begin{cases} c_0 & \text{if } n = \mathbf{s} \\ \bigsqcap \{ [\![m]\!](\mathbf{pre}(m)) \mid m \in pred_G(n) \} & \text{otherwise.} \end{cases}$$

Denoting the greatest solution of Equation System 2.2.2.1 with respect to the start information $c_0 \in \mathcal{C}$ by $\mathbf{pre}_{c_0}$, the solution of the *MFP*-approach is defined by:

The *MFP*-Solution: $\forall\, n \in N\ \forall\, c_0 \in \mathcal{C}.\ MFP_{(G,[\![\ ]\!])}(n)(c_0) = \mathbf{pre}_{c_0}(n)$.

For monotonic functionals,[1] this leads to a suboptimal but algorithmic description (see Algorithm A.1 in Appendix A). The question of optimality of the *MFP*-solution was elegantly answered by Kildall [1972; 1973] and Kam and Ullman [1977]:

THEOREM 2.2.2.2. (THE (SEQUENTIAL) COINCIDENCE THEOREM). *Given a flow graph $G = (N, E, \mathbf{s}, \mathbf{e})$, the MFP-solution and the MOP-solution coincide, i.e., $\forall\, n \in N\ \forall\, c_0 \in \mathcal{C}.\ MOP_{(G,[\![\ ]\!])}(n)(c_0) = MFP_{(G,[\![\ ]\!])}(n)(c_0)$, whenever all the semantic functions $[\![n]\!]$, $n \in N$, are distributive.*[2]

2.2.3 *The Functional Characterization of the MFP-Solution.* From interprocedural DFA, it is well known that the *MFP*-solution can alternatively be defined by means of a functional approach [Sharir and Pnueli 1981]. Here, one iteratively approximates the greatest solution of a system of equations specifying consistency between functions $[\![n]\!]$, $n \in N$. Intuitively, a function $[\![n]\!]$ transforms a data flow information which is assumed to be valid at the start node of the program into the data flow information being valid before the execution of n.

Definition 2.2.3.1. (*The Functional Approach*). The functional $[\![\ ]\!] : N \to (\mathcal{C} \to \mathcal{C})$ is defined as the greatest solution of the equation system given by:

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

The following equivalence result is important [Knoop and Steffen 1992]:

THEOREM 2.2.3.2. $\forall\, n \in N\ \forall\, c_0 \in \mathcal{C}.\ MFP_{(G,[\![\ ]\!])}(n)(c_0) = [\![n]\!](c_0)$.

The functional characterization of the *MFP*-solution will be the (intuitive) key for computing the parallel version of the maximal-fixed-point solution. As we are only dealing with Boolean values later on, this characterization can easily be coded back into the standard form.

3. PARALLEL PROGRAMS

As usual, we consider a parallel imperative programming language with interleaving semantics. Formally, this means that we view parallel programs semantically

[1] A function $f : \mathcal{C} \to \mathcal{C}$ is called *monotonic* iff $\forall\, c, c' \in \mathcal{C}.\ c \sqsubseteq c'$ implies $f(c) \sqsubseteq f(c')$.

[2] A function $f : \mathcal{C} \to \mathcal{C}$ is called *distributive* iff $\forall\, C' \subseteq \mathcal{C}.\ f(\bigsqcap C') = \bigsqcap \{f(c) \mid c \in C'\}$. It is well known that distributivity is a stronger requirement than monotonicity in the following sense: A function $f : \mathcal{C} \to \mathcal{C}$ is monotonic iff $\forall\, C' \subseteq \mathcal{C}.\ f(\bigsqcap C') \sqsubseteq \bigsqcap \{f(c) \mid c \in C'\}$.

as "abbreviations" of usually much larger nondeterministic programs, which result from a product construction between parallel components [Chow and Harrison 1992; 1994; Cousot and Cousot 1984]. In fact, in the worst case the size of the nondeterministic "product" program grows exponentially in the number of parallel components of the corresponding parallel program. This immediately clarifies the dilemma of data flow analysis for parallel programs: even though it can be reduced to standard data flow analysis on the corresponding nondeterministic program, this approach is unacceptable in practice for complexity reasons. Fortunately, as we will see in Section 3.3, unidirectional bitvector analyses, which are most relevant in practice, can be performed as efficiently on parallel programs as on sequential programs.

The following section establishes the notational background for the formal development and the proofs.

3.1 Representation

Syntactically, parallelism is 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. Without special synchronization statements this already introduces the phenomena of interference and synchronization 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 Chow and Harrison [1994] and Vollmer [1994; 1995].

Similar to Srinivasan et al. [1993] and Grunwald and Srinivasan [1993a], 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 a **par** statement a parallel flow graph is a nondeterministic flow graph in the sense of Section 2, i.e., 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 possess no predecessors and successors, respectively. As in Section 2, we assume that every node $n \in N^*$ lies on a path from $\mathbf{s}^*$ to $\mathbf{e}^*$ and that the start node and the end node of a parallel flow graph represent the empty statement **skip**. Additionally, let $pred_{G^*}(n) =_{df} \{ m \mid (m, n) \in E^* \}$ 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 **par** statement and each of its components are also considered parallel flow graphs. The graph G representing a complete **par** statement arises from linking its component graphs by means of a **ParBegin** node and a **ParEnd** node having 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 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** nodes and **ParEnd** nodes by ellipses, and additionally we separate the component graphs of

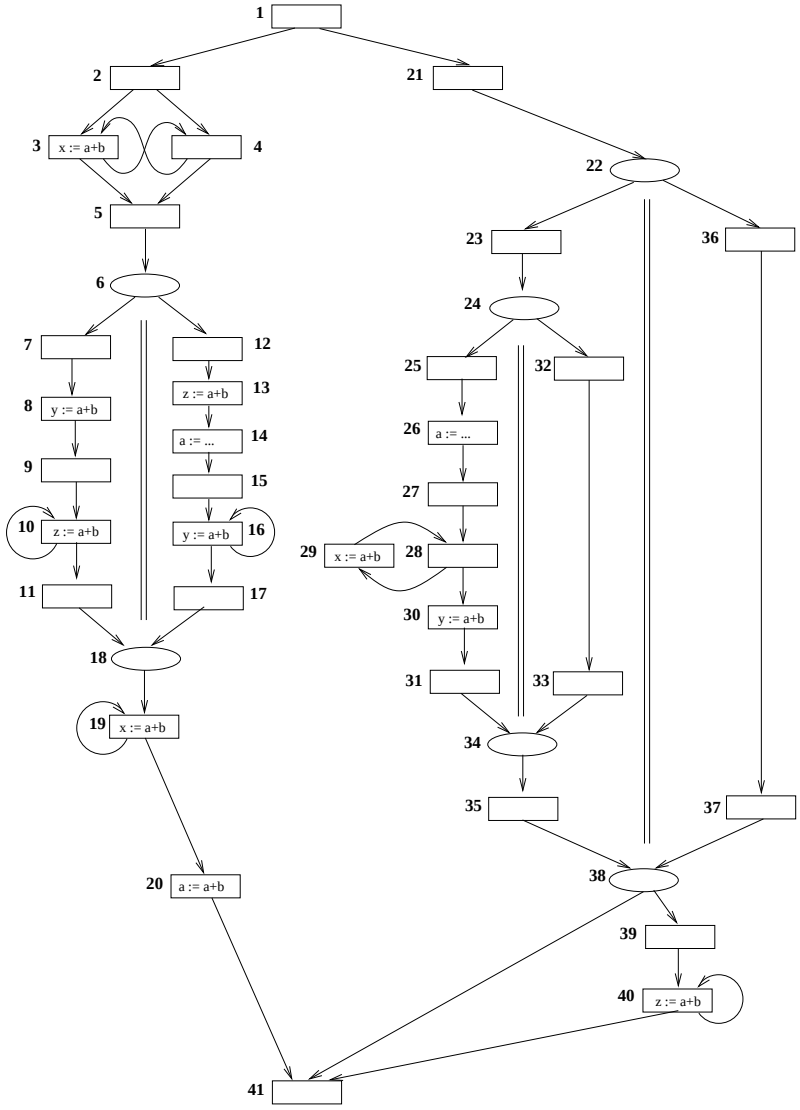

Fig. 1. The parallel flow graph G^*.

parallel statements by two parallels in the figures of this article.

For a parallel flow graph G^*, let $\mathcal{G}_{\mathcal{P}}(G^*)$ denote the set of all subgraphs representing a **par** statement. Additionally, let $\mathcal{G}_{\mathcal{C}}(G')$, $G' \in \mathcal{G}_{\mathcal{P}}(G^*)$, denote the set of component graphs of G', and let $\mathcal{G}_{\mathcal{C}}(G^*)$ be an abbreviation of $\bigcup \{ \mathcal{G}_{\mathcal{C}}(G') \mid G' \in \mathcal{G}_{\mathcal{P}}(G^*) \}$. Note that for $G \in \mathcal{G}_{\mathcal{P}}(G^*)$, G and all its component graphs $G' \in \mathcal{G}_{\mathcal{C}}(G)$ are single-entry/single-exit regions of G^*. Let

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

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 $\mathcal{G}_{\mathcal{P}}(G^*)$, respectively.[3] For every flow graph $G \in \mathcal{G}_{\mathcal{P}}(G^*)$ we introduce a *rank* that is recursively defined by:

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

This is illustrated in Figure 2 and Figure 3, which show the set of parallel subgraphs of rank 1 and of rank 0 of the parallel flow graph of Figure 1.

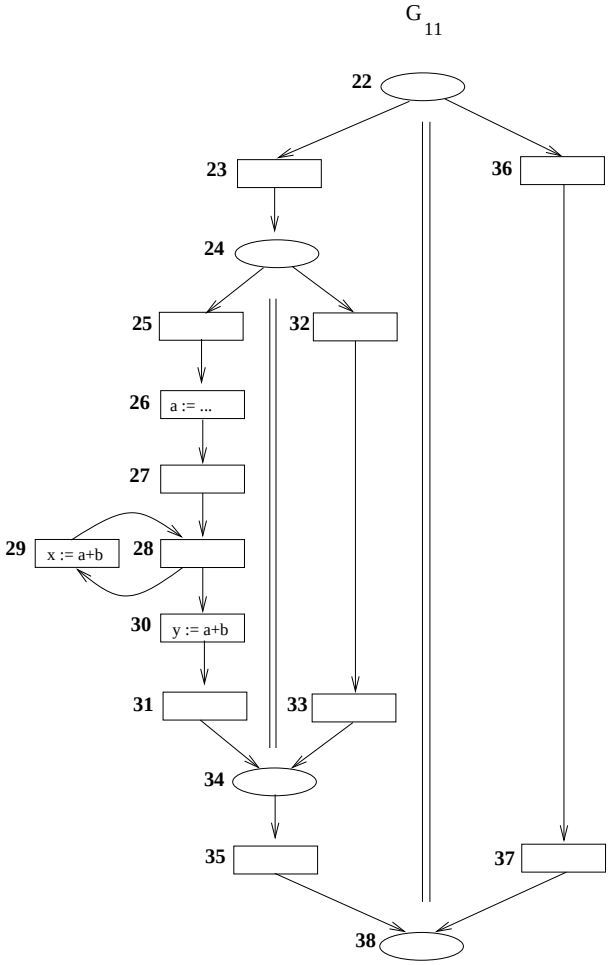

Fig. 2. $\{G \mid G \in \mathcal{G}_{\mathcal{P}}(G^*) \wedge rank(G) = 1\} = \{G_{11}\}$.

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

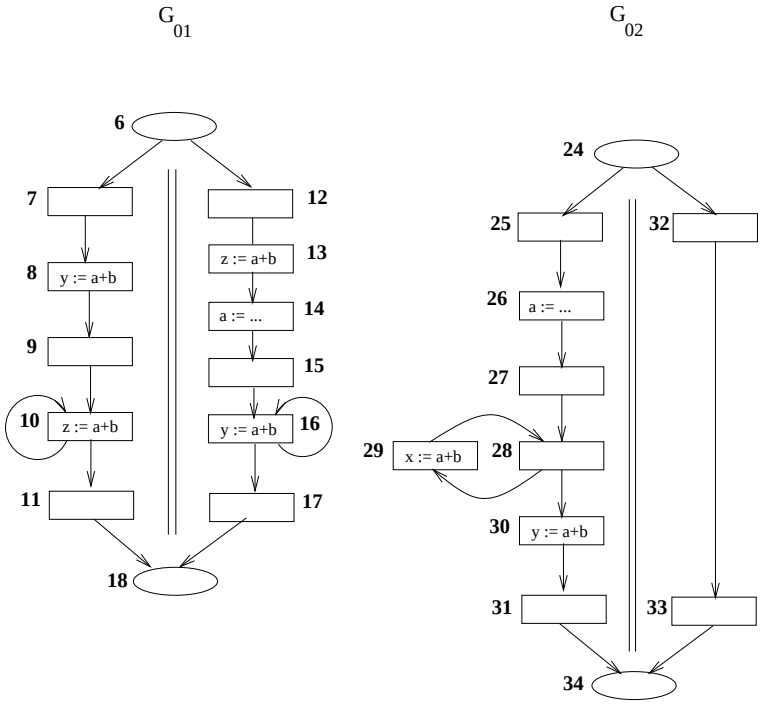

Fig. 3. $\{G \mid G \in \mathcal{G}_{\mathcal{P}}(G^*) \wedge rank(G) = 0\} = \{G_{01}, G_{02}\}$.

For every (parallel) flow graph we define the functions *Nodes*, *start*, and *end*, which map their argument to its node set, its start node, and end node, respectively. Let

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

denote the sets of start nodes (i.e., `ParBegin` nodes) and end nodes (i.e., `ParEnd` nodes) of the graphs of $\mathcal{G}_{\mathcal{P}}(G^*)$. Additionally, we introduce the function *pfg* which maps a node n occuring in a flow graph $G \in \mathcal{G}_{\mathcal{P}}(G^*)$ to the smallest flow graph of $\mathcal{G}_{\mathcal{P}}(G^*)$ containing it, and to G^* otherwise, i.e.,

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

We extend *pfg* to graphs $G \in \mathcal{G}_{\mathcal{C}}(G^*)$ by identifying $pfg(G)$ with $pfg(start(G))$. Moreover, we define a function *cfg* which maps a node n occurring in a flow graph $G \in \mathcal{G}_{\mathcal{C}}(G^*)$ to the smallest flow graph of $\mathcal{G}_{\mathcal{C}}(G^*)$ containing it, and to G^* otherwise, i.e.,

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

Similar to *pfg*, we extend *cfg* to graphs $G \in \mathcal{G}_{\mathcal{P}}(G^*)$ by identifying $cfg(G)$ with $cfg(start(G))$. Both for *pfg* and *cfg* the overloading of notation is harmless. In fact, they are well defined because `par` statements in a program are either unrelated

or properly nested.

Finally, 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 graphs $G' \in \mathcal{G}_{\mathcal{P}}^{max}(G)$ 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 in the sense of Section 2. This is illustrated in Figure 4 and in Figure 5, which show the sequentialized versions of the parallel flow graphs of Figure 1 and Figure 2, respectively.

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 of its *static predecessors* $pred_G(n)$. 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. In the program of Figure 1 for example, node **27** can dynamically be preceded not only by its unique static predecessor **26** but also by the nodes **32**, **33**, **36**, and **37**.

We denote these "potentially parallel" nodes as *interleaving predecessors*. This notion can easily be defined by means of the function $ParRel$ mapping a graph of $\mathcal{G}_{\mathcal{C}}(G^*)$ to the set of its *parallel relatives*, i.e., the set of component graphs which are executed in parallel, i.e.,

$$ParRel : \mathcal{G}_{\mathcal{C}}(G^*) \to \mathcal{P}(\mathcal{G}_{\mathcal{C}}(G^*))$$

is defined by

$$ParRel(G) =_{df} \mathcal{G}_{\mathcal{C}}(pfg(G)) \backslash G \ \cup \begin{cases} \emptyset & \text{if } pfg(G) \in \mathcal{G}_{\mathcal{P}}^{max}(G^*) \\ ParRel(cfg(pfg(G))) & \text{otherwise} \end{cases}$$

where $\mathcal{P}$ denotes the power set operator. The set of interleaving predecessors of a node $n \in N^*$ is then given by the function

$$ItlvgPred_{G^*} : N^* \to \mathcal{P}(N^*)$$

which is defined by:

$$ItlvgPred_{G^*}(n) =_{df} \begin{cases} Nodes(ParRel(cfg(n))) & \text{if } n \in Nodes(\mathcal{G}_{\mathcal{C}}(G^*)) \\ \emptyset & \text{otherwise.} \end{cases}$$

For illustration see Figure 6, which shows the sets of static and interleaving predecessors of the nodes **9** and **33** of Figure 1. We have:

$$pred_{G^*}(\mathbf{9}) = \{\mathbf{8}\} \quad \text{and} \quad ItlvgPred_{G^*}(\mathbf{9}) = \{\mathbf{12}, \ldots, \mathbf{17}\}$$

and

$$pred_{G^*}(\mathbf{33}) = \{\mathbf{32}\} \quad \text{and} \quad ItlvgPred_{G^*}(\mathbf{33}) = \{\mathbf{25}, \ldots, \mathbf{31}, \mathbf{36}, \mathbf{37}\}.$$

The set of interleaving predecessors of a node n is a subset of the set of its *coexecutable* nodes introduced by Callahan and Subhlok [1988] and considered also by Grunwald and Srinivasan [1993b]: two nodes are coexecutable if they are interleaving predecessors of each other or if they are connected via a path in the control flow graph.

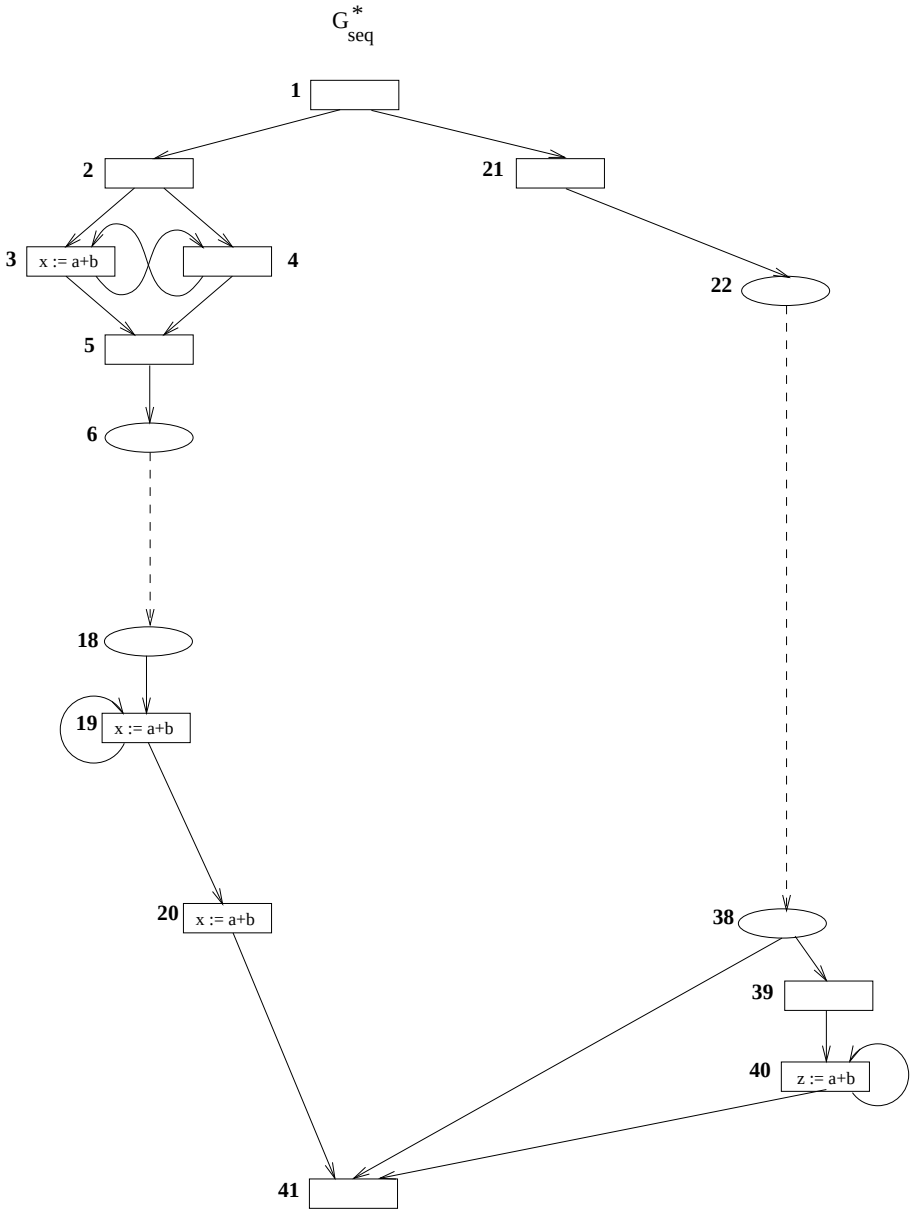

Fig. 4. The sequentialized version G^*_{seq} of G^*.

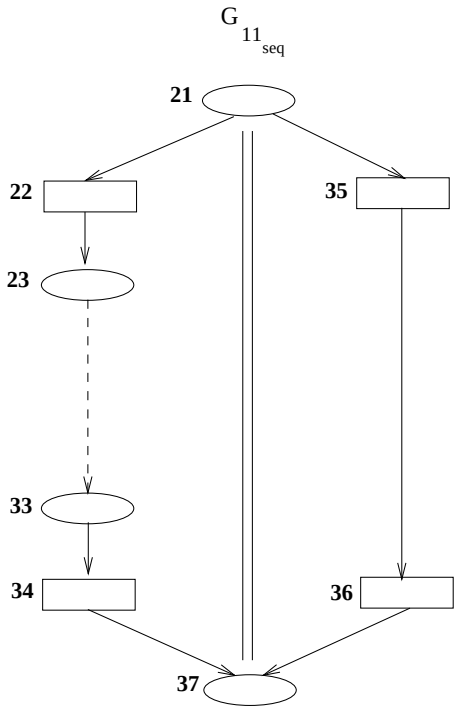

Fig. 5. $\{G_{seq} \mid G \in \mathcal{G}_{\mathcal{P}}(G^*) \wedge rank(G) = 1\} = \{G_{11_{seq}}\}$.

Remark 3.1.1. The definition of interleaving predecessors as given above is a consequence of considering assignments as atomic operations. This implies that parallel processes always have a common and consistent memory view. Thus, our approach even models architectures relying on strong memory consistency correctly.

Program Paths of Parallel Programs. As mentioned before, the interleaving semantics of a parallel imperative programming language can be defined via a translation which reduces parallel programs to (much larger) nondeterministic programs. However, there is also an alternative way 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 [1981]. They start by interpreting every branch statement purely nondeterministically, which allows to simply use the definition of *finite path* as introduced in Section 2. 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 3.1.2. (*G-Well-Formedness*). Let G be a (parallel) flow graph, and let $p =_{df} (n_1, \ldots, n_q)$ be a sequence of nodes. Then p is *G-well-formed* if and only if

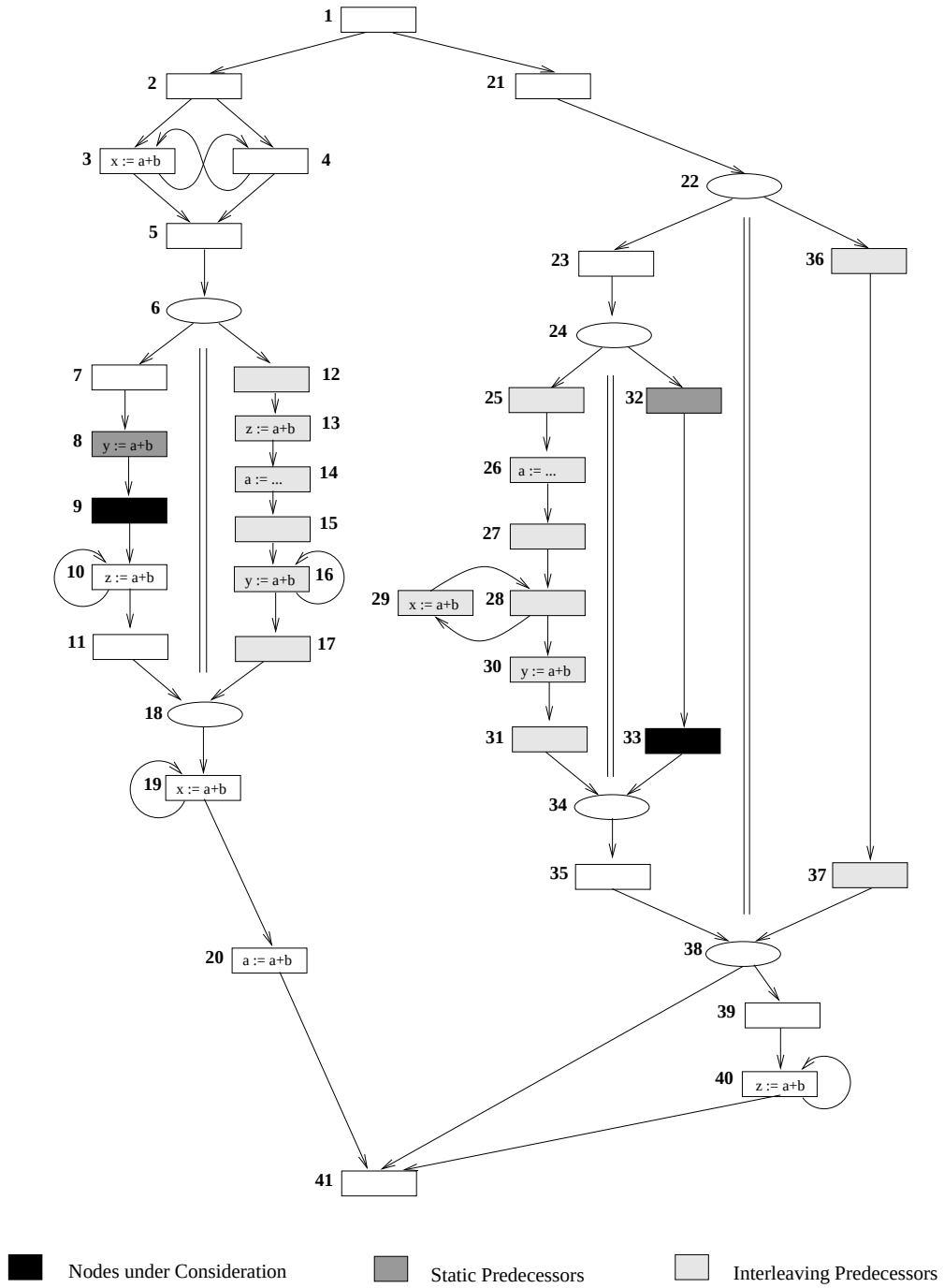

Fig. 6. Static and interleaving predecessors.

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

(1) $\mathbf{s}^*$ *to* $\mathbf{e}^*$ if and only if p is G^*-well-formed.

(2) 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

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

3.2 Data Flow Analysis of Parallel Programs

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 $\mathcal{C}$.

As in the sequential case it is straightforward to extend a local semantic functional to cover also finite parallel paths. Thus, given a node n of a parallel program G^*, the parallel version of the *MOP*-solution is clear, and as in the sequential case, it marks the desired solution of the considered data flow problem:

The *PMOP*-Solution:

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

Referring to the nondeterministic "product program," which explicitly represents all the possible interleavings, would allow us to straightforwardly adapt the sequential situation and to state a Coincidence Theorem. This, however, would not be of much practical use, as this approach would require that one defines the *MFP*-solution relative to the potentially exponential product program. Fortunately, as we will see in the following section, for bitvector problems there exists an elegant and efficient way out.

3.3 Bitvector Analyses

Unidirectional bitvector problems can be 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 4 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 of a variable or availability of a term to powerful program optimizations like code motion and partial dead-code elimination (cf. Section 1).

We are now going to show how to optimize the effort for computing the *PMOP*-solution for this class of 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 3.3.1.

(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.*

Remark 3.3.2. Note that for functions from $\mathcal{B}^k$ to $\mathcal{B}$, $k \geq 2$, monotonicity usually does not imply distributivity, which is an essential premise of our main result, the Parallel Bitvector Coincidence Theorem 3.3.7. Fortunately, all the local semantic functions involved in problems like liveness, availability, very busyness, definition-use chaining, code motion, assignment motion, partial dead-code elimination, and strength reduction are functions of $\mathcal{F}_{\mathcal{B}}$. In fact, we do not know of any practically relevant bitvector problem which cannot be encoded by functions of $\mathcal{F}_{\mathcal{B}}$.

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 3.3.3. (MAIN LEMMA). *Let $f_i : \mathcal{F}_{\mathcal{B}} \to \mathcal{F}_{\mathcal{B}}$, $1 \leq i \leq q$, $q \in \mathbb{N}$, 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 this lemma for our application 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 $m \in ItlvgPred_{G^*}(n)$, 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 $ItlvgPred_{G^*}(n)$. 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 $ItlvgPred_{G^*}(n)$ 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

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

Intuitively, *NonDestructible* indicates that no node of a parallel component destroys the property under consideration, i.e., for all $m \in ItlvgPred_{G^*}(n)$ holds:

$$[\![\, m \,]\!] \neq Const_{ff}.$$

Note that only the constant function with the precomputed value of this predicate is used in Definition 3.3.6 to model interference, and in fact, Theorem 3.3.7 guarantees that this modeling is sufficient. Obviously, this predicate is easily and efficiently computable. Algorithm B.1 computes it as a side result. For illustration see Figure 7, which is annotated with the local semantic functional for computing the set of "up-safe" program points with respect to the computation "$a + b$" (cf. Section 4.1). In this example, the predicate *NonDestructible* is violated for example for the nodes **9** and **33**, because they both have an interleaving predecessor (node **14** and node **26**, respectively (see Figure 6)), which is annotated with the local semantic function $Const_{ff}$. On the other hand, *NonDestructible* for example holds for the nodes **14** and **26**, as all of their interleaving predecessors (nodes **7**, ..., **11** and nodes **30**, **32**, **36**, and **37**, respectively) are annotated with $Const_{tt}$ or $Id_{\mathcal{B}}$.

Synchronization. Besides taking care of possible interference, we also need to take care of the synchronization required by nodes in N^*_X: control may only leave a parallel statement after all parallel components terminated. The corresponding information can be computed by a hierarchical algorithm that only considers purely sequential programs. The central idea coincides with that of interprocedural analysis [Knoop and Steffen 1992]: 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.[4] The following definition, which is illustrated in Section 4, describes the complete three-step procedure:

(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 of Definition 2.2.3.1.

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

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

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

[4] Also, in Srinivasan et al. [1993] parallel statements are investigated in an innnermost fashion.

This three-step algorithm is a straightforward hierarchical adaptation of the algorithm for computing the functional version of the *MFP*-solution for the sequential case. Only the second step realizing the synchronization at nodes in N^*_X needs some explanation, which is summarized in the following lemma.

LEMMA 3.3.4. *The PMOP-solution of a parallel flow graph* $G \in \mathcal{G}_{\mathcal{P}}(G^*)$ *that only consists of purely sequential parallel components* $G_1, \ldots, G_k$ *is given by:*

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

Also the proof of this lemma is a consequence of the Main Lemma 3.3.3. As a single statement is responsible for the entire effect of a path, 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 3.3.4.

Now the following theorem can be proved by means of a straightforward inductive extension of the functional version of the sequential Coincidence Theorem 2.2.2.2, which is tailored to cover complete paths, i.e., paths going from the start to the end of a parallel statement:

THEOREM 3.3.5. (THE HIERARCHICAL COINCIDENCE THEOREM). *Let* $G \in \mathcal{G}_{\mathcal{P}}(G^*)$ *be a parallel flow graph, and let* $[\![\ ]\!] : N^* \to \mathcal{F}_{\mathcal{B}}$ *be a local semantic functional. Then we have:*

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

After this hierarchical preprocess the following modification of the equation system for sequential bitvector analyses leads to optimal results:

Definition 3.3.6. The functional $[\![\ ]\!] : N^* \to \mathcal{F}_{\mathcal{B}}$ is defined as the greatest solution of the equation system given by:[5]

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

This allows us to define the $PMFP_{BV}$-solution, a fixed-point solution for the bitvector case, in the following fashion:

The $PMFP_{BV}$-Solution:

[5]Note that $[\![\ ]\!]$ is the straightforward extension of the functional defined in Definition 2.2.3.1. Thus the overloading of notation is harmless, as no reference to the sequential version is made in this definition.

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

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

As in the sequential case the $PMFP_{BV}$-solution is practically relevant, because it can efficiently be computed (see Algorithm B.1 in Appendix B). The following theorem now establishes that it also coincides with the desired *PMOP*-solution.

THEOREM 3.3.7. (THE PARALLEL BITVECTOR COINCIDENCE THEOREM). *Let $G^* = (N^*, E^*, \mathbf{s}^*, \mathbf{e}^*)$ be a parallel flow graph, and let $[\![\ ]\!] : N^* \to \mathcal{F}_{\mathcal{B}}$ be a local semantic functional. Then we have that the PMOP-solution and the $PMFP_{BV}$-solution coincide, i.e.,*

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

PROOF. The proof follows the same pattern as for the known versions of the coincidence theorem [Kam and Ullman 1977; Knoop and Steffen 1992]: induction on the number of steps of the fixed-point iteration for computing the $PMFP_{BV}$-solution for establishing

$$PMOP_{(G^*,[\![\ ]\!])} \Rightarrow PMFP_{BV(G^*,[\![\ ]\!])} \tag{A}$$

and induction on the length of a parallel path for the converse implication

$$PMFP_{BV(G^*,[\![\ ]\!])} \Rightarrow PMOP_{(G^*,[\![\ ]\!])}. \tag{B}$$

Whereas the proof of (A) is only slightly altered, the proof of (B) requires some extra effort.

Let $b \in \mathcal{B}$ and $n \in N^*$, and let us assume for (A) without loss of generality that $PMOP_{(G^*,[\![\ ]\!])}(n)(b) = tt$. Obviously, for every statement m, which can be executed in parallel with n, there exists a path in $\mathbf{PP}_{G^*}[\mathbf{s}^*, n[$, having m as its last component, i.e.,

$$\mathbf{PP}_{G^*}[\mathbf{s}^*, n[\cap \mathbf{PP}_{G^*}[\mathbf{s}^*, m] \neq \emptyset.$$

Thus $NonDestructible(n) = tt$. Now the rest of the proof is the "standard induction" on the number of fixed-point iterations mentioned above [Kam and Ullman 1977; Knoop and Steffen 1992], refined to take care of the distinction between "ordinary" nodes and nodes taken from N^*_X, which requires the application of Theorem 3.3.5.

For (B), we can assume without loss of generality that

$$PMFP_{BV(G^*,[\![\ ]\!])}(n)(b) = tt \tag{$*$}$$

holds. In particular, this means that $NonDestructible(n) = tt$, i.e., none of the statements $m \in ItlvgPred_{G^*}(n)$ satisfies $[\![\, m \,]\!] = Const_{ff}$. Now it is the Main Lemma 3.3.3 which guarantees that this is already sufficient to guarantee that the standard sequential bitvector analysis is not interfered by any parallel statement. The proof is a slightly modified version of the standard induction on the path length:

Let $p = (n_1, \ldots, n_k) \in \mathbf{PP}_{G^*}[\mathbf{s}^*, n[$ be a parallel path. Then we must show that

$$[\![n_k]\!] \circ \cdots \circ [\![n_1]\!](b) \;=\; tt.$$

In the case of $k = 0$ this is trivial, as only the start node $\mathbf{s}^*$ is reachable. Thus our assumption forces $b = tt$ as desired.

For $k \geq 1$ we need to distinguish the "standard" cases from the case where $n \in N^*_X$. For the "standard" case, let $0 \leq l \leq k$ be the index of the last step of p that was done by a predecessor $m \in pred_{G^*}(n)$, i.e., $[\![n_l]\!] = [\![m]\!]$. Such a step must exist, as $k \geq 1$, which excludes $n = \mathbf{s}^*$, and we know by induction:

$$PMFP_{BV(G^*,[\![\ ]\!])}(m)(b) \Rightarrow [\![n_{l-1}]\!] \circ \cdots \circ [\![n_1]\!](b)$$

and therefore by monotonicity

$$[\![m]\!](PMFP_{BV(G^*,[\![\ ]\!])}(m)(b)) \Rightarrow [\![n_l]\!] \circ \cdots \circ [\![n_1]\!](b).$$

On the other hand, assumption $(*)$ forces

$$[\![m]\!](PMFP_{BV(G^*,[\![\ ]\!])}(m)(b)) \;=\; tt$$

and therefore, together, as all the n_i, $l + 1 \leq i \leq k$, are members of $ItlvgPred_{G^*}(n)$

$$[\![n_k]\!] \circ \cdots \circ [\![n_1]\!] \;=\; tt$$

which completes the proof for the standard case.

Thus it remains to consider the case, where $n \in N^*_X$. In this case, assumption $(*)$ reads as follows:

$$tt \;=\; PMFP_{BV(G^*,[\![\ ]\!])}(n)(b) \;=\; [\![\, pfg(n) \,]\!]^* \circ [\![\, start(pfg(n)) \,]\!].$$

Now let $0 \leq l \leq k$ be the index corresponding to $m = start(pfg(n))$. Then we can apply the induction hypothesis in order to obtain:

$$PMFP_{BV(G^*,[\![\ ]\!])}(m)(b) \Rightarrow [\![n_{l-1}]\!] \circ \cdots \circ [\![n_1]\!](b).$$

Now the application of Theorem 3.3.5 allows to complete the proof as in the standard case. □

Intuitively, the (sequential) Coincidence Theorem 2.2.2.2 can be read as that unidirectional distributive data flow analysis problems allow to model the confluence of control flow by merging the corresponding data flow informations during the iterative computation of the *MFP*-solution without losing accuracy. The intuition behind the Parallel Bitvector Coincidence Theorem 3.3.7 is the same, only the correspondence between control flow and program representation is more complicated due to the interleaving and synchronization effects.

3.4 Performance and Implementation

Our generic algorithm is based on a functional version of an *MFP*-solution, as it is common for interprocedural analyses. However, as bitvector algorithms only deal with Boolean values, proceeding argumentwise would simply require to apply a standard bitvector algorithm twice. In particular, for regular program structures, all the nice properties of bitvector algorithms apply. In fact, for the standard version of Algorithm B.1 a single execution is sufficient, as we can start here with the

same start information as the standard sequential analysis. Thus, even if we count the effort for computing the predicate *NonDestructible* separately, our analysis would simply be a composition of four standard bitvector analyses. In practice, however, our algorithm behaves much better, as the existence of a single destructive statement allows us to skip the analysis of large parts of the program. In fact, in our experience, the parallel version often runs faster than the sequential version on a program of similar size.

The same argumentation also indicates a way for a cheap implementation on top of existing bitvector algorithms. However, we recommend the direct implementation of the functional version, which to our experience, runs even faster than the decomposed standard version. This is not too surprising, as the functional version only needs to consider one additional value and does not require the argumentwise application.

4. APPLICATIONS

As mentioned before, unidirectional bitvector problems are highly relevant in practice because of their broad scope of applications ranging from simple analyses like determining the liveness of variables or the availability of terms to powerful program optimizations like code motion, partial dead-code elimination, assignment motion, and strength reduction. Using the methods of Section 3, all these techniques can now be made available for parallel programs. In this section we demonstrate this by sketching straightforward parallel extensions of the *code motion* and the *partial dead-code elimination* algorithms of Knoop et al. [1994a; 1994b]. We conjecture that, like their sequential counterparts, these algorithms are unique in optimally eliminating the partially redundant expressions and partially dead assignments in a parallel program, respectively.

4.1 Code Motion

Code motion improves the runtime 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 certain program points. For sequential programs it is well known that computationally optimal results can be obtained by placing the computations *as early as possible* in a program, while maintaining its semantics [Knoop et al. 1992; 1994a]. In the following we illustrate the essential steps of an algorithm for the parallel setting, which directly evolves from the *busy-code-motion* (*BCM*) transformation of Knoop et al. [1994a], by means of the example of Figure 1. In this example our algorithm, called the BCM_{PP}-transformation, is unique to achieve the optimization of Figure 11: it eliminates the partially redundant computations of $a + b$ at the nodes **3**, **13**, **16**, **19**, **20**, **29**, **30**, and **40** by moving them to the nodes **2**, **15**, and **27**, but it does not touch the partially redundant computations of $a + b$ at the nodes **8** and **10**, which cannot safely be eliminated.

As in the sequential case, placing the computations as early as possible in a program requires to compute the set of program points where a computation is *up-safe* and *down-safe*, i.e., where it has been computed on every program path reaching the program point under consideration, and where it will be computed

on every program continuation reaching the program's end node.[6] For the ease of presentation we here assume that the parallel statements of the argument program are free of "recursive" assignments, i.e., assignments whose left-hand-side variable occurs in its right-hand-side term, which can be handled as well but require a slightly refined treatment.

The DFA-problems for up-safety and down-safety are specified by the local semantic functionals $[\![\ ]\!]_{us}$ and $[\![\ ]\!]_{ds}$, where *Comp* and *Transp* are two local predicates, which are true for a node n with respect to a computation t, if t occurs in the right-hand-side term of the statement of n, and if no operand of t is modified by it, respectively.

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

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

Note that these are the very same functionals as in the sequential case because the effect of interference is completely taken care of by the corresponding versions of the predicate *NonDestructible*, which are automatically derived from the definitions of the local semantic functionals. In the literature the definitions of $[\![n]\!]_{us}$ and $[\![n]\!]_{ds}$ are usually given in the following equivalent form:

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

and

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

The functionals for up-safety and down-safety can directly be fed into the generic Algorithm B.1 computing the *PMFP*-solutions of these two properties. For the predicate up-safe this is illustrated in some detail in Figures 7, 8, and 9, which illustrate the hierarchical preprocess for computing the semantics of the parallel statements of G^*, i.e., of the subgraphs $G \in \mathcal{G}_{\mathcal{P}}(G^*)$. Figure 7 shows the flow graph of Figure 1 enhanced with the local semantic functions for up-safety with respect to the computation $a + b$. In the first step the hierarchical preprocess computes the semantics of all `par` statements of rank 0. The corresponding results are displayed in Figure 8. Subsequently, these results are used for computing the semantics of the single `par` statement of rank 1, which is illustrated in Figure 9.

After the semantics of all subgraphs $G \in \mathcal{G}_{\mathcal{P}}(G^*)$ with respect to up-safety and down-safety have been computed, the $PMFP_{BV}$-solution for these two properties can be computed essentially as in the sequential case. The result of the corresponding fixpoint computations is illustrated in Figure 10, which shows the set of

[6] Up-safety and down-safety are also known as *availability* and *anticipability* (*very busyness*), respectively.

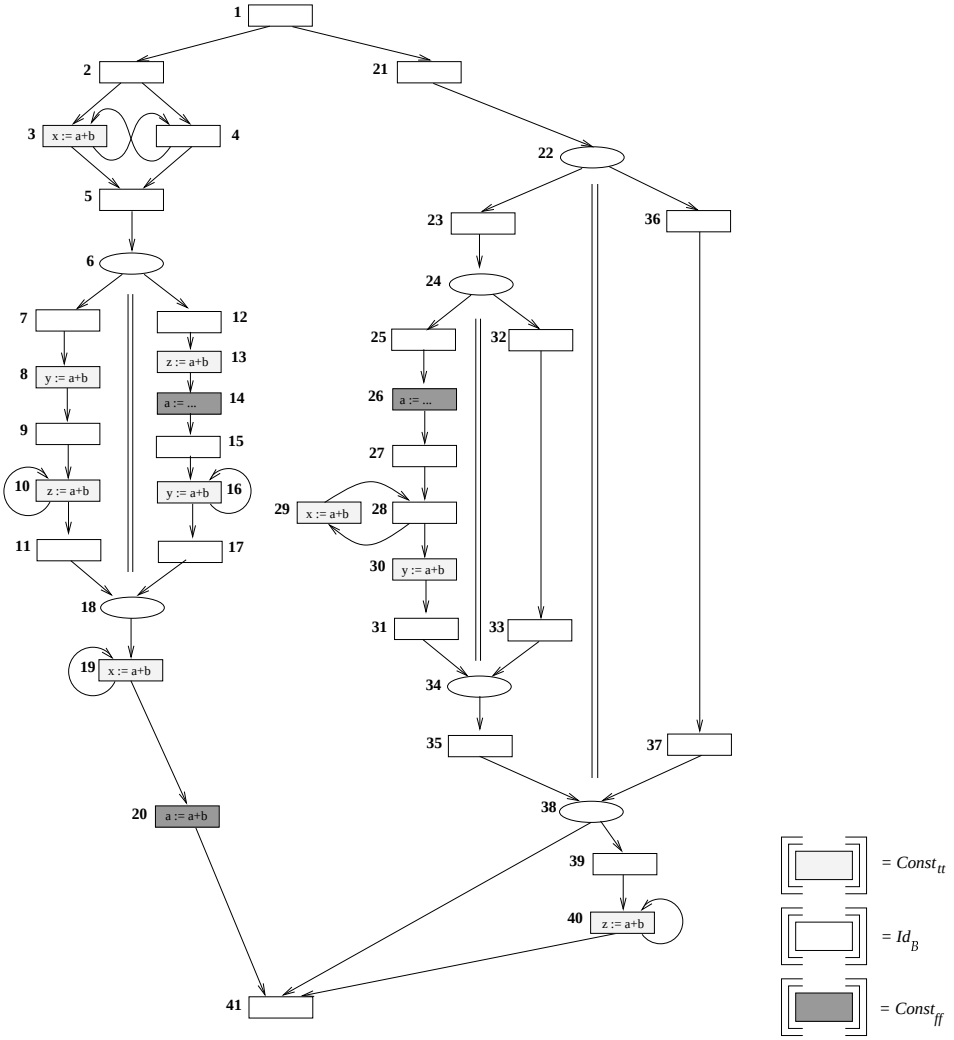

Fig. 7. G^* with the local semantic functional for up-safety $[\![\ ]\!]_{us}$ wrt $a+b$.

up-safe and down-safe program points of G^*, and the set of program points, where a placement of $a+b$ is *earliest*, i.e., where a computation has to be inserted and where an original computation of $a+b$ must be *replaced*.

As in the sequential case, down-safe start nodes are "earliest," as well as other down-safe but not up-safe nodes which either possess an "unsafe" predecessor (see node **2**) or a predecessor modifying an operand of the computation under consideration (see nodes **15** and **27**).

After inserting an initialization statement at the entry of each earliest node, every original computation occurring in a safe node is *replaced* by the corresponding temporary, as illustrated in Figure 10. Note that the replacement condition is stronger than in the sequential setting, where all original computations are replaced. The

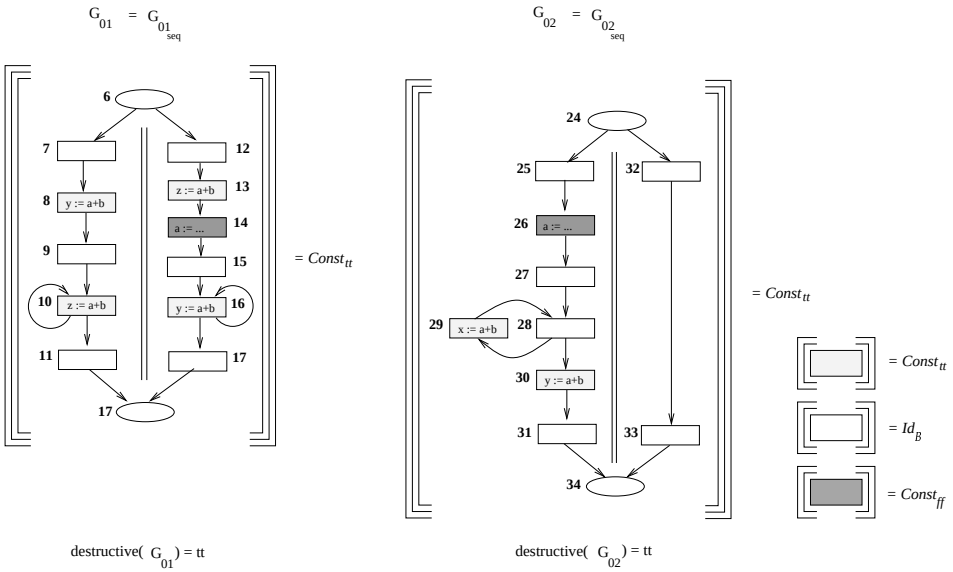

Fig. 8. Up-safety: After the first iteration of the outermost for-loop of GLOBEFF.

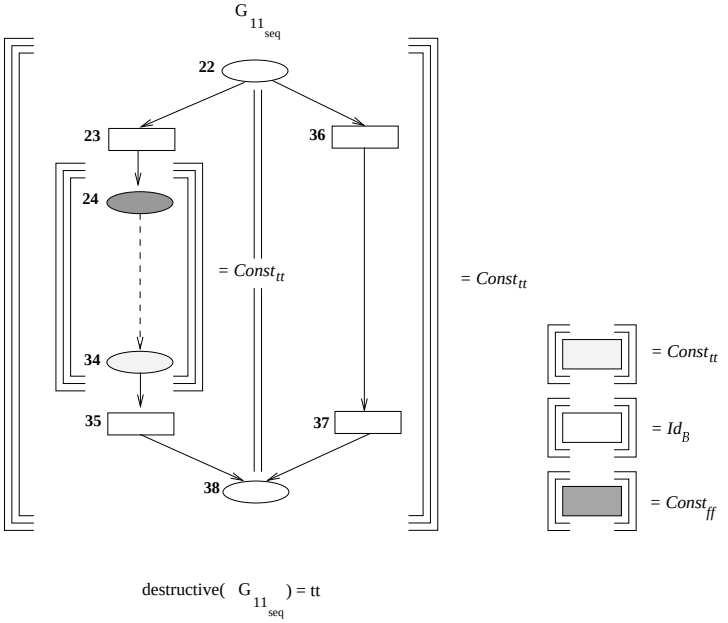

Fig. 9. Up-safety: After the second iteration of the outermost for-loop of GLOBEFF.

point here is that in the sequential setting the initialization of the temporaries at the "earliest" computations points guarantees that all paths from the start node of the program reaching a node with an original computation go through an initialization site of the temporary which is not followed by a modification of one of the operands of the computation under consideration. This, however, can be violated in the parallel setting because of interference between parallel components. In Figure 10 for example, the nodes **8** and **10** are not safe because of the possibly interfering statement at node **14**.

Figure 11, finally, shows the promised result of the BCM_{PP}-transformation for the program of Figure 1.

4.2 Partial Dead-Code Elimination

Intuitively, an assignment occurrence in a program is *dead* if on every program continuation starting at the assignment under consideration and reaching the end node every use of the assignment's left-hand-side variable is preceded by a redefinition of it. It is called *partially dead*, if it is dead along some program continuations. In Knoop et al. [1994b] it has been shown how to eliminate partially dead assignments in a sequential program by first moving them as far as possible in the direction of the control flow and second by eliminating all dead assignment occurrences. Optimal results can be obtained by repeating this two-step procedure until the program stabilizes, which is necessary in order to capture the second-order effects of partial dead-code elimination.

The central components of the complete algorithm are the procedures for assignment sinking and the detection of dead variables, and below we present the local semantic functionals $[\![\ ]\!]_{dl}$ and $[\![\ ]\!]_{dd}$ of the underlying unidirectional bitvector DFA-problems for assignment sinking and detecting dead variables, respectively, where $Used$, Mod, $LocDelay$, and $LocBlocked$ are local predicates of nodes. The predicates $Used$ and Mod are true for a node n, if the variable under consideration occurs on the right-hand side and the left-hand side of the statement of node n, respectively. $LocDelay$ holds for a node, if it contains an occurrence of the assignment pattern under consideration whose execution can be postponed to the end of this node, and $LocBlocked$, finally, is true, if the sinking of the assignment pattern under consideration is blocked by the statement of the argument node:

$$[\![n]\!]_{dd} =_{df} \begin{cases} Const_{tt} & \text{if } \neg Used(n) \wedge Mod(n) \\ Id_{\mathcal{B}} & \text{if } \neg(Used(n) \vee Mod(n)) \\ Const_{ff} & \text{otherwise} \end{cases}$$

$$[\![n]\!]_{dl} =_{df} \begin{cases} Const_{tt} & \text{if } LocDelay(n) \\ Id_{\mathcal{B}} & \text{if } \neg(LocDelay(n) \vee LocBlocked(n)) \\ Const_{ff} & \text{otherwise} \end{cases}$$

or equivalently

$$\forall\, b \in \mathcal{B}.\ [\![n]\!]_{dd}(b) =_{df} \neg Used(n) \wedge (b \vee Mod(n))$$

and

$$\forall\, b \in \mathcal{B}.\ [\![n]\!]_{dl}(b) =_{df} LocDelay(n) \vee (b \wedge \neg LocBlocked(n)).$$

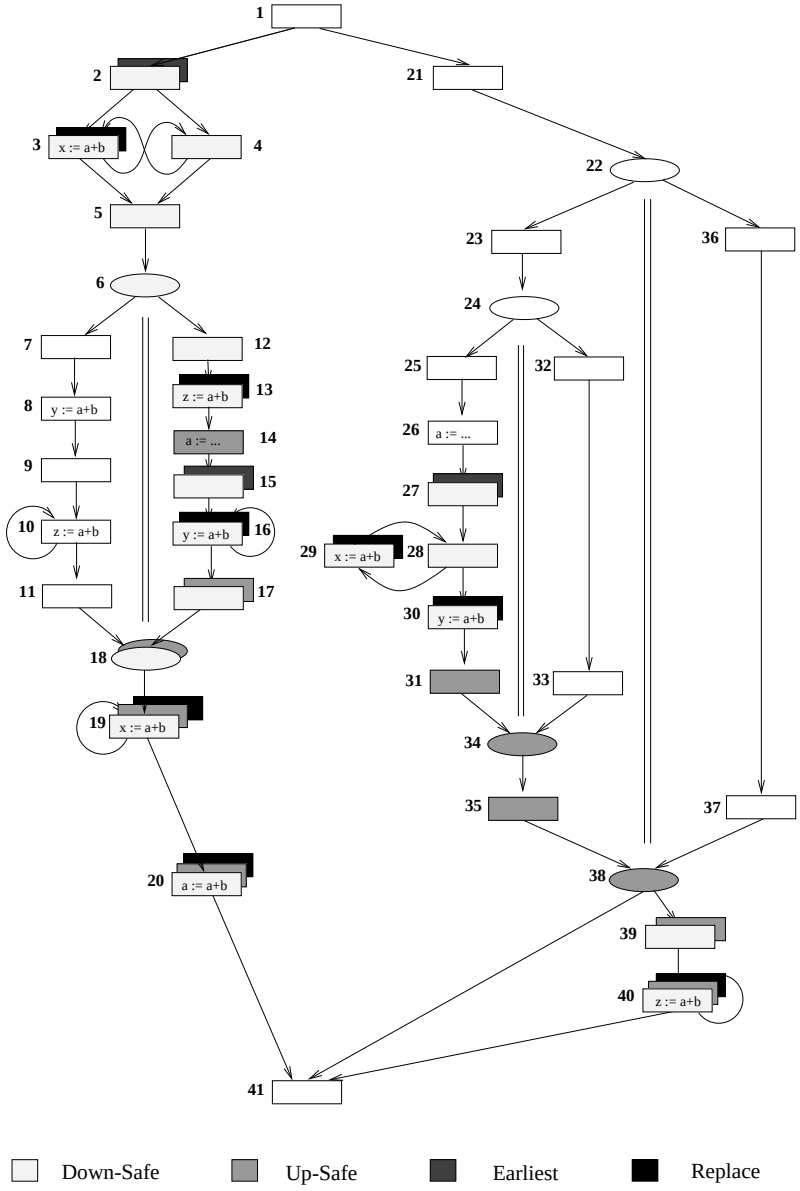

Fig. 10. Down-safe, up-safe, earliest, and replacement program points of $a + b$.

5. CONCLUSIONS

We have shown how to construct for parallel programs with shared memory optimal analysis algorithms for unidirectional bitvector problems that are as efficient as their purely sequential counterparts and which can easily be implemented. At the first sight, the existence of such algorithms is rather surprising, as the interleaving semantics underlying our programming language is an indication for an exponential

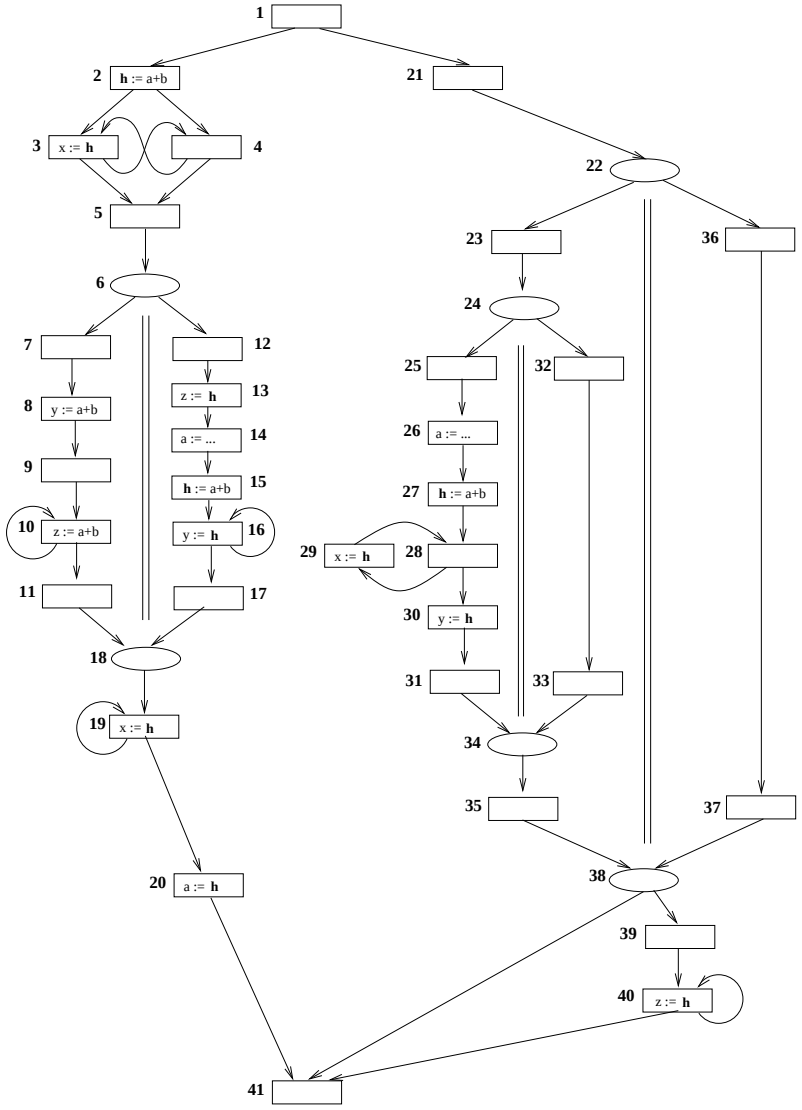

Fig. 11. The result of the BCM_{PP}-transformation.

effort. However, the restriction to bitvector analysis constrains the possible ways of interference in such a way that we could construct a generic fixed-point algorithm that directly works on the parallel program without taking any interleavings into account. The algorithm is implemented on the *Fixpoint Analysis Machine* of Steffen et al. [1995]. A variant of the code motion transformation of Section 4 is implemented in the ESPRIT project COMPARE number 5933 [Vollmer 1994; 1995].

APPENDIX

A. COMPUTING THE *MFP*-SOLUTION

Algorithm A.1. (*Computing the MFP-Solution*).

Input. A flow graph $G = (N, E, \mathbf{s}, \mathbf{e})$, and a local semantic functional $[\![\ ]\!] : N \to \mathcal{F}_{\mathcal{C}}$, where $\mathcal{F}_{\mathcal{C}}$ denotes the set of monotonic functions on a complete lattice $\mathcal{C}$. Additionally, a function $f_{init} \in \mathcal{F}_{\mathcal{C}}$, which reflects the assumptions on the context in which the procedure under consideration is called. Usually, f_{init} is given by the identity $Id_{\mathcal{C}}$ on $\mathcal{C}$.

Output. An annotation of G with functions $[\![n]\!] \in \mathcal{F}_{\mathcal{C}}$, $n \in N$, representing the greatest solution of the equation system of Definition 2.2.3.1. After the termination of the algorithm the functional $[\![\ ]\!]$ satisfies:

$$\forall\, n \in N.\ [\![n]\!] = MFP_{(G,[\![\ ]\!])}(n) \sqsubseteq MOP_{(G,[\![\ ]\!])}(n).$$

Remark. The function $Const_{\top}$, which maps every argument to the greatest element $\top$ of $\mathcal{C}$, denotes the "universal" function which is assumed to "contain" every other function of $\mathcal{F}_{\mathcal{C}}$.

BEGIN $MFP(G, [\![\ ]\!], f_{init})$ **END**.

where

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

(*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.

B. COMPUTING THE $PMFP_{BV}$-SOLUTION

Algorithm B.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 ff, 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 3.3, and with functions $[\![n]\!] \in \mathcal{F}_{\mathcal{B}}$, $n \in N^*$, representing the greatest solution of the equation system of Definition 3.3.6. 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 $[\![G_{seq}]\!]^* \in \mathcal{F}_{\mathcal{B}}$, $G \in \mathcal{G}_{\mathcal{C}}(G^*)$, store the global effect of G during the hierarchical computation of the $PMFP_{BV}$-solution. The global variables $destructive(G_{seq})$, $G \in \mathcal{G}_{\mathcal{C}}(G^*)$, store whether G contains a node n with $[\![n]\!] = Const_{ff}$. These variables are used to compute the value of the predicate *NonDestructible* of Section 3.3.

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 $G'' = (N'', E'', \mathbf{s}'', \mathbf{e}'') \in \{G'''_{seq} \mid G''' \in \mathcal{G}_{\mathcal{C}}(G')\}$ **DO**
LET $\forall\, n \in N''.\ [\![n]\!]'' =$

$$\begin{cases} Id_{\mathcal{B}} \sqcap Const_{\forall\, \bar{G} \in \mathcal{G}_{\mathcal{C}}(pfg(n)).\ \neg\, destructive(\bar{G})} & \text{if } n \in N^*_N \\ [\![pfg(n)]\!]^* & \text{if } n \in N^*_X \\ \overline{[\![n]\!]} & \text{otherwise} \end{cases}$$

BEGIN
$destructive(G'') := (\ |\{ n \in N'' \mid [\![n]\!]'' = Const_{ff} \}| \geq 1\ )$;
$MFP(G'', [\![\ ]\!]'', Id_{\mathcal{B}})$;
$[\![G'']\!]^* := [\![end(G'')]\!]$
END
OD;

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

OD
OD
END.

```
PROCEDURE PMFP_BV (G = (N, E, s, e) : ParallelFlowGraph;
                   [[ ]] : N → F_B  : LocalSemanticFunctional;
                   f_start : F_B;
                   destructive : B);
VAR f : F_B;
BEGIN
  IF destructive THEN FORALL n ∈ N DO [[ n ]] := Const_ff OD
  ELSE
    ( Initialization of the annotation array [[ ]] and the variable workset )
    FORALL n ∈ Nodes(G_seq)\{s} DO [[ n ]] := Const_tt OD;
    [[ s ]] := f_start;
    workset := { n ∈ Nodes(G_seq) | n ∈ N*_N ∪ {s} ∨ ¬[[ n ]] = Const_ff };

    ( Iterative fixed-point computation )
    WHILE workset ≠ ∅ DO
      LET n ∈ workset
        BEGIN
          workset := workset\{ n };
          IF n ∈ N\N*_N
            THEN
              f := ¬[[ n ]] ∘ [[ n ]];
              FORALL m ∈ succ_G(n) DO
                IF [[ m ]] ⊐ f
                  THEN [[ m ]] := f; workset := workset ∪ { m } FI
              OD
            ELSE
              FORALL G' ∈ G_C(pfg(n)) DO
                PMFP_BV(G', [ ], [[ n ]], ⊓_{G''∈G_C(pfg(n))\{G'}} destructive(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.
```

Let $[\![n]\!]_{alg}$, $n \in N^*$, denote the final values of the corresponding variables after the termination of Algorithm B.1, and $[\![n]\!]$, $n \in N^*$, the greatest solution of the equation system of Definition 3.3.6, then we have:

THEOREM B.2. $\forall n \in N^*.\ [\![n]\!]_{alg} = [\![n]\!]$.

ACKNOWLEDGMENTS

The authors would like to thank Marion Klein and the anonymous referees for their constructive comments.

References

CALLAHAN, D. AND SUBHLOK, J. 1988. Static analysis of low-level synchronization. In *Proceedings of the 1st ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging (WPDD'88)*. ACM SIGPLAN Notices, vol. *24*,1. Madison, Wisconsin, 100 – 111.

CHOW, J.-H. AND HARRISON, W. L. 1992. Compile time analysis of parallel programs that share memory. In *Conference Record of the 19th ACM Symposium on Principles of Programming Languages (POPL'92)*. Albuquerque, New Mexico, 130 – 141.

CHOW, J.-H. AND HARRISON, W. L. 1994. State space reduction in abstract interpretation of parallel programs. In *Proceedings of the International Conference on Computer Languages (ICCL'94)*. Toulouse, France, 277 – 288.

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

COUSOT, P. AND COUSOT, R. 1984. Invariance proof methods and analysis techniques for parallel programs. In *Automatic Program Construction Techniques*, A. W. Biermann, G. Guiho, and Y. Kodratoff, Eds. Macmillan Publishing Company, Chapter 12, 243 – 271.

DHAMDHERE, D. M. 1989. A new algorithm for composite hoisting and strength reduction optimisation (+ corrigendum). *International Journal of Computer Mathematics 27*, 1 – 14 (+ 31 – 32).

DHAMDHERE, D. M., ROSEN, B. K., AND ZADECK, F. K. 1992. How to analyze large programs efficiently and informatively. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'92)*. ACM SIGPLAN Notices, vol. *27*,7. San Francisco, California, 212 – 223.

DRECHSLER, K.-H. AND STADEL, M. P. 1993. A variation of Knoop, Rüthing and Steffen's lazy code motion. *ACM SIGPLAN Notices 28*, 5, 29 – 38.

DURI, S., BUY, U., DEVARAPALLI, R., AND SHATZ, S. M. 1993. Using state space methods for deadlock analysis in ada tasking. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (SSTA'93)*. Software Engineering Notes, vol. 18,3. 51 – 60.

DWYER, M. B. AND CLARKE, L. A. 1994. Data flow analysis for verifying properties of concurrent programs. In *Proceedings of the 2nd ACM SIGSOFT Symposium on Foundations of Software Engineering (SFSE'94)*. Software Engineering Notes, vol. 19,5. New Orleans, Lousiana, 62 – 75.

GODEFROID, P. AND WOLPER, P. 1991. Using partial orders for the efficient verification of deadlock freedom and safety properties. In *Proceedings of the 3rd International Workshop on Computer Aided Verification (CAV'91)*. Lecture Notes in Computer Science 575. Springer-Verlag, Aalborg, Denmark, 332 – 342.

GRUNWALD, D. AND SRINIVASAN, H. 1993a. Data flow equations for explicitely parallel programs. In *Proceedings of the 4th ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP'93)*. ACM SIGPLAN Notices, vol. *28*,7. San Diego, California, 159 – 168.

GRUNWALD, D. AND SRINIVASAN, H. 1993b. Efficient computation of precedence information in parallel programs. In *Proceedings of the 6th International Conference on Languages and Compilers for Parallel Computing (LCPC'93)*. Lecture Notes in Computer Science 768. Springer-Verlag, Portland, Oregon, 602 – 616.

HECHT, M. S. 1977. *Flow Analysis of Computer Programs*. Elsevier, North-Holland.

JOSHI, S. M. AND DHAMDHERE, D. M. 1982a. A composite hoisting-strength reduction transformation for global program optimization – part I. *International Journal of Computer Mathematics 11*, 21 – 41.

JOSHI, S. M. AND DHAMDHERE, D. M. 1982b. A composite hoisting-strength reduction transformation for global program optimization – part II. *International Journal of Computer*

Mathematics 11, 111 – 126.

KAM, J. B. AND ULLMAN, J. D. 1977. Monotone data flow analysis frameworks. *Acta Informatica 7*, 309 – 317.

KILDALL, G. A. 1972. Global expression optimization during compilation. Tech. Rep. No. 72-06-02, University of Washington, Computer Science Group, Seattle, Washington. Ph.D. dissertation.

KILDALL, G. A. 1973. A unified approach to global program optimization. In *Conference Record of the 1st ACM Symposium on Principles of Programming Languages (POPL'73)*. Boston, Massachusetts, 194 – 206.

KNOOP, J., RÜTHING, O., AND STEFFEN, B. 1992. Lazy code motion. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'92)*. ACM SIGPLAN Notices, vol. *27*,7. San Francisco, California, 224 – 234.

KNOOP, J., RÜTHING, O., AND STEFFEN, B. 1993. Lazy strength reduction. *Journal of Programming Languages 1*, 1, 71–91.

KNOOP, J., RÜTHING, O., AND STEFFEN, B. 1994a. Optimal code motion: Theory and practice. *ACM Transactions on Programming Languages and Systems 16*, 4, 1117–1155.

KNOOP, J., RÜTHING, O., AND STEFFEN, B. 1994b. Partial dead code elimination. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'94)*. ACM SIGPLAN Notices, vol. *29*,6. Orlando, Florida, 147 – 158.

KNOOP, J., RÜTHING, O., AND STEFFEN, B. 1994c. A tool kit for constructing optimal interprocedural data flow analyses. Tech. Rep. MIP-9413, Fakultät für Mathematik und Informatik, Universität Passau, Germany. 26 pages.

KNOOP, J., RÜTHING, O., AND STEFFEN, B. 1995. The power of assignment motion. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'95)*. ACM SIGPLAN Notices, vol. *30*,6. La Jolla, California, 233 – 245.

KNOOP, J. AND STEFFEN, B. 1992. The interprocedural coincidence theorem. In *Proceedings of the 4th International Conference on Compiler Construction (CC'92)*. Lecture Notes in Computer Science 641. Springer-Verlag, Paderborn, Germany, 125 – 140.

KNOOP, J. AND STEFFEN, B. 1993. Efficient and optimal bit-vector data flow analyses: A uniform interprocedural framework. Tech. Rep. 9309, Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität Kiel, Germany. 22 pages.

KNOOP, J., STEFFEN, B., AND VOLLMER, J. 1995a. Parallelism for free: Bitvector analyses $\rightarrow$ no state explosion! In *Proceedings of the 1st International Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'95)*. Lecture Notes in Computer Science 1019. Springer-Verlag, Aarhus, Denmark, 264 – 289.

KNOOP, J., STEFFEN, B., AND VOLLMER, J. 1995b. Parallelism for free: Efficient and optimal bitvector analyses for parallel programs. In *Preliminary Proceedings of the 1st International Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'95)*. BRICS Notes Series NS-95-2. Aarhus, Denmark, 319 – 333.

LONG, D. AND CLARKE, L. 1991. 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 Engineering Notes, vol. 16. Victoria, British Columbia, 21– 35.

MARRIOT, K. 1993. Frameworks for abstract interpretation. *Acta Informatica 30*, 103 – 129.

MCDOWELL, C. E. 1989. A practical algorithm for static analysis of parallel programs. *Journal of Parallel and Distributed Computing 6*, 3, 513 – 536.

MIDKIFF, S. P. AND PADUA, D. A. 1990. Issues in the optimization of parallel programs. In *Proceedings of the International Conference on Parallel Processing (ICPP'90)*. Vol. II. St. Charles, Illinois, 105 – 113.

MOREL, E. 1984. Data flow analysis and global optimization. In *Methods and tools for compiler construction*, B. Lorho, Ed. Cambridge University Press, 289 – 315.

MOREL, E. AND RENVOISE, C. 1979. Global optimization by suppression of partial redundancies. *Communications of the ACM 22*, 2, 96 – 103.

MOREL, E. AND RENVOISE, C. 1981. Interprocedural elimination of partial redundancies. In *Program Flow Analysis: Theory and Applications*, S. S. Muchnick and N. D. Jones, Eds. Prentice Hall, Englewood Cliffs, New Jersey, Chapter 6, 160 – 188.

MUCHNICK, S. S. AND JONES, N. D., Eds. 1981. *Program Flow Analysis: Theory and Applications*. Prentice Hall, Englewood Cliffs, New Jersey.

SHARIR, M. AND PNUELI, A. 1981. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*, S. S. Muchnick and N. D. Jones, Eds. Prentice Hall, Englewood Cliffs, New Jersey, Chapter 7, 189 – 233.

SRINIVASAN, H., HOOK, J., AND WOLFE, M. 1993. Static single assignment form for explicitly parallel programs. In *Conference Record of the 20^{th} ACM Symposium on Principles of Programming Languages (POPL'93)*. Charleston, South Carolina, 260 – 272.

SRINIVASAN, H. AND WOLFE, M. 1991. Analyzing programs with explicit parallelism. In *Proceedings of the 4^{th} International Conference on Languages and Compilers for Parallel Computing (LCPC'91)*. Lecture Notes in Computer Science 589. Springer-Verlag, Santa Clara, California, 405 – 419.

STEFFEN, B., CLASSEN, A., KLEIN, M., KNOOP, J., AND MARGARIA, T. 1995. The fixpoint-analysis machine. In *Proceedings of the 6^{th} International Conference on Concurrency Theory (CONCUR'95)*. Lecture Notes in Computer Science 962. Springer-Verlag, Philadelphia, Pennsylvania, 72 – 87.

VALMARI, A. 1990. A stubborn attack on state explosion. In *Proceedings of the 2^{nd} International Workshop on Computer Aided Verification (CAV'90)*. Lecture Notes in Computer Science 531. Springer-Verlag, New Brunswick, New Jersey, 156 – 165.

VOLLMER, J. 1994. Data flow equations for parallel programs that share memory. Tech. Rep. 2.11.1 of the ESPRIT Project COMPARE number 5933, Fakultät für Informatik, Universität Karlsruhe, Germany.

VOLLMER, J. 1995. 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, 168 – 177.

WOLFE, M. AND SRINIVASAN, H. 1991. Data structures for optimizing programs with explicit parallelism. In *Proceedings of the 1^{st} International Conference of the Austrian Center for Parallel Computation*. Lecture Notes in Computer Science 591. Springer-Verlag, Salzburg, Austria, 139 – 156.

Received October 1994; revised August 1995; accepted October 1995