Parallelism for Free: Bitvector Analyses $\Rightarrow$ No State Explosion!

Jens Knoop*
Bernhard Steffen*
Jürgen Vollmer†‡

ABSTRACT One of the central problems in the automatic analysis of distributed or parallel systems is the combinatorial state explosion leading to models, which are exponential in the number of their parallel components. The only known cure for this problem are application specific techniques, which avoid the state explosion problem under special frame conditions. In this paper we present a new such technique, which is tailored to bitvector analyses, which are very common in data flow analysis. In fact, our method allows to adapt most of the practically relevant optimizations for sequential programs, for a parallel setting with shared variables and arbitrary interference between parallel components.

1 Motivation

Parallel systems are of growing interest, as they are more and more supported by modern hardware environments. However, it is very difficult to guarantee their reliability (cf. [MP]): the adaptation of the successful techniques for sequential systems seems inevitably be tied to the combinatorial explosion of the systems' state space leading to models, which are exponential in the number of the parallel components. As a consequence, also classical data flow analysis for parallel programming languages was considered too expensive to be implemented in real programming environments. The only known cure for this problem are application specific techniques, which avoid the state explosion problem under usually very specific frame condi-

*Fakultät für Mathematik und Informatik, Universität Passau, Innstrasse 33, D-94032 Passau, Germany. E-mail: {knoop,steffen}@fmi.uni-passau.de

†Fakultät für Informatik, Institut für Programmstrukturen und Datenorganisation (IPD), Universität Karlsruhe, Vincenz-Prießnitz-Straße 3, D-76128 Karlsruhe, Germany. E-mail: vollmer@ipd.info.uni-karlsruhe.de

‡A preliminary version of this article was published in the preliminary proceedings of TACAS'95 (cf. [KSV1]).

tions. For data flow analysis, this ranges from special heuristics (cf. [McD]) and approaches which require data independence of the parallel components (cf. [GS]) or exclude shared variables (cf. [LC]) over approaches tailored for specific analyses like mutual exclusion or data races (cf. [DC]) to approaches that are based on state space reductions (cf. [CH1, CH2, DBDS, GW, Va]). The latter allow general synchronization mechanisms, but still require the investigation of an appropriately reduced version of the global state space, which is often still unmanageable.

In this paper we show how to construct for unidirectional bitvector analysis problems (which are most prominent in practice) 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 first property is a consequence of a Kam/Ullman-style ([KU]) 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 although the various interleavings of the executions of parallel components are semantically different, they need not be considered during bitvector analysis, which is the key observation of this paper.

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 3.3).

Thus, using our methods all the well-known algorithms for unidirectional bitvector analysis problems can be adapted for parallel programs at almost no cost on the runtime and the implementation side. This is highly relevant in practice as this class of bitvector problems has a broad scope of applications ranging from simple analyses like liveness, availability, very business, reaching definitions, and definition-use chains (cf. [He]) to more sophisticated and powerful program optimizations like code motion (cf. [DS, DRZ, KRS1, KRS2]), partial dead code elimination (cf. [KRS3]), assignment motion (cf. [KRS4]), and strength reduction (cf. [KRS5]). All these techniques, which only require unidirectional bitvector analyses, are now available for parallel programs. In Section 4 this is demonstrated by presenting a code motion algorithm, which evolves from the ***busy code motion*** transformation of [KRS2], and is unique in placing the computations of a parallel program computationally optimally.

Structure of the Paper

The next section will recall the sequential situation, while Section 3 develops the corresponding notions for parallel programs. Subsequently, Section 4 presents an application of our algorithm, and Section 5 contains the conclusions. 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: Program Models

In the sequential setting procedures are usually represented by ***directed flow graphs*** $G = (N, E, \mathbf{s}, \mathbf{e})$ with node set N and edge set E, where the nodes $n \in N$ represent the statements, and the edges $(n, m) \in E$ the ***nondeterministic*** branching structure of the procedure under consideration, while $\mathbf{s}$ and $\mathbf{e}$ denote the unique ***start node*** and ***end node*** of G. Without loss of generality, it is assumed that $\mathbf{s}$ and $\mathbf{e}$ do not have any predecessors and successors, respectively. Figure 1 shows the flow graph of some procedure for illustration.

However, similar to [St], we use here a different, transition system-like representation of a procedure, which we call a ***program model***. Like a flow graph, also a program model is a directed graph $T = (N, E, \mathbf{s}, \mathbf{e})$ with node set N, edge set E, and a unique ***start node*** $\mathbf{s}$ and ***end node*** $\mathbf{e}$ that are assumed to have no predecessors and successors, respectively. In contrast to a flow graph, however, the edges of T represent both the statements and the nondeterministic control flow of the underlying procedure, while the nodes only represent program points. This gives a program model the flavour of a transition system, and therefore, we will use the notions 'nodes' and 'states', and 'edges' and 'transitions' of a program model T synonymously.

Given a flow graph G the corresponding program model T results from the following simple transformation: For every node n of G do:

- introduce a new node n', and an edge e from n to n',
- label e with the assigment node n is labelled with in G, and remove the labelling of node n,

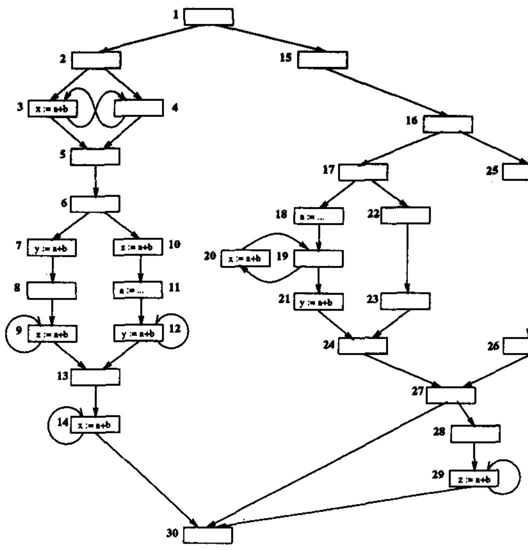

FIGURE 1. The Flow Graph G

- replace every edge starting in n (except for the one inserted in the first step) by a corresponding edge starting in n'.

Figure 2 shows the result of this transformation for the flow graph of Figure 1. It is worth noting that the two states of a program model corresponding to a node n of the underlying flow graph explicitly represent the usual distinction between the *entry point* and the *exit point* of n. This simplifies the formal development of the theory, as the implicit treatment of this distinction, which, unfortunately is usually necessary for the traditional flow graph representation, is obsolete here.

Given a program model T, then $pred_T(n) =_{df} \{ m \mid (m,n) \in E \}$ denotes the set of all immediate predecessors of a state n, and $source(e)$ and $dest(e)$ denote the source and the destination state of a transition e. A *finite path* in T is a sequence $(e_1, \ldots, e_q)$ of transitions such that $dest(e_j) = source(e_{j+1})$ for $j \in \{1, \ldots, q-1\}$; it is a path from m to n, if $source(e_1) = m$ and $dest(e_q) = n$. Moreover, $\mathbf{P}_T[m,n]$ denotes the set of all finite paths from m to n, and ε denotes the empty path containing no transition. Finally, without loss of generality we assume that every state $n \in N$ lies on a path from s to 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 (cf. [He, MJ]). For imperative languages, DFA provides informa-

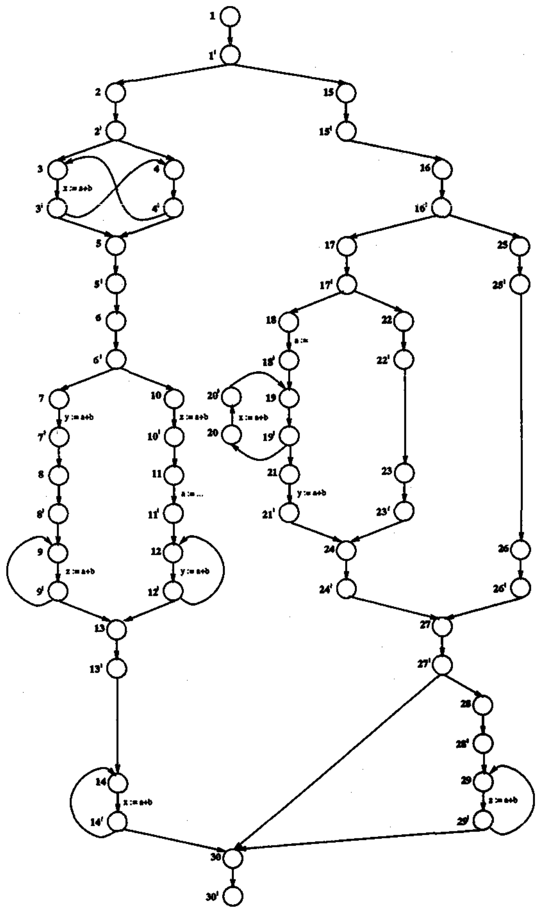

FIGURE 2. The Program Model T of G

tion 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. [CC1, 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. Usually, the abstract semantics is specified by a *local semantic functional*, which gives abstract meaning to every statement in terms of a transformation function on a complete lattice. Thus considering program models, the abstract semantics gives abstract meaning to every transition by means of a functional

$$[\![\ ]\!] : E \rightarrow (\mathcal{C} \rightarrow \mathcal{C})$$

where $(\mathcal{C}, \sqcap, \sqsubseteq, \perp, \top)$ denotes a complete lattice with least element $\perp$ and greatest element $\top$, whose elements express the data flow information of interest.[1]

Unlabelled transitions representing the empty statement **skip** are 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 $p = (e_1, \ldots, e_q) \in \mathbf{P}_G[m, n]$, we define:

$$[\![\, p \,]\!] =_{df} \begin{cases} Id_{\mathcal{C}} & \text{if } p \equiv \varepsilon \\ [\![\, (e_2, \ldots, e_q) \,]\!] \circ [\![\, e_1 \,]\!] & \text{otherwise} \end{cases}$$

The MOP-Solution of a DFA

The solution of the *meet over all paths (MOP)* approach in the sense of Kam and Ullman [KU] defines the intuitively desired solution of a DFA. This approach directly mimics possible program executions in that it "meets" (intersects) all information 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:**

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

The MFP-Solution of a DFA

The point of the *maximal fixed point (MFP)* approach in the sense of Kam and Ullman [KU] is to iteratively approximate the greatest solution of a system of equations which specifies the consistency between conditions expressed in terms of data flow information of $\mathcal{C}$:

Equation System 2.1

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

Denoting the greatest solution of Equation System 2.1 with respect to the start information $c_0 \in \mathcal{C}$ by $\mathbf{info}_{c_0}$, the solution of the *MFP*-approach is defined by:

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

For monotonic functionals,[2] this leads to a suboptimal but algorithmic description (cf. [KU]). The question of optimality of the *MFP*-solution

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

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

was elegantly answered by the Coincidence Theorem of Kildall [Kil, Ki2], and Kam and Ullman [KU], which we reformulate here for program models:

Theorem 2.2 (The (Sequential) Coincidence Theorem)
Given a program model $T = (N, E, \mathbf{s}, \mathbf{e})$, the MFP-solution and the MOP-solution coincide, i.e. $\forall n \in N.\ MOP_{(T,[\![\ ]\!])}(n) = MFP_{(T,[\![\ ]\!])}(n)$, whenever all the semantic functions $[\![e]\!]$, $e \in E$, are distributive.[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 [SP]. 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 data flow information that is assumed to be valid at the start node of the program into the data flow information being valid at n.

Definition 2.3 (The Functional Approach)
The functional $[\![\ ]\!] : N \to (\mathcal{C} \to \mathcal{C})$ is defined as the greatest solution of the equation system given by:

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

The following equivalence result is important [KS]:

Theorem 2.4 $\forall n \in N\ \forall c_0 \in \mathcal{C}.\ MFP_{(T,[\![\ ]\!])}(n)(c_0) = [\![n]\!](c_0)$

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, the functional form can be dealt with without performance penalty.

3 Parallel Programs

As usual, we consider a parallel imperative programming language with an *interleaving semantics*. Formally, this means that we view parallel programs semantically as 'abbreviations' for nondeterministic programs, which result from a product construction between parallel components (cf. [CC2,

[3]A function $f : \mathcal{C} \to \mathcal{C}$ is called ***distributive*** iff $\forall \mathcal{C}' \subseteq \mathcal{C}.\ f(\bigsqcap \mathcal{C}') = \bigsqcap\{f(c) \mid c \in \mathcal{C}'\}$. 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 $\forall \mathcal{C}' \subseteq \mathcal{C}.\ f(\bigsqcap \mathcal{C}') \sqsubseteq \bigsqcap\{f(c) \mid c \in \mathcal{C}'\}$.

CH1, CH2]). In fact, the size of the nondeterministic 'product' program may grow 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: Parallel Program Models

Syntactically, we express parallelism 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 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 in order to allow a dynamical process creation can be integrated along the lines of [CH2, Vo1, Vo2].

Following [SHW] and [GS], the standard representation of a parallel program is a nondeterministic *parallel flow graph* $G^* = (N^*, E^*, \mathbf{s}^*, \mathbf{e}^*)$ with node set N^* and edge set E^* as illustrated in Figure 3. This figure shows the flow graph of Figure 1, where some of the branch instructions have been replaced by parallel statements.[4] The components of a parallel statement are encapsulated by a `ParBegin` and a `ParEnd` node, which are represented by ellipses. For clarity we additionally separate the parallel components by two parallels.

In anology to Section 2, we represent parallel programs as *parallel program models*, which are a straightforward extension of program models to the parallel setting. Except for subgraphs representing **par** statements a parallel program model is a program model in the sense of Section 2, and in fact, all the standard notation can be transferred. Also the transformation from flow graphs to program models is the same, except that `ParBegin` and `ParEnd` nodes are not duplicated. Figure 4 displays the parallel program model of the parallel flow graph of Figure 3.

A **par** statement and each of its components are also considered parallel program models. The graph T_{par} representing a complete **par** statement arises from linking its component graphs by means of a `ParBegin` and a

[4]Of course, this replacement is not assumed to be semantics preserving.

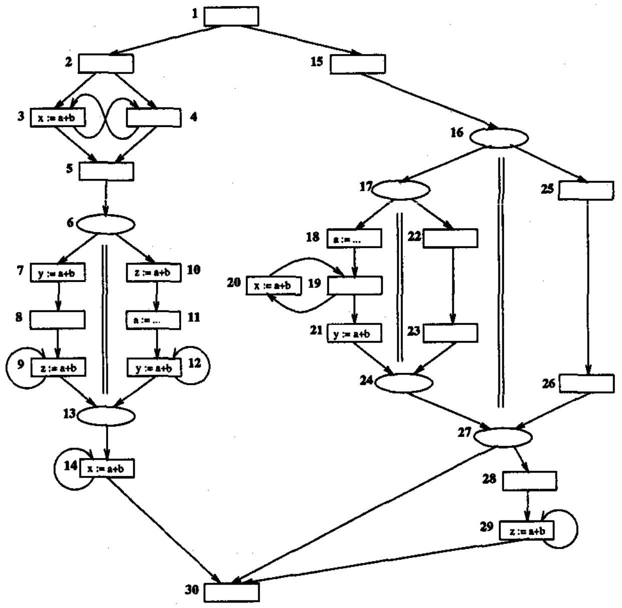

FIGURE 3. The Parallel Flow Graph G^*

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 T_{par}. They form the entry and the exit to program regions whose subgraph components are assumed to be executed in parallel making the synchronization points in the program explicit. As in a parallel flow graph, we represent the states corresponding to a ParBegin node or a ParEnd node of a parallel flow graph by ellipses and additionally separate the corresponding component graphs by two parallels as shown in Figure 4.

Moreover, $\mathcal{T}_\mathcal{P}(T^*)$ and $\mathcal{T}_\mathcal{P}^{max}(T^*)$ denote the set of all subgraphs and of all maximal subgraphs of T^* representing a par statement, i.e.,[5]

$$\mathcal{T}_\mathcal{P}^{max}(T^*) =_{df} \{\, T \in \mathcal{T}_\mathcal{P}(T^*) \mid \forall T' \in \mathcal{T}_\mathcal{P}(T^*).\ T \subseteq T' \Rightarrow T = T' \,\}$$

Additionally, $\mathcal{T}_\mathcal{C}(T')$, $T' \in \mathcal{T}_\mathcal{P}(T^*)$, denotes the set of component program models of T', and $\mathcal{T}_\mathcal{C}(T^*)$ is an abbreviation for $\bigcup \{\, \mathcal{T}_\mathcal{C}(T') \mid T' \in \mathcal{T}_\mathcal{P}(T^*) \,\}$. It is worth noting that every graph $T \in \mathcal{T}_\mathcal{P}(T^*)$ and all of its component program models $T' \in \mathcal{T}_\mathcal{C}(T)$ are single-entry/single-exit regions of T^*. Moreover, for technical reasons (see Section 'Interleaving Predecessors') we assume that the unique transitions ending in the start state or starting in the end state of a graph $T \in \mathcal{T}_\mathcal{C}(T^*)$ are edges of T.

[5]For parallel program models T and T' we define: $T \subseteq T'$ if and only if $N \subseteq N'$ and $E \subseteq E'$.

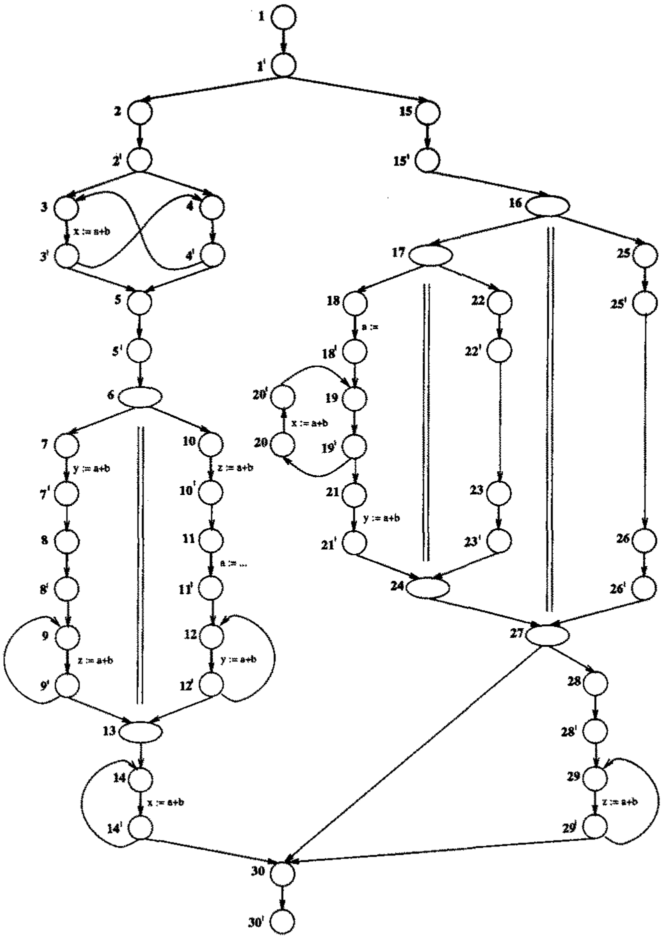

FIGURE 4. The Parallel Program Model T^* of G^*

Additionally, we need the functions *States*, *Trans*, *start*, and *end*, which map a parallel program model to its state set, its transition set, its start state, and its end state, respectively. Moreover, we need the polymorphic functions *ppm* and *cpm*, where *ppm* is defined for the states of graphs of $\mathcal{T}_{\mathcal{P}}(T^*)$ and for the graphs of $\mathcal{T}_{\mathcal{C}}(T^*)$, and *cpm* is defined for the states and the transitions of T^*. *ppm* maps its argument x to the smallest parallel program model of $\mathcal{T}_{\mathcal{P}}(T^*)$ containing x, i.e.,

$$ppm(x) =_{df} \begin{cases} \bigcap\{ T' \in \mathcal{T}_{\mathcal{P}}(T^*) \,|\, x \in States(T') \} & \text{if } x \in States(\mathcal{T}_{\mathcal{P}}(T^*)) \\ \bigcap\{ T' \in \mathcal{T}_{\mathcal{P}}(T^*) \,|\, x \subseteq T' \} & \text{if } x \in \mathcal{T}_{\mathcal{C}}(T^*) \end{cases}$$

Similarly, the polymorphic function *cpm* maps its argument x, which is a state or a transition of T^* to the smallest parallel component model containing x, if it lies in a graph $T \in \mathcal{T}_C(T^*)$, and to T^* itself otherwise, i.e.,[6]

$$cpm(x) =_{df} \begin{cases} \bigcap \{ T' \in \mathcal{T}_C(T^*) \mid x \in T' \} & \text{if } x \in \mathcal{T}_C(T^*) \\ T^* & \text{otherwise} \end{cases}$$

Note that both *ppm* and *cpm* are well-defined because par statements in a program are either unrelated or properly nested.

Additionally, 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 $\mathcal{T}_\mathcal{P}(T^*)$:

$$N_N^* =_{df} \{ start(T) \mid T \in \mathcal{T}_\mathcal{P}(T^*) \} \quad \text{and} \quad N_X^* =_{df} \{ end(T) \mid T \in \mathcal{T}_\mathcal{P}(T^*) \}$$

Finally, given a parallel program model T, we define an associated sequential program model T_{seq}, which results from T by replacing all states belonging to a component parallel model of some graph $T' \in \mathcal{T}_\mathcal{P}^{max}(T)$ together with all transitions starting or ending in such a state by a transition leading from $start(T')$ to $end(T')$. Note that T_{seq} is a sequential program model in the sense of Section 2. This is illustrated in Figure 5, which shows the sequentialized version of the parallel program model encapsulated by the nodes **16** and **27** of Figure 4.

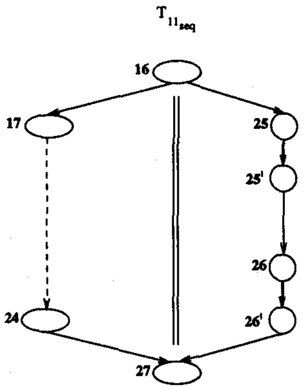

FIGURE 5. A Sequentialized Program Model

Interleaving Predecessors

Given a sequential program model T, the set of transitions that might precede a transition e at run-time is precisely given by the set of *static*

[6] $x \in \mathcal{T}_C(T^*)$ is an abbreviation for $x \in States(\mathcal{T}_C(T^*)) \cup Trans(\mathcal{T}_C(T^*))$.

predecessors, the incoming transitions of *source*(e). For parallel program models, however, the interleaving of parallel components must also be taken into account: here each transition occurring in a component of some **par** statement can dynamically also be preceded by any transition of another component of this **par** statement.

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

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

is defined by

$$ParRel(T) =_{df}$$

$$\mathcal{T}_{\mathcal{C}}(ppm(T))\backslash T \;\cup \begin{cases} \emptyset & \text{if } ppm(T) \in \mathcal{T}_{\mathcal{P}}^{max}(T^*) \\ ParRel(ppm(T)) & \text{otherwise} \end{cases}$$

where $\mathcal{P}$ denotes the power set operator.

Based on this function, the set of interleaving predecessors of a transition $e \in E^*$ is given by the function $ItlvgPred_{T^*} : E^* \to \mathcal{P}(E^*)$ defined by:

$$ItlvgPred_{T^*}(e) =_{df} \begin{cases} Trans(ParRel(cpm(e))) & \text{if } e \in Trans(\mathcal{T}_{\mathcal{P}}(T^*)) \\ \emptyset & \text{otherwise} \end{cases}$$

For illustration consider the transition $\mathbf{e} \equiv (\mathbf{21}, \mathbf{21'})$ of Figure 6. While $\mathbf{e_0}$ is the only transition, which statically precedes this transition, its execution may be interleaved with all transitions of the shadowed components.

Program Paths of Parallel Program Models

As mentioned already, the interleaving semantics reduces parallel programs to (much larger) nondeterministic sequential programs representing all the possible interleavings explicitly (cf. [HU]). Paths in these nondeterministic 'product programs' model the possible executions of a parallel program model. We therefore define that an edge sequence of a parallel program model is a *parallel path* iff it is a path in the corresponding nondeterministic sequential product program, and we denote the set of all parallel paths from m to n by $\mathbf{PP}_{T^*}[m,n]$.[7]

[7]In [KSV1] an alternative and technically much more complicated definition was given for parallel flow graphs.

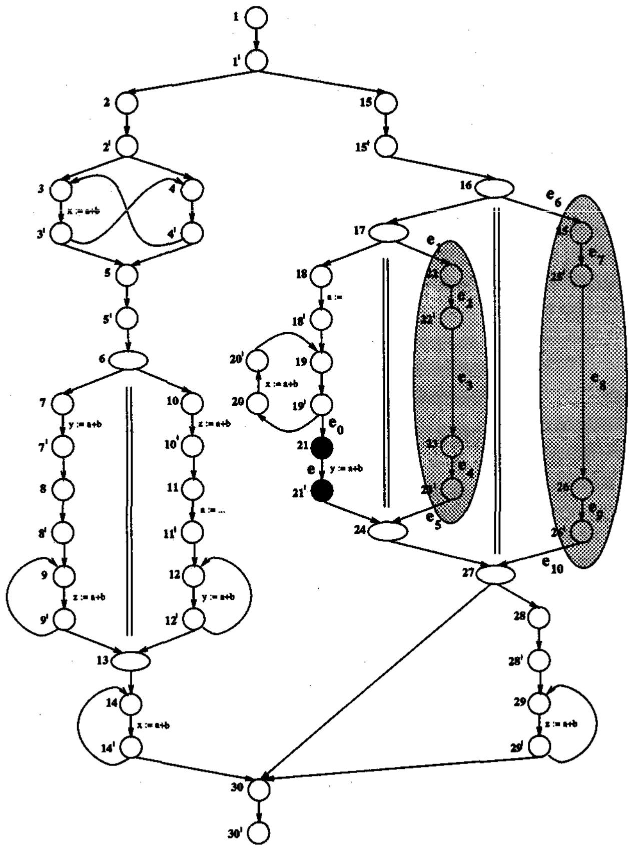
FIGURE 6. Parallel Relatives and Interleaving Predecessors

3.2 Data Flow Analysis of Parallel Programs

As before, a DFA for a parallel program model is completely specified by a local semantic functional $[\![\ ]\!] : E^* \to (\mathcal{C} \to \mathcal{C})$, which can straightforward be extended to cover finite parallel paths as well. Thus, given a state n of a parallel program model T^*, the parallel version of the 'desired' *MOP*-solution is given by:

The *PMOP*-Solution:

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

Note that the corresponding nondeterministic product program would allow us to straightforward adapt the sequential situation with all its results. However, the involved potentially exponential product construction is unacceptable in practice. Fortunately, as we will see in the next 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

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

which specifies the effect of a transition e on a particular component of the bitvector (cf. Section 4 for illustration). Here, $\mathcal{B}$ is the lattice of Boolean truth values $(\{ff, tt\}, \sqcap, \sqsubseteq)$ 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, partial dead code elimination, assignment motion, and strength reduction.

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

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 3.2 (Main-Lemma)
Let $f_i : \mathcal{F}_\mathcal{B} \to \mathcal{F}_\mathcal{B}$, $1 \le i \le q$, $q \in I\!N$, 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 this lemma for our application is that it restricts the way of possible interference within a parallel program model: each possible interference is due to a single transition within a parallel component. Combining this observation with the fact that for $e' \in ItlvgPred_{T^*}(e)$, there exists a parallel path, where e' is directly executed after e, we obtain that the potential of interference, which in general would be given in terms of paths, is fully characterized by the set $ItlvgPred_{T^*}(e)$. 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 $dest(e)$ that does not require the execution of any transition of $ItlvgPred_{T^*}(e)$ implies that the only effect of interference is 'destruction'. This motivates the introduction of the predicate *NotKilled*, which we derive from a predicate *Kills* defined for graphs of $\mathcal{T}_C(T^*)$, which is true for such a graph if it contains a transition e with $[\![e]\!] = Const_{ff}$. Note that this predicate can easily be computed by a statical examination of T^*. Based on *Kills*, we now define the desired:

$$NotKilled(n) =_{df} \begin{cases} \bigwedge\{ \neg Kills(T') \mid T' \in ParRel(cpm(n)) \} & \\ & \text{if } cpm(n) \in \mathcal{T}_C(T^*) \\ tt & \text{otherwise} \end{cases}$$

Intuitively, *NotKilled* indicates that no transition of a parallel relative destroys the property under consideration, i.e. $[\![e']\!] \neq Const_{ff}$ for all $e' \in ItlvgPred_{T^*}(e)$, $e \in Trans(cpm(n))$. Note that only the constant function given by the precomputed value of this predicate is used in Definition 3.5 to model interference, and in fact, Theorem 3.6 guarantees that this modelling is sufficient. Obviously, this predicate is easily and efficiently computable. Algorithm 1.1 computes it as a side result.

Synchronization

Besides taking care of possible interference, we also need to take care of the synchronization required at 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 program models. The underlying 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.[8] The following definition, which is illustrated in Section 4, describes the complete three-step procedure:

[8]Also in [SHW] parallel statements are investigated in an innnermost fashion.

1. Terminate, if T does not contain any parallel statement. Otherwise, select successively all maximal program models T' occurring in a graph of $\mathcal{T}_\mathcal{P}(T)$ that do not contain any parallel statement, and determine the effect $[\![T']\!]$ of this (purely sequential) graph according to the equational system of Definition 2.3.

2. Compute the effect $[\![T'']\!]^*$ of the innermost parallel statements T'' of T by

$$[\![T'']\!]^* = \begin{cases} \mathit{Const}_{\mathit{ff}} & \text{if } \exists\, T' \in \mathcal{T}_\mathcal{C}(T'').\ [\![\mathit{end}(T')]\!] = \mathit{Const}_{\mathit{ff}} \\ \mathit{Id}_\mathcal{B} & \text{if } \forall\, T' \in \mathcal{T}_\mathcal{C}(T'').\ [\![\mathit{end}(T')]\!] = \mathit{Id}_\mathcal{B} \\ \mathit{Const}_{\mathit{tt}} & \text{otherwise} \end{cases}$$

3. Transform T by replacing all innermost parallel statements $T'' = (N'', E'', \mathrm{s}'', \mathrm{e}'')$ by $(\{\mathrm{s}'', \mathrm{e}''\}, \{(\mathrm{s}'', \mathrm{e}'')\}, \mathrm{s}'', \mathrm{e}'')$, define the local semantics of $(\mathrm{s}'', \mathrm{e}'')$ by $[\![T'']\!]^*$, and set the predicate $\mathit{Kills}(\mathrm{s}'')$ to tt, if one of the start nodes of the parallel components of T'' satisfies the predicate Kills, and to ff otherwise. Continue with step 1.

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 *The PMOP-solution of a parallel program model $T \in \mathcal{T}_\mathcal{P}(T^*)$ that only consists of purely sequential parallel components $T_1, \ldots, T_k$ is given by:*

$$\mathit{PMOP}_{(T, [\![\,]\!])}(\mathit{end}(T)) = \begin{cases} \mathit{Const}_{\mathit{ff}} & \text{if } \exists\, 1 \le i \le k.\ [\![\mathit{end}(T_i)]\!] = \mathit{Const}_{\mathit{ff}} \\ \mathit{Id}_\mathcal{B} & \text{if } \forall\, 1 \le i \le k.\ [\![\mathit{end}(T_i)]\!] = \mathit{Id}_\mathcal{B} \\ \mathit{Const}_{\mathit{tt}} & \text{otherwise} \end{cases}$$

Also the proof of this lemma is a consequence of Main Lemma 3.2. As a single transition 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 transition. 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.

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, which is tailored to cover complete paths, i.e. paths going from the start to the end of a parallel statement:

Theorem 3.4 (The Hierarchical Coincidence Theorem)
Let $T \in \mathcal{T}_{\mathcal{P}}(T^*)$ be a parallel program model, and $[\![\ ]\!] : E^* \to \mathcal{F}_{\mathcal{B}}$ a local semantic functional. Then we have:

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

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

Definition 3.5 *The functional $[\![\ ]\!] : N^* \to \mathcal{F}_{\mathcal{B}}$ is defined as the greatest solution of the equation system given by:*[9]

$$[\![\, n\,]\!] = \begin{cases} Id_{\mathcal{B}} & \text{if } n = \mathbf{s}^* \\ [\![\, ppm(n)\,]\!]^* \circ [\![\, start(ppm(n))\,]\!] \sqcap Const_{NotKilled(n)} & \text{if } n \in N_X^* \\ \bigsqcap \{ [\![\, (m,n)\,]\!] \circ [\![\, m\,]\!] \mid m \in pred_{T^*}(n) \} \sqcap Const_{NotKilled(n)} & \text{otherwise} \end{cases}$$

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:

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

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

As in the sequential case the $PMFP_{BV}$-solution is practically relevant, because it can efficiently be computed (see Algorithm 1.1 in Appendix 1). The following theorem now establishes that it coincides with the desired *PMOP*-solution.

Theorem 3.6 (The Parallel Bitvector Coincidence Theorem)
Let $T^* = (N^*, E^*, \mathbf{s}^*, \mathbf{e}^*)$ be a parallel program model, and $[\![\ ]\!] : E^* \to \mathcal{F}_{\mathcal{B}}$ a local semantic functional. Then we have that the PMOP-solution and the $PMFP_{BV}$-solution coincide, i.e.,

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

[9]Note that $[\![\ ]\!]$ is the straightforward extension of the functional defined in Definition 2.3. Thus the overloading of notation is harmless, as no reference to the sequential version is made in this definition.

Intuitively, the (sequential) Coincidence Theorem 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 3.6 is the same, only the correspondence between control flow and program representation is more complicated due to the interleaving and synchronization effects.

4 Application: Code Motion

In this section we demonstrate the practicality of our framework by sketching a *code motion* algorithm, which is unique in placing the computations of a parallel program *computationally optimally*. The power of this algorithm, which evolves as the straightforward extension of its sequential counterpart, the *busy code motion* transformation of [KRS2], is illustrated by means of the example of Figure 3, where our algorithm achieves the optimization result of Figure 7. It eliminates the partially redundant computations of $a + b$ at the nodes **3**, **10**, **12**, **14**, **20**, **21**, **29** by moving them to the nodes **2**, **11** and **18**, but it does not touch the partially redundant computations of $a + b$ at the nodes **7** and **9**, which cannot safely be eliminated.

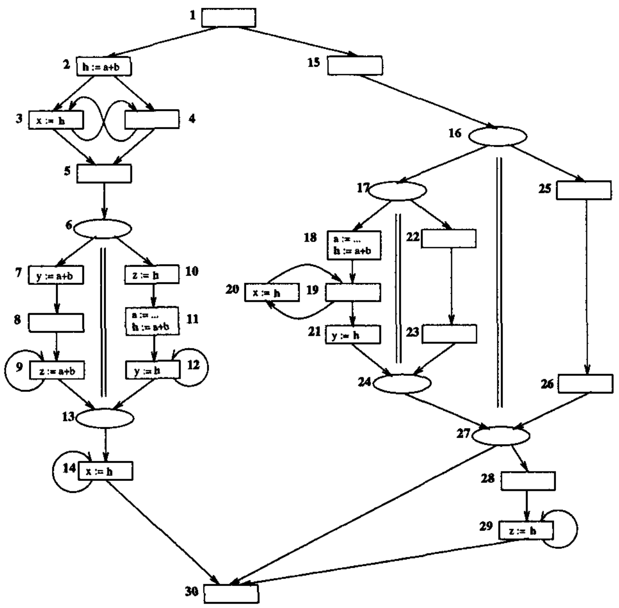

FIGURE 7. The Result of the BCM_{PP}-Transformation

Intuitively, code motion improves the run-time efficiency of a program by avoiding unnecessary recomputations of values at run-time. 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 placing the computations *as early as possible* in a program, while maintaining its semantics, leads to computationally optimal results (cf. [KRS1, KRS2]). This carries over to the parallel setting.

As in the sequential case the as-early-as-possible placement of computations requires the computation of 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.[10] For the ease of presentation we assume here that 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.[11] The DFA-problems for up-safety and down-safety are then specified by the local semantic functionals $[\![\ ]\!]_{us}$ and $[\![\ ]\!]_{ds}$, where *Comp* and *Transp* are two local predicates, which are true for a transition e with respect to a computation t, if t occurs in the right hand side term of the statement of e, and if no operand of t is modified by it, respectively.

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

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

It is worth noting 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 *NotKilled*, which are automatically derived from the definitions of the local semantic functionals.

Moreover, the functionals can directly be fed into the generic Algorithm 1.1 for computing the *PMFP*-solutions of down-safety and up-safety, as illustrated in Figure 8. As in the sequential case, down-safe start states are 'earliest', as well as other down-safe but not not up-safe states that

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

[11]Recursive assignments can also be handled but require a slightly refined treatment.

either possess an 'unsafe' predecessor (see node **2**) or an incoming transition modifying an operand of the computation under consideration (see nodes **11′** and **18′**).

After inserting an initialization statement at each earliest state, all original computations belonging to transitions with a safe source state can be *replaced* by the corresponding temporary, as illustrated in Figure 8. This transformation results in the promised parallel program of Figure 7, which is indeed computationally optimal with respect to $a + b$.

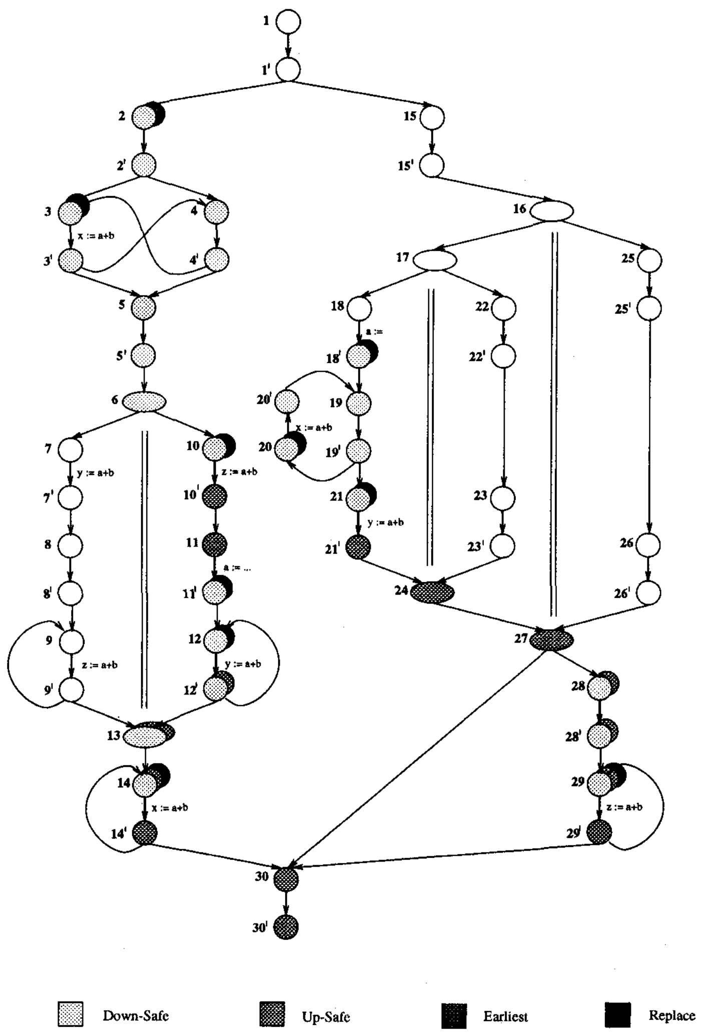

FIGURE 8. Down-Safe, Up-Safe, Earliest, and Replacement Points of $a + b$

5 Conclusions

We have shown how to construct for unidirectional bitvector problems optimal analysis algorithms for parallel programs with shared memory that are as efficient as their purely sequential counterparts, and which can easily be implemented. At the first sight, the existence of such an algorithm is rather surprising, as the interleaving semantics underlying our programming language is an indication for an exponential 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. This algorithm is implemented on the *Fixpoint Analysis Machine* of [SCKKM]. Moreover, the 'lazy' variant (cf. [KRS1, KRS2]) of the code motion transformation of Section 4 is implemented in the ESPRIT project COMPARE #5933 [Vo1, Vo2].

6 References

[CC1] Cousot, P., and Cousot, R. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the 4th International Symposium on Principles of Programming Languages (POPL'77)*, Los Angeles, California, 1977, 238 - 252.

[CC2] Cousot, P., and Cousot, R. Invariance proof methods and analysis techniques for parallel programs. In *Biermann, A. W., Guiho, G., and Kodratoff, Y. (eds.) Automatic Program Construction Techniques*, chapter 12, 243 - 271, Macmillan Publishing Company, 1984.

[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.

[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.

[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.

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

[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.

[DS] 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 3rd 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.

[HU] Hopcroft, J. E., and Ullman, J. E. Introduction to automata theory, languages, and computation. Addison-Wesley, Reading, Massach., 1979.

[Ki1] Kildall, G. A. Global expression optimization during compilation. Ph.D. dissertation, Technical Report No. 72-06-02, University of Washington, Computer Science Group, Seattle, Washington, 1972.

[Ki2] Kildall, G. A. A unified approach to global program optimization. In *Conference Record of the 1st ACM Symposium on Principles of Programming Languages (POPL'73)*, Boston, Massachusetts, 1973, 194 - 206.

[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. 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.

[KRS4] 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.

[KRS5] Knoop, J., Rüthing, O., and Steffen, B. Lazy strength reduction. *Journal of Programming Languages 1*, 1 (1993), 71 - 91.

[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 1st 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. Optimal code motion for parallel programs. Fakultät für Mathematik und Informatik, Universität Passau, Germany, MIP-Bericht 9511 (1995), 30 pages.

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

[LC] Long, D., and Clarke, L. A. Data flow analysis of concurrent systems that use the rendezvous model of synchronization. In *Proceedings of the ACM SIGSOFT'91 Symposium on Testing, Analysis, and Verification (TAV4)*, Victoria, British Columbia, *Software Engineering Notes 16*, (1991), 21- 35.

[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.

[St] Steffen, B. Generating data flow analysis algorithms from modal specifications. *Science of Computer Programming 21*, (1993), 115 - 139.

[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.

1 Computing the $PMFP_{BV}$-Solution

Algorithm 1.1 (Computing the $PMFP_{BV}$-Solution)

Input: *A parallel program model $T^* = (N^*, E^*, \mathbf{s}^*, \mathbf{e}^*)$, a local semantic functional $[\![\ ]\!] : E^* \rightarrow \mathcal{F}_\mathcal{B}$, a function $f_{init} \in \mathcal{F}_\mathcal{B}$ and a Boolean value $b_{init} \in \mathcal{B}$, where f_{init} and b_{init} reflect the assumptions on the context in which the program model under consideration is called. Usually, f_{init} and b_{init} are given by $Id_\mathcal{B}$ and $f\!f$, respectively.*

Output: *An annotation of T^* with functions $[\![T]\!]^* \in \mathcal{F}_\mathcal{B}$, $T \in \mathcal{T}_\mathcal{P}(T^*)$, representing the semantic functions computed in step 2 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 3.5. In fact, after the termination of the algorithm the functional $[\![\ ]\!]$ satisfies:*

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

Remark: *The global variables $[\![T]\!]^*$, $T \in \mathcal{T}_\mathcal{C}(T^*)$, each of which is storing a function of $\mathcal{F}_\mathcal{B}$, are used for storing the global effects of component graphs of graphs $T \in \mathcal{T}_\mathcal{P}(T^*)$ during the hierarchical computation of the $PMFP_{BV}$-solution. The global variables $Kills(start(T))$, $T \in \mathcal{T}_\mathcal{C}(T^*)$, store whether T contains a transition e with $[\![e]\!] = Const_{f\!f}$. These variables are used to compute the value of the predicate NotKilled of Section 3.3. Moreover, every program model $T \in \mathcal{T}_\mathcal{P}(T^*)$ is assumed to have a* rank, *which is recursively defined by:*

$$rank(T) =_{df} \begin{cases} 0 & \text{if } T \in \mathcal{T}_\mathcal{P}^{min}(T^*) \\ max\{\, rank(T') \mid T' \in \mathcal{T}_\mathcal{P}(T^*) \wedge T' \subset T \,\} + 1 & \text{otherwise} \end{cases}$$

where $\mathcal{T}_\mathcal{P}^{min}(T^*) =_{df} \{\, T \in \mathcal{T}_\mathcal{P}(T^*) \mid \forall T' \in \mathcal{T}_\mathcal{P}(T^*).\ T' \subseteq T \Rightarrow T' = T \,\}$ denotes the set of minimal *graphs of $\mathcal{T}_\mathcal{P}(T^*)$. Finally, $succ_T(n) =_{df} \{\, m \mid (n, m) \in E^* \,\}$*

denotes the set of all immediate successors of a state n of a parallel program model T, and MFP denotes the standard procedure for computing the MFP-solution in the sequential case.

```
BEGIN
  ( Synchronization: Computing $[\![ T ]\!]^*$ for all $T \in \mathcal{T_P}(T^*)$ )
  $GLOBEFF(T^*, [\ ])$;

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

where

```
PROCEDURE GLOBEFF ($\underline{T} = (N, E, \mathbf{s}, \mathbf{e})$ : ParallelProgramModel;
                   $\overline{[\ ]} : E \to \mathcal{F_B}$ : LocalSemanticFunctional);
VAR $i$ : integer;
BEGIN
  FOR $i := 0$ TO $rank(T)$ DO
    FORALL $T' \in \{T'' \mid T'' \in \mathcal{T_P}(T) \wedge rank(T'') = i\}$ DO
      FORALL $T'' \equiv (N'', E'', \mathbf{s}'', \mathbf{e}'') \in \{T'''_{seq} \mid T''' \in \mathcal{T_C}(T')\}$ DO
        LET $\forall e \in E''.\ [\, e \,]'' = \begin{cases} [\![ ppm(dest(e)) ]\!]^* & \text{if } e \in N_N^* \times N_X^* \\ \overline{[\, e \,]} & \text{otherwise} \end{cases}$
          BEGIN
            $Kills(start(T'')) := (\ |\{ n \in N'' \mid Kills(n) \}| \geq 1 ) \vee$
                                  $(\ |\{ e \in E'' \mid [\, e \,]'' = Const_{ff} \}| \geq 1 )$;
            $MFP(T'', [\ ]'', Id_{\mathcal{B}})$;
            $[\![ T'' ]\!]^* := [\![ end(T'') ]\!]$
          END
      OD;
      $[\![ T' ]\!]^* := \begin{cases} Const_{ff} & \text{if } \exists T'' \in \mathcal{T_C}(T').\ [\![ T''_{seq} ]\!]^* = Const_{ff} \\ Id_{\mathcal{B}} & \text{if } \forall T'' \in \mathcal{T_C}(T').\ [\![ T''_{seq} ]\!]^* = Id_{\mathcal{B}} \\ Const_{tt} & \text{otherwise} \end{cases}$
    OD
  OD
END.
```

```
PROCEDURE PMFP_BV ($\underline{T} = (N, E, \mathbf{s}, \mathbf{e})$ : ParallelProgramModel;
                   $\overline{[\ ]} : E \to \mathcal{F_B}$ : LocalSemanticFunctional;
                   $f_{start} : \mathcal{F_B}$; IsKilled : $\mathcal{B}$);
VAR $f : \mathcal{F_B}$;
BEGIN
  IF IsKilled THEN FORALL $n \in N$ DO $[\![ n ]\!] := Const_{ff}$ OD
  ELSE
    ( Initialization of the annotation arrays $[\![\ ]\!]$ and the variable workset )
    FORALL $n \in States(T_{seq}) \backslash \{\mathbf{s}\}$ DO
      $[\![ n ]\!] := \begin{cases} Const_{ff} & \text{if } \exists e \in E.\ dest(e) = n \wedge \overline{[\, e \,]} = Const_{ff} \\ Const_{tt} & \text{otherwise} \end{cases}$
    OD;
```

```
        $[\![ s ]\!] := f_{start}$;
        $worklist := \{ n \in States(T_{seq}) \mid n \in N_N^* \cup \{s\} \vee [\![ n ]\!] = Const_{ff} \}$;

        ( 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
                  FORALL $m \in succ_T(n)$ DO
                    $f := \overline{[\![ (n,m) ]\!]} \circ [\![ n ]\!]$;
                    IF $[\![ m ]\!] \sqsupset f$
                      THEN
                        $[\![ m ]\!] := f$;
                        $workset := workset \cup \{ m \}$
                    FI
                  OD
                ELSE
                  FORALL $T' \in \mathcal{T}_C(ppm(n))$ DO
                    $PMFP_{BV}(T', \overline{[\![\ ]\!]}, [\![ n ]\!], \sum_{T'' \in \mathcal{T}_C(ppm(n)) \backslash \{T'\}} Kills(start(T'')))$
                  OD;
                  $f := [\![ ppm(n) ]\!]^* \circ [\![ n ]\!]$;
                  IF $[\![ end(ppm(n)) ]\!] \sqsupset f$
                    THEN
                      $[\![ end(ppm(n)) ]\!] := f$;
                      $workset := workset \cup \{ end(ppm(n)) \}$
                  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 1.1, and $[\![n]\!]$, $n \in N^*$, the greatest solution of the equation system of Definition 3.5, then we have:

Theorem 1.2 $\forall n \in N^*.\ [\![n]\!]_{alg} = [\![n]\!]$