Parallel Data-Flow Analysis of Explicitly Parallel Programs

Jens Knoop

Universität Dortmund, D-44221 Dortmund, Germany
phone: ++49-231-755-5803, fax: ++49-231-755-5802
knoop@ls5.cs.uni-dortmund.de
http://sunshine.cs.uni-dortmund.de/~knoop

Abstract. In terms of program verification *data-flow analysis* (*DFA*) is commonly understood as the computation of the *strongest* postcondition for every program point with respect to a precondition which is assured to be valid at the entry of the program. Here, we consider DFA under the dual *weakest* precondition view of program verification. Based on this view we develop an approach for *demand-driven* DFA of explicitly parallel programs, which we exemplify for the large and practically most important class of bitvector problems. This approach can directly be used for the construction of online debugging tools. Moreover, it is tailored for parallelization. For bitvector problems, this allows us to compute the information provided by conventional, strongest postcondition-centered DFA at the costs of answering a data-flow query, which are usually much smaller. In practice, this can lead to a remarkable speed-up of analysing and optimizing explicitly parallel programs.

1 Motivation

Intuitively, the essence of *data-flow analysis* (*DFA*) is to determine run-time properties of a program without actually executing it, i.e., at compile-time. In its conventional peculiarity, DFA means to compute for every program point the most precise, i.e., the "largest" data-flow fact which can be guaranteed under the assumption that a specific data-flow fact is valid at the entry of the program. In terms of program verification (cf. [1]), this corresponds to a strongest postcondition approach: Compute for every program point the *strongest* postcondition with respect to a precondition assured to be valid at the entry of the program.

In this article we consider DFA under the dual, weakest precondition view of program verification, i.e., under the view of asking for the *weakest* precondition, which must be valid at the entry of the program in order to guarantee the validity of a specific "requested" postcondition at a certain program point of interest. In terms of DFA this means to compute the "smallest" data-flow fact which has to be valid at the entry of the program in order to guarantee the validity of the "requested" data-flow fact at the program point of interest.

This dual view of DFA, which has its paragon in program verification, is actually the conceptual and theoretical backbone of approaches for *demand-driven*

P. Amestoy et al. (Eds.): Euro-Par'99, LNCS 1685, pp. 391–400, 1999.

(*DD*) DFA. Originally, such approaches have been proposed for the interprocedural analysis of sequential programs, in particular aiming at the construction of online debugging tools (cf. [3, 7, 16]), but were also successfully applied to other fields (cf. [4, 17]).

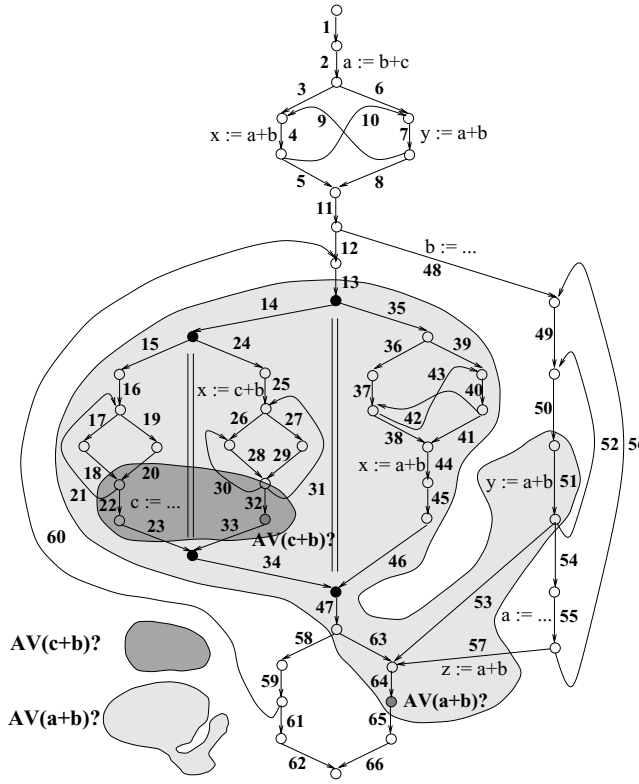

Fig. 1. Demand-drivenly, only small portions of the program need to be investigated.

In this article we apply this approach to the analysis of explicitly parallel programs with shared memory and interleaving semantics. The central benefit of this approach, which leads to the efficiency gain in practice, is illustrated in Figure 1 using the *availability of terms*, a classical DFA-problem (cf. [6]), for illustration. Intuitively, a term is *available* at a program point, if it has been computed on every program path reaching this point without a succeeding modification of any of its operands. Using the conventional approach to DFA, the program of Figure 1 must completely be investigated in order to detect the availability and the inavailability of the terms $a+b$ and $c+b$ at the destination nodes of the edges **64** and **32**, respectively. In contrast, our DD-approach has only to consider small portions of the complete program in order to answer these two

data-flow queries. Though generally DD-approaches have the same worst-case time complexity as their conventional counterparts, in practice they are usually much more efficient (cf. [5]). Additionally, unlike its conventional counterpart, the DD-approach we are going to propose here can immediately be parallelized: Different data-flow queries can concurrently be processed. Obviously, this is tailored for a setting offering parallel processes on shared memory because the processes can easily share the answers to data-flow queries computed by other processes, which leads to a further speed-up of the analysis. Additionally, for unidirectional bitvector analyses (cf. [6]), the class of DFAs the framework of [15] is designed for, this approach can directly be used to mimic conventional DFA, for which it is characteristic to decide the DFA-property of interest for every program point. In our approach, this is achieved by answering the corresponding DFA-queries for all program points in parallel. In fact, this yields the information delivered by conventional DFA at the costs of answering a data-flow query, which are usually much smaller. This is important as subsequent program optimizations usually rely on the "full" information.

2 Background: DD-DFA in the Sequential Setting

In DFA programs are commonly represented by flow graphs $G = (N, E, \mathbf{s}, \mathbf{e})$ with node set N, edge set E, and a unique start node $\mathbf{s}$ and end node $\mathbf{e}$, which are assumed to have no incoming and outgoing edges, respectively. Here, we assume that edges represent the statements and the nondeterministic control flow of the underlying program, while nodes represent program points. We denote the sets of immediate predecessors and successors of a node n by $pred(n)$ and $succ(n)$, and the set of all finite paths from a node m to a node n by $\mathbf{P}[m, n]$.

In order to decide some (aspect of a) run-time property by a DFA, one usually replaces the "full" semantics of the program by a simpler, more abstract version (cf. [2]), which is tailored for the problem under consideration. Usually, the abstract semantics is given by a *semantic functional* $[\![\ ]\!] : E \to (\mathcal{C} \to \mathcal{C})$, which gives abstract meaning to the elementary statements of a program in terms of a transformation on a complete lattice $\mathcal{C}$ with least element $\bot$ and greatest element $\top$. The elements of $\mathcal{C}$ represent the data-flow facts of interest. Without loss of generality we assume that $\top \in \mathcal{C}$ represents "unsatisfiable (inconsistent)" data-flow information, and is invariant under all semantic functions.

Conventional DFA: Strongest Postcondition View. Conventionally, DFA aims at computing for every program point the strongest postcondition with respect to a precondition $c_{\mathbf{s}} \in \mathcal{C} \backslash \{\top\}$ assured to be valid at $\mathbf{s}$. Formally, this is expressed by the *meet-over-all-paths* (*MOP*) approach, which induces the *MOP*-solution (cf. [9]). Note that $[\![p]\!]$ denotes the straightforward extension of $[\![\ ]\!]$ to paths, i.e., $[\![p]\!]$, $p = \langle e_1, \ldots, e_q \rangle$, equals the identity $Id_{\mathcal{C}}$ on $\mathcal{C}$, if $q < 1$, and $[\![\langle e_2, \ldots, e_q \rangle]\!] \circ [\![e_1]\!]$, otherwise.

The *MOP*-Solution: The Strongest Postcondition

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

Demand-Driven DFA: Weakest Precondition View. Fundamental for the dual, weakest precondition view of DFA is the notion of the *reversed semantic functional* $[\![\ ]\!]_R : E \to (\mathcal{C} \to \mathcal{C})$. It is induced by $[\![\ ]\!]$, and defined as follows:

$$\forall\, e \in E\ \forall\, c \in \mathcal{C}.\ [\![\, e\,]\!]_R(c) =_{df} \bigsqcap \{\, c' \mid [\![\, e\,]\!](c') \sqsupseteq c\,\}.$$

Intuitively, the reversed function $[\![\, e\,]\!]_R$, $e \in E$, maps a data-flow fact $c \in \mathcal{C}$, which plays here the role of a postcondition, to the least data-flow fact, which is sufficient to guarantee the validity of c after executing e. We have (cf. [8]):

Proposition 1. *For all edges $e \in E$ we have: (1) $[\![\, e\,]\!]_R$ is well-defined and monotonic. (2) If $[\![\, e\,]\!]$ is distributive, then $[\![\, e\,]\!]_R$ is additive.*

Like their counterparts also the reversed functions can easily be extended to capture finite paths. $[\![\, p\,]\!]_R$, $p = \langle e_1, \ldots, e_q\rangle$, equals the identity $Id_{\mathcal{C}}$ on $\mathcal{C}$, if $q < 1$, and $[\![\, \langle e_1, \ldots, e_{q-1}\rangle\,]\!]_R \circ [\![\, e_q\,]\!]_R$, otherwise. Note that this means a "backward" traversal of p as expected when dealing with weakest preconditions.

In order to complete the presentation of the weakest precondition view of DFA we introduce the following simple extension of G, which simplifies the formal development. If $q \in N$ is the program point of interest, i.e., the *query node*, we assume that G is extended by a copy $\mathbf{q}$ of q having the same predecessors as q, but no successors. Obviously, the *MOP*-solutions for q in G and for $\mathbf{q}$ in the extended program coincide; a fact, which is important for the correctness of considering the extended program. Denoting the data-flow query by c_q, where the index implicitly fixes the query node, the straightforward reversal of the *MOP*-approach leads from the strongest postcondition view to its dual weakest precondition view. Besides reversing the analysis direction, the duality also requires to consider the dual lattice operation of *joining* data-flow facts. The dual analogue of the meet-over-all-paths approach is thus the *reversed-join-over-all-paths* (*R-JOP*) approach. Intuitively, the solution of the *R-JOP*-approach at a program point is the join of all data-flow facts required by some program continuation reaching $\mathbf{q}$ in order to assure c_q at $\mathbf{q}$ (and q).

The *R-JOP*-Solution: The Weakest Precondition

$$\forall\, c_q \in \mathcal{C}\ \forall\, n \in N.\ \textit{R-JOP}_{([\![\ ]\!], c_q)}(n) =_{df} \bigsqcup \{\, [\![\, p\,]\!]_R(c_q) \mid p \in \mathbf{P}[n, \mathbf{q}]\,\}$$

The Connecting Link. The following theorem establishes the link between the strongest postcondition and the weakest preconditon view of DFA. If the semantic functions $[\![\, e\,]\!]$, $e \in E$, are all distributive (as it will be the case in Section 4!), the *MOP*-solution and the *R-JOP*-solution are equally informative.

Theorem 1 (Link Theorem).
Let all functions $[\![\, e\,]\!]$, $e \in E$, be distributive. Then for every pair of a node $q \in N$ and its copy $\mathbf{q}$, and for every pair of data-flow facts $c_{\mathbf{s}}, c_q \in \mathcal{C}$ we have:

$$\textit{MOP}_{([\![\ ]\!], c_{\mathbf{s}})}(q) \sqsupseteq c_q \iff \textit{R-JOP}_{([\![\ ]\!], c_q)}(\mathbf{s}) \sqsubseteq c_{\mathbf{s}}$$

In particular, we have: (1) $\textit{R-JOP}_{([\![\ ]\!], \textit{MOP}_{([\![\ ]\!], c_{\mathbf{s}})}(q))}(\mathbf{s}) \sqsubseteq c_{\mathbf{s}}$ and
(2) $\textit{MOP}_{([\![\ ]\!], \textit{R-JOP}_{([\![\ ]\!], c_q)}(\mathbf{s}))}(q) = \textit{MOP}_{([\![\ ]\!], \textit{R-JOP}_{([\![\ ]\!], c_q)}(\mathbf{s}))}(\mathbf{q}) \sqsupseteq c_q$

Computing Strongest Postconditions and Weakest Preconditions. The *MOP*-solution and the *R-JOP*-solution are both specificational in nature. They generally do not induce an effective computation procedure. In practice, this is taken care of by complementing each approach with a fixed-point approach. These are based on the following two equation systems, which impose consistency constraints on an annotation of the program with data-flow facts, where ***ass*** and ***req-ass*** remind to "assertion" and "required assertion," respectively.

Equation System 1 (*MaxFP*-Approach: The *MOP*-Counterpart).

$$\mathbf{ass}(n) = \begin{cases} c_{\mathbf{s}} & \text{if } n = \mathbf{s} \\ \bigsqcap \{ [\![(m,n)]\!](\mathbf{ass}(m)) \mid m \in pred(n) \} & \text{otherwise} \end{cases}$$

Equation System 2 (*R-MinFP*-Approach: The *R-JOP*-Counterpart).

$$\mathbf{req\text{-}ass}(n) = \begin{cases} c_q & \text{if } n = \mathbf{q} \\ \bigsqcup \{ [\![(n,m)]\!]_R(\mathbf{req\text{-}ass}(m)) \mid m \in succ(n) \} & \text{otherwise} \end{cases}$$

Denoting the greatest solution of Equation System 1 with respect to the start assertion $c_{\mathbf{s}} \in \mathcal{C}$ by $\mathtt{ass}_{c_{\mathbf{s}}}$, and the least solution of Equation System 2 with respect to the data-flow query c_q at $\mathbf{q}$ by $\mathtt{req\text{-}ass}_{c_q}$, the solutions of the *MaxFP*-approach and the *R-MinFP*-approach are defined as follows:

The *MaxFP*-Solution: $\forall\, c_{\mathbf{s}} \in \mathcal{C}\ \forall\, n \in N.\ MaxFP_{([\![\ ]\!],c_{\mathbf{s}})}(n) =_{df} \mathtt{ass}_{c_{\mathbf{s}}}(n)$

The *R-MinFP*-Solution: $\forall c_q \in \mathcal{C}\ \forall n \in N.\ R\text{-}MinFP_{([\![\ ]\!],c_q)}(n) =_{df} \mathtt{req\text{-}ass}_{c_q}(n)$

Both fixed-point solutions can effectively be computed, when the semantic functions are monotonic and the lattice is free of infinite chains. Moreover, they coincide with their pathwise defined counterparts, when the semantic functions are distributive, which implies additivity of their reversed counterparts. We have:

Theorem 2 (Coincidence Theorem).
The MaxFP-solution and the MOP-solution coincide, i.e., $\forall\, c_{\mathbf{s}} \in \mathcal{C}\ \forall\, n \in N.\ MaxFP_{([\![\ ]\!],c_{\mathbf{s}})}(n) = MOP_{([\![\ ]\!],c_{\mathbf{s}})}(n)$*, if the semantic functions* $[\![\, e \,]\!]$*,* $e \in E$*, are all distributive.*

Theorem 3 (Reversed Coincidence Theorem).
The R-MinFP-solution and the R-JOP-solution coincide, i.e., $\forall\, c_q \in \mathcal{C}\ \forall\, n \in N.\ R\text{-}MinFP_{([\![\ ]\!],c_q)}(n) = R\text{-}JOP_{([\![\ ]\!],c_q)}(n)$*, if the semantic functions* $[\![\, e \,]\!]$*,* $e \in E$*, are all distributive.*

Hence, for distributive semantic functions the *R-MinFP*-solution and the *R-JOP*-solution coincide. Together with the Link Theorem 1, this means that the *R-MinFP*-solution is equally informative as the *MOP*-solution. Moreover, in the absence of infinite chains it can effectively be computed by an iterative process. This is central for demand-driven DFA, however, its idea goes a step further: It stops the computation *as early as possible*, i.e., as soon as the validity or invalidity of the data-flow query under consideration is detected. Reduced to

a slogan, demand-driven (DD) DFA can thus be considered the "sum of computing weakest preconditions and early termination." The following algorithm realizes this concept of DD-DFA for sequential programs. After its termination the variable *answerToQuery* stores the answer to the data-flow query under consideration, i.e., whether c_q holds at node q. In Section 4 we will show how to extend this approach to explicitly parallel programs.

(Prologue: Initialization of the annotation array *reqAss* and the variable *workset*)
FORALL $n \in N \backslash \{\mathbf{q}\}$ DO $reqAss[n] := \bot$ OD;
$reqAss[\mathbf{q}] := c_q$; $workset := \{\, \mathbf{q} \,\}$;

(Main process: Iterative fixed-point computation)
WHILE $workset \neq \emptyset$ DO
  LET $m \in workset$
    BEGIN
      $workset := workset \backslash \{\, m \,\}$;
      (Update the predecessor-environment of node m)
      FORALL $n \in pred(m)$ DO
        $join := [\![\, (n, m) \,]\!]_R(reqAss[m]) \sqcup reqAss[n]$;
        IF $join = \top \ \vee \ (n = \mathbf{s} \wedge join \not\sqsubseteq c_{\mathbf{s}})$
          THEN (`Query failed!`)
            $reqAss[\mathbf{s}] := join$; (`Indicate Failure`)
            $goto\ 99$ (`Stop Computation`)
        ELSIF $reqAss[n] \sqsubset join$
          THEN $reqAss[n] := join$;
               $workset := workset \cup \{\, n \,\}$ FI OD END OD;

(Epilogue)
99 : $answerToQuery := (reqAss[\mathbf{s}] \sqsubseteq c_{\mathbf{s}})$.

3 The Parallel Setting

In this section we sketch our parallel setup, which has been presented in detail in [15]. We consider parallel imperative programs with shared memory and interleaving semantics. Parallelism is syntactically expressed by means of a `par` statement. As usual, we assume that there are neither jumps leading into a component of a parallel statement from outside nor vice versa. As shown in Figure 1, we represent a parallel program by an edge-labelled *parallel flow-graph* G with node set N and edge set E having a unique start node $\mathbf{s}$ and end node $\mathbf{e}$ without incoming and outgoing edges, respectively.

Additionally, $IntPred(n) \subseteq E$, $n \in N$, denotes the set of so-called *interleaving predecessors* of a node n, i.e., the set of edges whose execution can be interleaved with the execution of the statements of n's incoming edge(s) (cf. [15]). A *parallel program path* is an edge sequence corresponding to an interleaving sequence of G. By $\mathbf{PP}[m, n]$, $m, n \in N$, we denote the set of all finite parallel program paths from m to n.

4 Demand-Driven DFA in the Parallel Setting

As in [15] we concentrate on *unidirectional bitvector* (*UBV*) problems. The most prominent representatives of this class are the availability and very busyness of terms, reaching definitions, and live variables (cf. [6]).

UBV-problems are characterized by the simplicity of the semantic functional $[\![\ ]\!] : E \to (\mathcal{B}_X \to \mathcal{B}_X)$ defining the abstract program semantics. It specifies the effect of an edge e on a particular component of the bitvector, where $\mathcal{B}_X$ is the lattice $(\{ff, tt, failure\}, \sqcap, \sqsubseteq)$ of Boolean truth values with $ff \sqsubseteq tt \sqsubseteq failure$ and the logical "and" as meet operation $\sqcap$ (or its dual counterpart). The element *failure* plays here the role of the top element representing unsatisfiable data-flow information. Hence, for each edge e, the function $[\![e]\!]$ is an element of the set of functions $\{Cst^X_{tt}, Cst^X_{ff}, Id_{\mathcal{B}_X}\}$, where $Id_{\mathcal{B}_X}$ denotes the identity on $\mathcal{B}_X$, and Cst^X_{tt} and Cst^X_{ff} the extensions of the constant functions Cst_{tt} and Cst_{ff} from $\mathcal{B}$ to $\mathcal{B}_X$ leaving the argument *failure* invariant.

The reversed counterparts $R\text{-}Cst^X_{tt}$, $R\text{-}Cst^X_{ff}$, and $R\text{-}Id_{\mathcal{B}_X}$ of these functions, and hence the reversed semantic functions $[\![e]\!]_R$, $e \in E$, are defined as follows:

$$\forall\, b \in \mathcal{B}_X.\ R\text{-}Cst^X_{tt}(b) =_{df} \begin{cases} ff & \text{if } b \in \mathcal{B} \\ failure & \text{otherwise (i.e., if } b = failure) \end{cases}$$

$$\forall\, b \in \mathcal{B}_X.\ R\text{-}Cst^X_{ff}(b) =_{df} \begin{cases} ff & \text{if } b = ff \\ failure & \text{otherwise} \end{cases}$$

$$R\text{-}Id_{\mathcal{B}_X} =_{df} Id_{\mathcal{B}_X}$$

Note that all the functions Cst^X_{tt}, Cst^X_{ff}, and $Id_{\mathcal{B}_X}$ are distributive. Hence, the reversed functions are additive (cf. Proposition 1(2)).

The *MOP*-approach and the *R-JOP*-approach of the sequential setting can now straightforwardly be transferred to the parallel setting. Instead of sequential program paths we are now considering parallel program paths representing interleaving sequences of the parallel program. We define:

The $PMOP_{UBV}$-Solution: Strongest Postcondition

$$\forall\, b_0 \in \mathcal{B}_X\ \forall\, n \in N.\ PMOP_{([\![\ ]\!], b_0)}(n)(b_0) = \bigsqcap \{\, [\![p]\!](b_0) \mid p \in \mathbf{PP}[\mathbf{s}, n] \,\}$$

The $R\text{-}PJOP_{UBV}$-Solution: Weakest Precondition

$$\forall\, b_0 \in \mathcal{B}_X\ \forall\, n \in N.\ R\text{-}JOP_{([\![\ ]\!], b_0)}(n) =_{df} \bigsqcup \{\, [\![p]\!]_R(b_0) \mid p \in \mathbf{PP}[n, \mathbf{q}] \,\}$$

Note that actually *tt* is the only interesting, i.e., non-trivial data-flow query for UBV-problems. In fact, *ff* is always a valid assertion, while *failure* is never.

Like their sequential counterparts, the $PMOP_{UBV}$-solution and the $R\text{-}PJOP_{UBV}$-solution are specificational in nature. They do not induce an effective computation procedure. In [15], however, it has been shown, how to compute for UBV-problems the parallel version of the *MOP*-solution by means of a fixed-point process in a fashion similar to the sequential setting. Basically, this is a

two-step approach. First, the effect of parallel statements is computed by means of an hierarchical process considering innermost parallel statements first and propagating information computed about them to their enclosing parallel statements. Subsequently, these results are used for computing the parallel version of the *MFP*-solution. This can be done almost as in the sequential case since in this phase parallel statements can be considered like elementary ones. The complete process resembles the one of *interprocedural* DFA of sequential programs, where first the effect functions of procedures are computed, which subsequently are used for computing the interprocedural version of the *MFP*-solution (cf. [10]).

Important for the success of the complete approach is the restriction to UBV-problems. For this problem class the effects of *interference* and *synchronization* between parallel components can be computed without consideration of any interleaving (cf. [15]). The point is that for UBV-problems the effect of each interleaving sequence is determined by a single statement (cf. [15]). In fact, this is also the key for the successful transfer of the DD-approach from the sequential to the parallel setting. In addition, the two steps of the conventional approach are fused together in the DD-approach. Since in sequential program parts the algorithm works as its sequential counterpart of Section 2, we here consider only the treatment of parallel program parts in more detail focusing on how to capture *interference* and *synchronization*. The complete DD-algorithm is given in [13].

Interference. In a parallel program part, the DD-algorithm checks first the existence of a "bad guy." This is an interleaving predecessor destroying the property under consideration, i.e., an edge e with $[\![e]\!] = Cst^X_{ff}$. If a bad guy exists, the data-flow query can definitely be answered with ff, and the analysis stops. This check is realized by means of the function *Interference*. We remark that every edge is investigated at most once for being a bad guy.

```
FUNCTION Interference ( m ∈ N ) : B;
BEGIN
    IF IntPred(m) ∩ badGuys ≠ ∅
       THEN (Failure of the data-flow query because of a bad guy!)
          return(tt)
       ELSE
          WHILE edgesToBeChecked ∩ IntPred(m) ≠ ∅ DO
             LET e ∈ edgesToBeChecked ∩ IntPred(m)
                BEGIN
                   edgesToBeChecked := edgesToBeChecked \ { e };
                   IF [[ e ]] = Cst_ff
                      THEN
                         badGuys := badGuys ∪ { e }; return(tt) FI END OD;
          return(ff) FI
END;
```

Synchronization. In the DD-approach synchronization of the parallel components of a parallel statement takes place when a data-flow query reaches a start node of a parallel component. This actually implies the absence of bad

guys, which has been checked earlier. Synchronization reduces thus to checking whether some of the parallel siblings guarantee the data-flow query (i.e., there is a sibling whose start node carries the annotation *ff*), otherwise, the data-flow query has to be propagated across the parallel statement. This is realized by the procedure *Synchronization*, where pfg, $\mathcal{G}_C$, and *start* denote functions, which yield the smallest parallel statement containing a node m, the set of all component graphs of a parallel statement, and the start node of a graph, respectively.

PROCEDURE *Synchronization* ($m \in N$);
BEGIN
  IF $\forall\, n \in start(\mathcal{G}_C(pfg(m)))$. $reqAss[n] = tt$
    THEN IF $reqAss[start(pfg(m))] \sqsubset tt$
      THEN $reqAss[start(pfg(m))] := tt$;
        $workset := workset \cup \{\, start(pfg(m)) \,\}$ FI FI
END;

The following point is worth to be noted. Unless the data-flow query under consideration fails, in the sequential setting the workset must always completely be processed. In the parallel setting, however, even in the opposite case parts of the workset can sometimes be skipped because of synchronization. If a component of a parallel statement is detected to guarantee the data-flow query under consideration (which implies the absence of destructive interference), nodes of its immediate parallel siblings need not to be investigated as synchronization guarantees that the effect of the component carries over to the effect of the enclosing parallel statement. This is discussed in more detail in [13].

5 Typical Applications

The application spectrum of our approach to DD-DFA of explicitly parallel programs is the same as the one of any DD-approach to program analysis. As pointed out in [5] this includes: (1) *Interactive tools* in order to handle user queries, as e.g., online debugging tools. (2) *Selective optimizers* focusing e.g. on specific program parts like loops having selective data-flow requirements. (3) *Incremental tools* selectively updating data-flow information after incremental program modifications. Thus, our approach is quite appropriate for a broad variety of tools supporting the design, analysis, and optimization of explicitly parallel programs.

6 Conclusions

In this article we complemented the conventional, strongest postcondition-centered approach to DFA of explicitly parallel programs of [15] with a demand-driven, weakest precondition-centered approach. Most importantly, this approach is tailored for parallelization, and allows us to mimic conventional DFA at the costs of answering a data-flow query, which are usually much smaller. This can directly be used to improve the performance of bitvector-based optimizations of explicitly parallel programs like *code motion* [14] and *partial dead-code elimination* [11]. However, our approach is not limited to this setting. In a similar

fashion it can be extended e.g. to the *constant-propagation* optimization proposed in [12].

References

[1] K. R. Apt and E.-R. Olderog. *Verification of Sequential and Concurrent Programs.* 2nd edition, Springer-V., 1997.

[2] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conf. Rec. 4th Symp. Principles of Prog. Lang.* (*POPL'77*), pages 238 – 252. ACM, NY, 1977.

[3] E. Duesterwald, R. Gupta, and M. L. Soffa. Demand-driven computation of interprocedural data flow. In *Conf. Rec. 22nd Symp. Principles of Prog. Lang.* (*POPL'95*), pages 37 – 48. ACM, NY, 1995.

[4] E. Duesterwald, R. Gupta, and M. L. Soffa. A demand-driven analyzer for data flow testing at the integration level. In *Proc. IEEE Int. Conf. on Software Engineering* (*CoSE'96*), pages 575 – 586, 1996.

[5] E. Duesterwald, R. Gupta, and M. L. Soffa. A practical framework for demand-driven interprocedural data flow analysis. *ACM Trans. Prog. Lang. Syst.*, 19(6):992 – 1030, 1997.

[6] M. S. Hecht. *Flow Analysis of Computer Programs.* Elsevier, North-Holland, 1977.

[7] S. Horwitz, T. Reps, and M. Sagiv. Demand interprocedural dataflow analysis. In *Proc. 3rd ACM SIGSOFT Symp. Foundations of Software Eng.* (*FSE'95*), pages 104 – 115, 1995.

[8] J. Hughes and J. Launchbury. Reversing abstract interpretations. *Science of Computer Programming*, 22:307 – 326, 1994.

[9] J. B. Kam and J. D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7:305 – 317, 1977.

[10] J. Knoop. *Optimal Interprocedural Program Optimization: A new Framework and its Application.* PhD thesis, Univ. of Kiel, Germany, 1993. LNCS Tutorial 1428, Springer-V., 1998.

[11] J. Knoop. Eliminating partially dead code in explicitly parallel programs. *TCS*, 196(1-2):365 – 393, 1998. (Special issue devoted to *Euro-Par'96*).

[12] J. Knoop. Parallel constant propagation. In *Proc. 4th Europ. Conf. on Parallel Processing* (*Euro-Par'98*), LNCS 1470, pages 445 – 455. Springer-V., 1998.

[13] J. Knoop. Parallel data-flow analysis of explicitly parallel programs. Technical Report 707/1999, Fachbereich Informatik, Universität Dortmund, Germany, 1999.

[14] J. Knoop and B. Steffen. Code motion for explicitly parallel programs. In Proc. 7th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP'99), Atlanta, Georgia, pages 13 - 24, 1999.

[15] J. Knoop, B. Steffen, and J. Vollmer. Parallelism for free: Efficient and optimal bitvector analyses for parallel programs. *ACM Trans. Prog. Lang. Syst.*, 18(3):268 – 299, 1996.

[16] T. Reps. Solving demand versions of interprocedural analysis problems. In *Proc. 5th Int. Conf. on Compiler Construction* (*CC'94*), LNCS 786, pages 389 – 403. Springer-V., 1994.

[17] X. Yuan, R. Gupta, and R. Melham. Demand-driven data flow analysis for communication optimization. *Parallel Processing Letters*, 7(4):359 – 370, 1997.