A Code Placement Framework and its Application to Communication Generation

Reinhard v. Hanxleden
Ken Kennedy

CRPC-TR93337-S
October, 1993

Center for Research on Parallel Computation
Rice University
P.O. Box 1892
Houston, TX 77251-1892

A Code Placement Framework and its Application to Communication Generation[*][†]

Reinhard von Hanxleden
reinhard@rice.edu

Ken Kennedy
ken@rice.edu

(713) 285-5188
Center for Research on Parallel Computation
Department of Computer Science, Rice University
P.O. Box 1892, Houston, TX 77251

Abstract

We present a dataflow framework that extends classical partial redundancy elimination techniques and uses a general producer-consumer concept. Consumers express a demand for certain computations to be performed, either before or after consumption takes place, depending on the kind of problem to be solved. Producers satisfy this demand, unless it is already satisfied as a side effect by some other operations. Production can be viewed not only as an atomic operation, but it can also be separated into a beginning and an end. Beyond the safety requirements of classical code motion techniques, the option of splitting production implies a need for *Balancedness*, where a production, once started, is guaranteed to be finished, without an intervening restart.

We describe the general framework and its application to the problem of generating communication statements for distributed memory machines. In the communication generation problem, we assume that references to distributed data result in global reads, which can be split into a send from the owner of the data and a receive at the processor referencing the data. We also assume that a definition of distributed data may result in a global write, which can be split into a send from the processor writing the data and a receive at the processor owning them. This presents four different problems, all of which can be solved with the same framework. We implemented the framework as part of a Fortran D compiler prototype, where it is used to solve communication generation problems including the ones described in this paper.

1 Introduction

An important step in compiling data parallel languages, like Fortran D [HKT92a] or High Performance Fortran [Hig93], onto distributed memory machines is the generation of communication statements that allow each processor to reference and define data that they do not own by default. Since generating an individual message for each not-owned datum would be prohibitively expensive on most architectures, optimizations like message vectorization, latency hiding, and avoiding redundant communication are crucial for achieving acceptable performance [HKT92b]. Dependence analysis can guide such optimizations, for example by guaranteeing the safety of hoisting communication out of a loop nest.

However, dependence analysis alone is not powerful enough to take advantage of all optimization opportunities, since it only compares pairs of occurrences (*i.e.*, references or definitions) and does not take into account how control flow links them together. Gross and Steenkiste combine dependence analysis with data flow analysis and regular sections to analyze, for example, reaching definitions [GS90]. It turns out that efficient communication generation is closely related to partial redundancy elimination techniques, which were originally developed for transformations like loop invariant code motion, common subexpression elimination, or strength reduction. The original dataflow framework for performing partial redundancy elimination was developed by Morel and Renvoise [MR79] and has since then experienced various refinements [JD82, DS88, Dha88a, Dha91, DRZ92, KRS92].

[*]Available via anonymous ftp from softlib.rice.edu as pub/CRPC-TRs/reports/CRPC-TR93337-S.

[†]This work is supported by an IBM fellowship.

Dhamdhere uses partial redundancy elimination techniques for placing register loads and stores [Dha88b]. He uses an iterative approach, with separate frameworks for loads and stores. Carr and Kennedy combine partial redundancy elimination with dependence analysis to perform scalar replacement [CK92]. Granston and Veidenbaum combine dependence analysis and partial redundancy elimination to detect redundant global memory accesses in parallelized and vectorized codes [GV91]. Their technique tries to eliminate these operations where possible, also across loop nests and in the presence of conditionals, and they eliminate reads of not-owned variables if these variables have already been read or written locally. However, they assume that the program is already annotated with read/write operations, so they do not try to hoist memory accesses to less frequently executed regions. They also treat each global memory access as a monolithic operation that is not, for example, divided into a send and receive; therefore, they do not provide opportunities for prefetching or latency hiding. Duesterwald *et al.* incorporate iteration distance vectors into an array reference data flow framework which is then applied to memory optimizations and controlled loop unrolling [DGS93]. Amarasinghe and Lam optimize communication generation using Last Write Trees [AL93]. They assume regular array access patterns and distributions. Gupta and Schonberg use Available Section Descriptors, computed by interval based data flow analysis, to determine the availability of data on a virtual processor grid [GS93]. They apply (regular) mapping functions to map this information to individual processors and list redundant communication elimination and communication generation as possible applications. An iterative data flow framework for generating communication statements in the presence of indirection arrays has been developed by von Hanxleden *et al.* [HKK$^+$92]. It uses a loop flow graph designed to ease the hoisting of communication out of innermost loops. However, it handles nested loops and jumps out of loops only to a limited degree, and communication is treated monolithically.

This paper presents a data flow framework, called GIVE-N-TAKE, that is designed for a broad class of code generation placement/problems, including the classical domains of partial redundancy elimination techniques as well as communication generation. Roughly speaking, GIVE-N-TAKE takes as input lists of items that are needed, killed, or "provided for free" at each control flow graph node, and produces as output for each node a list of items that have to be provided in order to satisfy the specified needs. As described in Section 2.1, there are some fixed correctness and optimality criteria that GIVE-N-TAKE is subject to; for example, any generated code should be executed as infrequently as possible. However, the solutions computed by GIVE-N-TAKE vary depending on which kind of problem it is applied to. In a BEFORE problem, items have to be produced before they are needed (like reading in not-owned data), whereas in an AFTER problem, they have to be produced afterwards (like writing out not-owned data). Orthogonally we can classify a problem as EAGER when it asks for production as early as possible (like sending a message), or as LAZY when it wants production as late as possible (like receiving a message); this definition assumes a BEFORE problem, for an AFTER problem "early" and "late" have to be interchanged. Partial redundancy elimination, for example, can be classified as a LAZY, BEFORE problem.

We have implemented GIVE-N-TAKE in the PARASCOPE environment at the Center for Research on Parallel Computation at Rice University as part of a FORTRAN D compiler prototype, where it is used to generate communication statements for distributed memory architectures. We are applying the framework to a variety of communication generation tasks, namely global READ's, global WRITE's, and WRITE's combined with several possible reduction types (addition, multiplication, etc.). Furthermore, separate sends and receives can be separated for each type of communication. The information derived by GIVE-N-TAKE allows the compiler to reduce both the number and the volume of messages and to hide communication latencies with other computation. Having a single framework for generating all necessary types of communication statements simplified the compiler and accelerated both high and low level debugging.

The rest of this paper is organized as follows. Section 2 begins with giving some further intuition for the GIVE-N-TAKE framework, followed by a brief review of interval analysis. It then describes the actual equations and argues informally for their correctness and effectiveness. Section 3 describes the application of GIVE-N-TAKE to the task of generating communication for a distributed memory architecture and gives a sample program with the communications generated for it (see Figure 10). This application will also be used in other sections as an illustrating example. Section 4 lists some further extensions of the framework, some of which are already implemented, and concludes with a brief summary. The appendix contains formal correctness proofs for the equations stated in Section 2.3 and for the algorithm given in Section 2.4.

2 The Give-N-Take Framework

The basic idea behind the GIVE-N-TAKE framework is to view the given code generation problem as a producer-consumer process. In addition to being produced and consumed, data may also be destroyed before consumption.

Furthermore, whatever has been produced can be consumed arbitrarily often, until it gets destroyed.

For example, a processor running a distributed memory machine application typically locally references both owned data, that by default reside on the processor, and not-owned data, that reside on other processors. Local references to not-owned data induce a need for reading these data from other processors. The problem of generating these READ's can be interpreted as a BEFORE problem as follows:

- Each reference to not-owned data *consumes* these data.
- Each READ, where a processor p sends data that it owns to another processor q that receives and references these data, *produces* the data sent.
- Each non-local definition (*i.e.*, a definition at another processor) of not-owned data *destroys* these data.

If we want to split each READ into a READ_{Send} (the send issued at the owner) and a READ_{Recv} (the receive issued by the referencing processor), then we can compute both the EAGER and the LAZYsolution of the framework. Since this is a BEFORE problem and we want to send as early as possible, the READ_{Send}'s will be given by the EAGER solution, and the READ_{Recv}'s will be the LAZY Solution.

In addition, if we do not use a strict owner computes rule, we might have local definitions of not-owned data. We assume that these data have to be written back to their owners before they can used by other processors. (A feasible, but especially in the presence of indirect references difficult alternative would be the direct exchange between a not-owner that writes data and another not-owner that reads them.) These not-owned definitions can now be viewed as consumers, just as not-owned references, and we have to insert producers that in this case communicate data to their owners (instead of from their owners). Since we want to write data after they have been defined, this is an AFTER problem. Note that in this scenario, the previous problem of analyzing communication for not-owned references can be modified to take advantage of not-owned definitions if they are later locally referenced; *i.e.*, not-owned definitions can also be viewed as statements that produce not-owned references as a side effect ("for free"), potentially saving unnecessary communication to and from the owner. Again, we can split each WRITE into a WRITE_{Send} (given by the LAZY solution, since WRITE is a BEFORE problem) and a WRITE_{Recv} (the EAGER solution).

2.1 Correctness and optimality

Given a program with some pattern of consumption and destruction, our framework has to determine a set of producers that meet certain correctness requirements and optimality criteria. The requirements that GIVE-N-TAKE has to meet in order to be correct are the following (with their specific implications for the communication generation).

(C1) *Balancedness*: If we compute both the EAGER and the LAZY solution for a given problem, then these solutions have to match each other. (For each executed READ send, a matching READ receive will be executed, and vice versa; similarly for WRITE sends and receives; see Figure 1.)

(C2) *Safety*: Everything produced will be consumed; see Section 2.2.1 for a discussion of safety in the presence of zero-trip loop constructs. (No unnecessary READ's or WRITE's; see Figure 2, we assume that $X2$ destroys $X1$. In our specific case this is more an optimization than a correctness issue.)

(C3) *Sufficiency*: For each consumer at node n in the program, there must be a producer on each incoming path reaching n, without any destroyer in between. (All reads of not-owned data must be locally satisfiable due to proceeding READ's or local definitions, without intervening non-local definitions, and all definitions of not-owned data must be brought back to their owners by WRITE's before being referenced non-locally or read; see Figure 3.)

The optimization criteria, subject to the correctness constraints stated above, are:

(O1) Nothing produced already (and not destroyed yet) will be produced again. (Nothing will recommunicated, unless it has been non-locally redefined; see Figure 4.)

(O2) There are as few producers as possible. (Communicate as little as possible; see Figure 5.)

(O3) Things are produced as early as possible for EAGER, BEFORE and LAZY, AFTER problems. (Send as early as possible; see Figure 6.)

(O3′) Things are produced as late as possible for LAZY, BEFORE and EAGER, AFTER problems. (Receive as late possible; see Figure 7.)

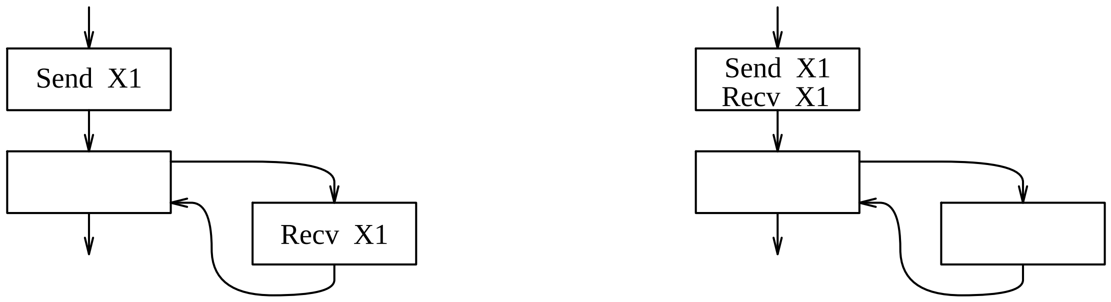

Figure 1: Unbalanced production (left) and possible solution (right).

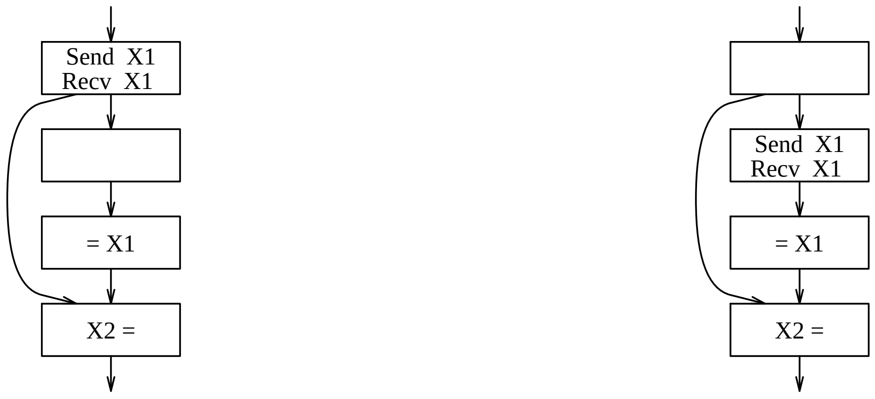

Figure 2: Unsafe production (left) and possible solution (right).

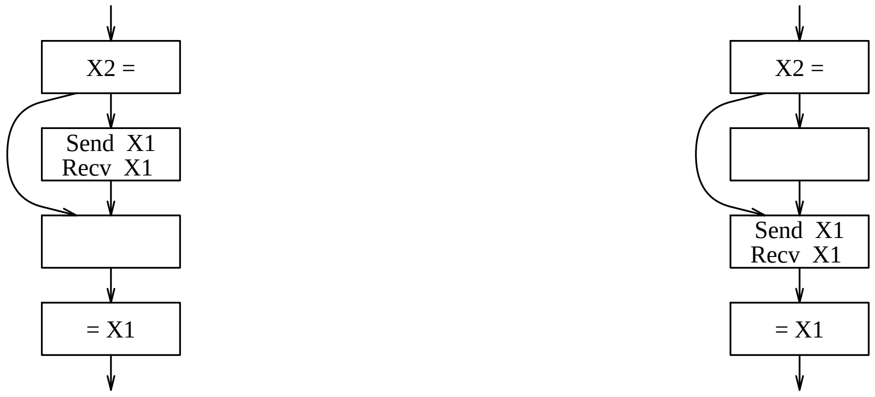

Figure 3: Insufficient production (left) and possible solution (right).

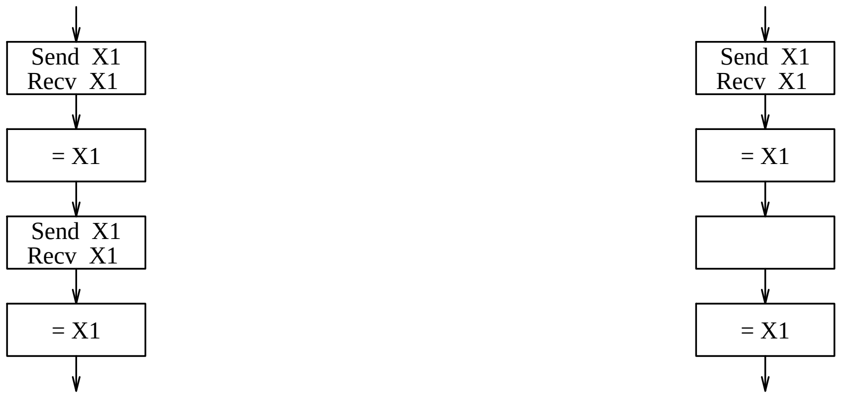

Figure 4: Redundant production (left) and possible solution (right).

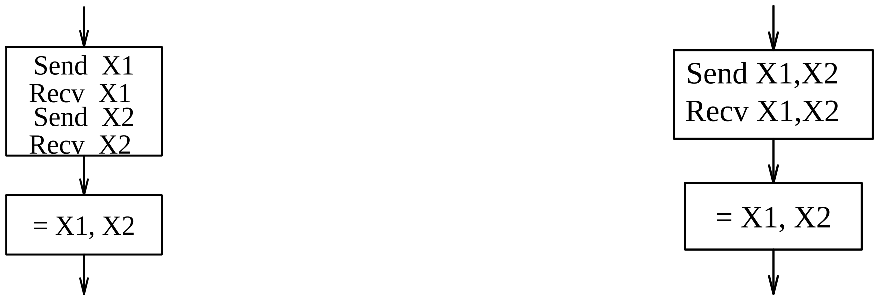

Figure 5: Too many producers (left) and possible solution (right).

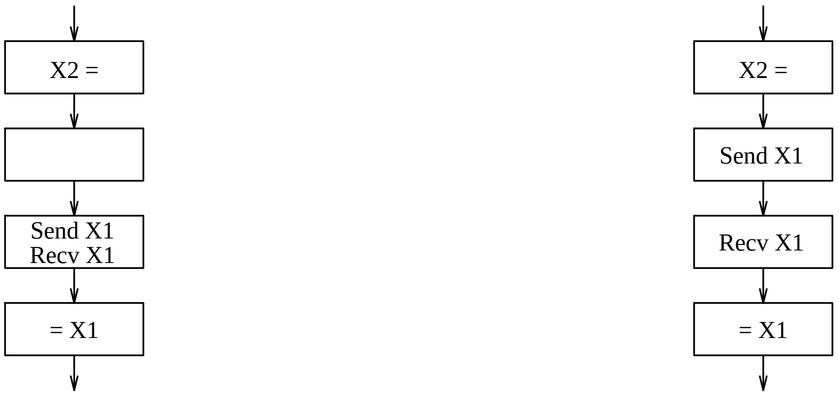

Figure 6: Too late production (left) and possible solution (right).

2.2 The data flow domain

Let $G = (N, E)$ be a control flow graph with nodes N and edges E. Each $n \in N$ has a set of *data flow variables* associated with it. Each variable represents a set of elements from the data flow universe, for example by a bit vector. The GIVE-N-TAKE framework consists of a set of data flow equations, stated in Section 2.3, that define the variables for each n. This definition often depends on the values of variables at nodes that are connected to n through edges in E.

A general data flow analysis algorithm that considers loop nesting hierarchies is interval analysis. It can be used for forward problems (like available expressions) [All70, Coc70] and backward problems (like live variables) [Ken71], and it has also been used for code motion [DP93]. We are using a variant of interval analysis that is based on Tarjan intervals [Tar74]. Like Allen-Cocke intervals, a Tarjan interval $T(h)$ is a set of control flow nodes that corresponds to a loop in the program text, entered through a unique header node h, where $h \notin T(h)$. However, Tarjan intervals include only nodes that are part of this loop (*i.e.*, together with their headers they form nested,

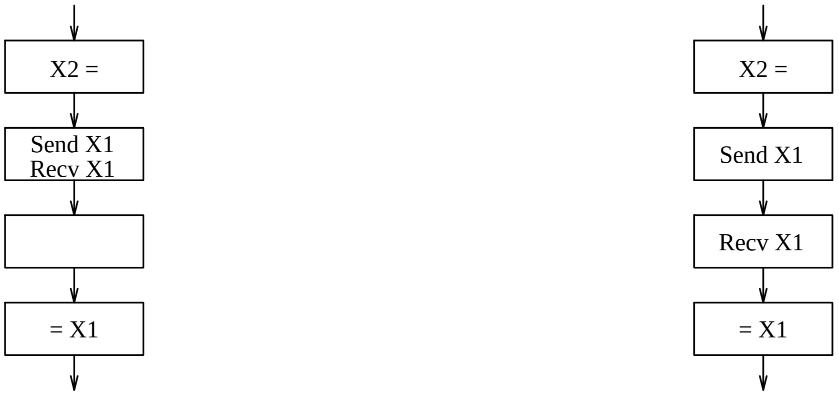

Figure 7: Too early production (left) and possible solution (right).

strongly connected regions), whereas Allen-Cocke intervals include in addition all nodes whose predecessors are all in $T(h)$, *i.e.*, they might include an acyclic structure dangling of the loop. Note that a node nested in multiple loops is a member of the Tarjan interval of the header of each enclosing loop. Unlike in classical interval analysis, we do not explicitly construct a sequence of graphs in which intervals are recursively collapsed into single nodes, but instead we take the interval structure into account when traversing the control flow graph in one of the orderings described in Section 2.2.3. $\textsc{Root} \in N$ is the unique root of G. For $n \in N$, $\textsc{Level}(n)$ is the loop nesting level of n, counted from the outside in; $\textsc{Level}(\textsc{Root}) = 0$.

Figure 10 shows an example code, corresponding control flow graph is in Figure 11. The deepest loop nesting level is three. The h-loop, for example, corresponds to the interval formed by nodes 3 ... 20, with header 2 (again, the header itself is not part of the interval). Note that this interval includes the intervals headed by nodes 4, 10, 13, and 18. Node 5, for example, is included in the three intervals headed by nodes 4, 2, and 0.

2.2.1 Zero-trip loop constructs

One difficulty with flow analysis has traditionally been the treatment of zero-trip loop constructs, like for example a Fortran DO loop. We are interested in hoisting computation out of such loops as well, but this may introduce statements on paths where they have not existed before, which is generally considered unsafe. Several techniques exist to circumvent this difficulty, like for example adding an extra guard and a preheader node to each loop [Sor89], explicitly introducing zero-trip paths [DK83], or collapsing innermost loops [HKK$^+$92]. These strategies, however, result in some loss of information, and they do not fully apply to nested loops. Therefore, the Give-N-Take framework generally treats a loop as if it will be executed at least once. In case this approach is not valid as such for a particular application of the framework, there are several relatively simple refinements to guarantee safety:

- The compiler can try to prove that a loop l will be executed at least once.
- Hoisting a statement S out of a loop l can be prohibited by adding S to $\mathsf{STEAL}(l)$ (see Section 2.3.2).
- S may implicitly be void in case l does not execute; for example, if l has n iterations and S is a statement communicating $x(1{:}n)$, then $n \leq 0$ results in an empty statement.
- We can explicitly guard S by a test whether l will be executed.
- S might be statement that results in some extra, but harmless computation, like an unnecessary communication statement. In this case, we might be willing to pay that extra cost if it is amortized as soon as l is executed at least once.
- In case we rely on a statement S to be executed within l (for example, to bring in some not-owned data that are needed outside of l as well), we can add a test after the loop that explicitly executes S in case l is empty. (These data are indicated in the framework by $\mathsf{GIVE}(l) \setminus \mathsf{GIVE}_{init}(l)$).

2.2.2 The control flow graph

We classify each edge $e = (m, n) \in E$ as one of the following.

Entry Edge: An edge from an interval header to a node within the interval; $n \in T(m)$.

Back Edge: An edge from a node in an interval to the header of the interval; $m \in T(n)$.

Exit edge: An edge from a node in an interval to a node outside of the interval that is not the header node; $\exists l \neq n : m \in T(l), n \notin T(l)$. This corresponds to a jump out of a loop.

Forward Edge: An edge that is none of the above; $\forall l : m \in T(l) \Longleftrightarrow n \in T(l)$.

In Figure 11, all non-forward edges are labeled as either entry, back, or exit edge. Note that forward edges are the only edges not crossing nesting level boundaries.

We extend G to have the following properties:

- G is *reducible*; *i.e.*, each loop has a unique header node. This can be achieved, for example, by node splitting [CM69].

- For each non-empty interval $T(h)$, there exists a unique $g \in T(h)$ such that $(g, h) \in E$; *i.e.*, there is only one back edge out of $T(h)$. This can be achieved by adding a *post body node* to $T(h)$. Note that exit edges are not excluded, since these do not go to the header.
- There are no *critical edges*, which connect a node with multiple outgoing edges to a node with multiple incoming edges. (Intuitively, a critical edge might indicate a location in the program where we cannot place production without affecting paths that are not supposed to be affected by the production.) This can be achieved, for example, by inserting *synthetic nodes* [KRS92]. Code generated for synthetic nodes would reside in newly created basic blocks, like for example a new ELSE branch or a landing pad for a jump out of a loop.

It can easily be seen that the graph shown in Figure 11 fulfills these properties. The empty nodes are the synthetic nodes inserted to break critical edges. For example, each interval header has by definition multiple successors (one of them being its first child) and multiple predecessors (one of them being its last child). Therefore, each edge directly connecting two interval headers needs to be split with a synthetic node.

2.2.3 Traversal orders and neighbor relations

The order in which the nodes of the interval graph are visited depends on the given problem type (BEFORE/AFTER, EAGER/LAZY) and on the pass of the GIVE-N-TAKE framework that is currently solved. E induces two partial orderings on N:

Horizontally: Given a forward/exit edge (m, n), a FORWARD order visits m before n, and a BACKWARD order visits m after n.

Vertically: Given $m, n \in N$ such that $m \in T(n)$, an UPWARD order visits m before n, whereas a DOWNWARD order visits n after m.

Since these partial orderings do not conflict with each other, they can be combined into PREORDER (FORWARD and DOWNWARD), POSTORDER (FORWARD and UPWARD), REVERSEPREORDER (BACKWARD and UPWARD), and REVERSEPOSTORDER (BACKWARD and DOWNWARD). For example, the nodes in Figure 11 are numbered in PREORDER.

A data flow variable for some $n \in N$ might be defined in terms of variables of other nodes that are in some relation to n with respect to G. Therefore, we not only have to walk G in a certain order, but we also have to access for each $n \in N$ a subset of $N \setminus \{n\}$ that has a certain relationship with n.

CHILDREN(n): Assuming n heads an interval, the members of this interval that are one level deeper than n; $\text{CHILDREN}(n) = \{c \mid c \in T(n), \text{LEVEL}(c) = \text{LEVEL}(n) + 1\}$. (In Figure 11, it is $\text{CHILDREN}(2) = \{3 \dots 20\}$)

FIRSTCHILD(n): Assuming n heads an interval, the first child of n; $(n, \text{FIRSTCHILD}(n))$ is an entry edge in E. (*E.g.*, $\text{FIRSTCHILD}(2) = \{3\}$.) $\text{START} = \text{FIRSTCHILD}(\text{ROOT})$ is the beginning of the program.

LASTCHILD(n): Assuming n is a header of an interval, the last child of n; $(\text{LASTCHILD}(n), n)$ is a back edge in E. (*E.g.*, $\text{LASTCHILD}(2) = \{20\}$.) $\text{STOP} = \text{LASTCHILD}(\text{ROOT})$ is the end of the program.

HEADER(n): The header of the interval that directly encloses n; $n \in \text{CHILDREN}(\text{HEADER}(n))$. (*E.g.*, $\text{HEADER}(2) = 0$.) We also introduce $J(n) \equiv T(\text{HEADER}(n))$ as a shorthand for the interval directly enclosing n.

SUCCS(n): The sink nodes of forward or exit edges originating from n; $s \in \text{SUCCS}(n) \Longleftrightarrow (n, s) \in E, \forall h \in N : s \in T(h) \Longrightarrow n \in T(h)$. (*E.g.*, $\text{SUCCS}(2) = \{21\}, \text{SUCCS}(8) = \{9, 25\}$.) We refer to the transitive closure of SUCCS(n) as the *descendants* of n. Note that SUCCS(n) does not include all successors, where by "successors" we generally refer to the set of all sinks of sources of edges originating in n (3 as well is a successor of 2).

PREDS(n): The source nodes of forward or exit edges reaching n; $p \in \text{PREDS}(n) \Longleftrightarrow n \in \text{SUCCS}(p)$. (*E.g.*, $\text{PREDS}(25) = \{8\}$.) We refer to the transitive closure of PREDS(n) as the *ancestors* of n.

SUCCS$^+(n)$: The sink nodes of forward or exit edges originating from n, plus the sink nodes, excluding $T(n)$, of any exit edge originating from a node in $T(n)$; $s \in \text{SUCCS}^+(n) \Longleftrightarrow \exists m \in T(n) \cup \{n\} : (m, s) \in E, \forall h \in N : s \in T(h) \Longrightarrow n \in T(h)$. (*E.g.*, $\text{SUCCS}^+(2) = \{21, 25\}, \text{SUCCS}^+(8) = \{9, 25\}$.)

$\textsc{Preds}^+(n)$**:** The source nodes of forward or exit edges reaching n, plus the headers of intervals that enclose the source but not the sink of these edges; $p \in \textsc{Preds}^+(n) \Longleftrightarrow n \in \textsc{Succs}^+(p)$. (*E.g.*, $\textsc{Preds}^+(25) = \{2, 8\}$.)

$\textsc{Succs}^-(n)$**:** The sink nodes of forward edges originating from n; $s \in \textsc{Succs}^-(n) \Longleftrightarrow (n, s) \in E, s \in J(n)$. (*E.g.*, $\textsc{Succs}^-(2) = \{21\}, \textsc{Succs}^-(8) = \{9\}$.)

Note that the lack of critical edges has several implications for some of the sets defined above:

- Let $e = (n, s)$ be an exit edge. Then there exists an $h \in N$ with $n \in T(h), s \notin T(h)$. Since $T(h) \cup h$ is by definition strongly connected, n must have successors within $T(h) \cup h$. Since s as well is a successor of n, n has multiple outgoing edges. However, G does not have critical edges, therefore s has only one predecessor; *i.e.*, $\textsc{Preds}(s) = \{n\}$. In other words, the sink of an exit edge never has any predecessors besides the source of the exit edge.
- Let $e = (n, h)$ be a back edge. h then is an interval header, which by definition has multiple predecessors. Since h is a successor of n, n may not have any other successors (otherwise e would be critical). However, by definition of $\textsc{Succs}$ it is $h \notin \textsc{Succs}(n)$. It follows $\textsc{Succs}(n) = \emptyset$ for each source n of a back edge.

Even though the equations and their correctness and effectiveness are the same for both BEFORE and AFTER problems, we will for simplicity assume in the following that we are solving a BEFORE problem unless noted otherwise. To get the equivalent formulations for an AFTER problem, all occurrences of FORWARD, FIRSTCHILD, PREDS, PREORDER, and POSTORDER have to be interchanged with BACKWARD, LASTCHILD, SUCCS, REVERSEPOSTORDER, and REVERSEPREORDER, respectively.

2.3 Give-N-Take equations

In the following, let $n \in N$, let $\perp$ denote the empty set, and let $\top$ be the whole data flow universe. If an equation asks for certain neighbors (like $\textsc{Preds}(n)$) and there are no such neighbors (like for a loop entry node), then an empty set results. For example, if $\textsc{Succs}^+(n)$ is empty, then Equation 4 will set $\mathsf{TAKEN}_{out}(n)$ to $\perp$. Subscripts *in*, *out* denote variables for the entry and the exit of a node, respectively. Subscript *loc* indicates information collected only from nodes within the same interval (nodes in $J(n)$), and *init* identifies variables that are supplied as input to GIVE-N-TAKE. Figure 8 contains the equations for the data flow variables, which will be introduced in the following sections (with their specific meaning for our communication generation example).

2.3.1 Initial variables

The following variables get initialized depending on the problem to solve, where $\perp$ is the default value. Sections 3.2 and 3.3 describe the initializations specific to generating READ's and WRITE's for distributed memory accesses, respectively.

$\mathsf{STEAL}_{init}(n)$**:** All elements whose production would be voided at n. (In our communication problem, this includes a portion p – see Section 3.1 – if either the contents of this portion get modified at n, or if the p itself gets changed, for example if p is an indirect array reference and n modifies the indirection array.)

$\mathsf{GIVE}_{init}(n)$**:** All elements that "come for free," *i.e.*, who are already produced at n. (If we do not use the owner computes rule, then this includes local definitions of not-owned data, since a later reference to these data does not need to communicate them in any more.)

$\mathsf{TAKE}_{init}(n)$**:** The set of consumers at n. (The set of not-owned array portions.)

2.3.2 Propagating consumption

The following variables, together with the variables defined in Section 2.3.3, analyze consumption.

$\mathsf{STEAL}(n)$**:** All elements whose production would be voided by n, or by some $m \in T(n)$ without being resupplied by a descendant of m within $T(n)$.

$\mathsf{GIVE}(n)$**:** All elements that are already produced at n, or at some node in $T(n)$ without being stolen later within $T(n)$.

$\mathsf{BLOCK}(n)$**:** Elements whose production is blocked by n, *i.e.*, whose production cannot be hoisted across n because it is stolen or already produced at n or a node in $T(n)$.

$$
\begin{aligned}
\mathsf{STEAL}(n) &= \mathsf{STEAL}_{init}(n) \cup \mathsf{STEAL}_{loc}(\textsc{LastChild}(n)) && (1)\\
\mathsf{GIVE}(n) &= \mathsf{GIVE}_{init}(n) \cup \mathsf{GIVE}_{loc}(\textsc{LastChild}(n)) && (2)\\
\mathsf{BLOCK}(n) &= \mathsf{STEAL}(n) \cup \mathsf{GIVE}(n) \cup \mathsf{BLOCK}_{loc}(\textsc{FirstChild}(n)) && (3)\\
\mathsf{TAKEN}_{out}(n) &= \bigcap_{s \in \textsc{Succs}^{+}(n)} \mathsf{TAKEN}_{in}(s) && (4)\\
\mathsf{TAKE}(n) &= \mathsf{TAKE}_{init}(n) \cup (\mathsf{TAKEN}_{in}(\textsc{FirstChild}(n)) \setminus \mathsf{STEAL}(n))\\
&\quad \cup(\mathsf{TAKE}_{loc}(\textsc{FirstChild}(n)) \cap (\mathsf{TAKEN}_{out}(n) \setminus \mathsf{BLOCK}(n))) && (5)\\
\mathsf{TAKEN}_{in}(n) &= \mathsf{TAKE}(n) \cup (\mathsf{TAKEN}_{out}(n) \setminus \mathsf{BLOCK}(n)) && (6)\\
\mathsf{BLOCK}_{loc}(n) &= (\mathsf{BLOCK}(n) \cup \bigcup_{s \in \textsc{Succs}^{-}(n)} \mathsf{BLOCK}_{loc}(s)) \setminus \mathsf{TAKE}(n) && (7)\\
\mathsf{TAKE}_{loc}(n) &= \mathsf{TAKE}(n) \cup (\bigcup_{s \in \textsc{Succs}^{-}(n) \cup \textsc{FirstChild}(n)} \mathsf{TAKE}_{loc}(s) \setminus \mathsf{BLOCK}(n)) && (8)
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{GIVE}_{loc}(n) &= (\mathsf{GIVE}(n) \cup \mathsf{TAKE}(n) \cup \bigcap_{p \in \textsc{Preds}(n)} \mathsf{GIVE}_{loc}(p)) \setminus \mathsf{STEAL}(n) && (9)\\
\mathsf{STEAL}_{loc}(n) &= \mathsf{STEAL}(n) \cup \bigcup_{p \in \textsc{Preds}(n)} (\mathsf{STEAL}_{loc}(p) \setminus \mathsf{GIVE}_{loc}(p)) \cup \bigcup_{p \in \textsc{Preds}^{+}(n) \setminus \textsc{Preds}(n)} \mathsf{STEAL}_{loc}(p) && (10)
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{GIVEN}_{in}(n) &= \begin{cases} \mathsf{GIVEN}(\textsc{Header}(n)) & \text{if } n \text{ is a first child,}\\ \bigcap_{p \in \textsc{Preds}(n)} \mathsf{GIVEN}_{out}(p) \cup (\mathsf{TAKEN}_{in}(n) \cap \bigcup_{p \in \textsc{Preds}(n)} \mathsf{GIVEN}_{out}(p)) & \text{otherwise.} \end{cases} && (11)\\
\mathsf{GIVEN}(n) &= \mathsf{GIVEN}_{in}(n) \cup \begin{cases} \mathsf{TAKEN}_{in}(n) & \text{for an } \textsc{Eager} \text{ Problem,}\\ \mathsf{TAKE}(n) & \text{for a } \textsc{Lazy} \text{ Problem.} \end{cases} && (12)\\
\mathsf{GIVEN}_{out}(n) &= (\mathsf{GIVE}(n) \cup \mathsf{GIVEN}(n)) \setminus \mathsf{STEAL}(n) && (13)
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{RES}_{in}(n) &= \mathsf{GIVEN}(n) \setminus \mathsf{GIVEN}_{in}(n) && (14)\\
\mathsf{RES}_{out}(n) &= \bigcup_{s \in \textsc{Succs}(n)} \mathsf{GIVEN}_{in}(s) \setminus \mathsf{GIVEN}_{out}(n) && (15)
\end{aligned}
$$

Figure 8: GIVE-N-TAKE equations.

$\mathsf{TAKEN}_{out}(n)$: Things guaranteed to be consumed (before being stolen) on all paths originating in n, excluding n itself. Here we also have to take loop exit edges originating in $T(n)$ into account.

$\mathsf{TAKE}(n)$: The set of consumers at n. This includes items that are guaranteed to be consumed by nodes in $T(n)$ and not stolen at n, and items that may be consumed by $T(n)$ and are guaranteed to be consumed on exit from n without being blocked by n.

$\mathsf{TAKEN}_{in}(n)$: Similar to TAKEN_{out}, except that the effect of n itself is included.

$\mathsf{BLOCK}_{loc}(n)$: Items blocked by n or by descendants of n within $J(n)$.

$\mathsf{TAKE}_{loc}(n)$: Items taken by n, by descendants of n within $J(n)$, or by nodes within $T(n)$. Here (unlike for BLOCK_{loc}) we have to explicitly include $\mathsf{TAKE}_{loc}(\textsc{FirstChild}(n))$ since this is not guaranteed to be in TAKE(whereas $\mathsf{BLOCK}_{loc}(\textsc{FirstChild}(n))$ is always included in $\mathsf{BLOCK}(n)$).

2.3.3 Blocking consumption

The following variables are used by the interval headers to determine whether items are stolen or taken within the interval.

$\mathsf{GIVE}_{loc}(n)$**:** Items produced by n or by ancestors of n within the same interval. Here items are treated as produced also if they are consumed, since consumption is guaranteed to be satisfied by production.

$\mathsf{STEAL}_{loc}(n)$**:** Items stolen by n, or stolen by a predecessor p of n without being resupplied by p. Furthermore, if there exists an $h \in \text{PREDS}^{+}(n) \setminus \text{PREDS}(n)$ (*i.e.*, n is the sink of an exit edge, and h is the header of an interval enclosing the source of the exit edge but not n itself), then we also have to include items stolen by h; however, since the interval headed by h is not guaranteed to be completed before n is reached (since taking the exit edge corresponds to a jump from within the interval), we cannot exclude items resupplied by h.

2.3.4 Placing production

After analyzing what is consumed (and not already produced) at each node, the production needed to satisfy all consumers is computed by the following variables.

$\mathsf{GIVEN}_{in}(n)$**:** Things that are guaranteed to be available at the entry of n. If n is a first child, then it has everything available that is available at $\text{HEADER}(n)$. Otherwise, things are guaranteed to be produced if they are produced along all incoming paths, or if they are produced at least along some incoming paths and guaranteed to be consumed. In the latter case, the result variables will ensure that things will be produced also along the paths that originally did not have them available.

$\mathsf{GIVEN}(n)$**:** Items guaranteed to be available at n itself, either because they come from predecessors of n, or because they are consumed at n itself (for a LAZY problem) or a descendant of n (for an EAGER problem).

$\mathsf{GIVEN}_{out}(n)$**:** Things that are available on exit from n. This includes whatever comes for free at n itself, but it excludes things stolen by n.

2.3.5 Result variables

The result of GIVE-N-TAKE analysis is expressed by the following variables.

$\mathsf{RES}_{in}(n)$**:** The production generated at the entry of n. This includes everything that is guaranteed to be available at n itself but is not yet available at the entry of n.

$\mathsf{RES}_{out}(n)$**:** The production at the exit of n. This includes items whose availability has been guaranteed to some successors of n and that are not already available on exit from n.

2.4 The algorithm

This section presents an algorithm, *GiveNTake*, for solving a code placement problem using the GIVE-N-TAKE framework. Section 2.3 already presented the equations that lead from the initial data flow variables to the result variables. What is left towards an actual algorithm is a recipe for the order in which to evaluate these equations such that the values of the data flow variables reach a fixed point that is consistent with all equations (where a fixed point might be reached after a constant number of iterations, as is generally the case in interval analysis). In general, the evaluation order is also important for the convergence rate and, in some cases, termination behavior of the algorithm. In our case, there exists an order where the right hand side of each equation to be evaluated is already fully known due to previous computation. Therefore, *GiveNTake* has to evaluate each equation only once for each node, which implies guaranteed termination and low computational complexity. However, since the direction of the flow of information varies across the equations, we still need multiple passes over the control flow graph.

An objective for *GiveNTake* is to minimize the number of passes, therefore we partition the equations into different sets that can be evaluated concurrently, *i.e.*, within the same pass. It turns out that each of the Sections 2.3.2, 2.3.3, 2.3.4, and 2.3.5 defines one set of equations that can be evaluated concurrently. We will refer to these sets as S_1 (Equations 1...8), S_2 (Equations 9, 10), S_3 (Equations 11...13), and S_4 (Equations 14,15), respectively. Note that since all equations except Equation 12 in S_3 are the same for EAGER and LAZY problems and since S_1 and S_2 are computed before S_3, the variables defined in S_1 and S_2 are the same for both kinds of

```
**Procedure** *GiveNTake*

**Input:** $G = (N, E)$; $\forall n \in N$: $\mathsf{TAKE}_{init}(n)$, $\mathsf{STEAL}_{init}(n)$, $\mathsf{GIVE}_{init}(n)$
**Output:** $\forall n \in N$: $\mathsf{RES}^{eager}(n)$ and/or $\mathsf{RES}^{lazy}(n)$

forall $n \in N$, in ReversePreOrder[PostOrder]
    if (isHeader($n$)) then
        forall $c \in$ Children($n$), in Forward[Backward] order
            Compute Equations 9, 10
        endforall
    endif
    Compute Equations 1...8
endforall
forall $n \in N$, in PreOrder[ReversePostOrder]
    Compute Equations 11...13 for Eager and/or for Lazy
endforall
forall $n \in N$
    Compute Equations 14,15 for Eager and/or for Lazy
endforall
end
```

Figure 9: Algorithm *GiveNTake* for a Before[After] problem, computing an Eager and/or Lazy code generation.

problems. Therefore, we need to differentiate between Eager and Lazy only for variables defined in S_3 and S_4. We distinguish these variables by superscripts *eager* and *lazy*.

The resulting algorithm, with orderings given for a Before problem (followed by the corresponding orderings for an After problem in brackets), is shown in Figure 9. RES without subscripts stands for both RES_{in} and RES_{out}. A proof that it does indeed obey all ordering constraints can be found in Appendix B.

3 An example application: Communication analysis

The following sections outline the application of the Give-N-Take framework in the Fortran D compiler for generating Read's, *i.e.* reads of locally referenced not-owned data, and Write's, *i.e.* writes of locally defined not-owned data. In addition to the applications listed here, our actual implementation also recognizes reduction operations (like accumulating a sum or product) and generates special, combining messages for them (like WriteAdd and WriteMult).

3.1 The data flow universe

Our data flow universe consists of the set of *array portions* occurring in the program. An array portion corresponds to one or several array occurrences (i.e. references or definitions) of the form $x(sub)$, where x is an array and sub an arbitrary subscript expression, like a loop index or an indirection array lookup. A portion is identified by the following fields:

- Data array; *i.e.*, the symbol table index of x.
- Subscript value; *i.e.*, the value number of sub. This value numbering makes the portion classification independent of its syntactic representation. For example, in the program shown in Figure 10, value numbering allows us to recognize that $v(j)$ and $v(k)$ are equivalent.
- Data array distribution; *i.e.*, the distribution in effect for x at the point of reference. We assume that there is only one distribution statement reaching each occurrence; this can be ensured, for example, by cloning occurrences that are reached by multiple distributions [HHKT92].

- Computation distribution; *i.e.*, how the computation of the statement containing the occurrence is distributed.

Two occurrences are assigned to the same portion iff they agree on all four fields. For each control flow graph node n, each data flow variable is represented as a bitvector corresponding to the set of all portions; each portion corresponds to one bit. Our current prototype has a control flow graph node for every statement; however, they could also be basic blocks.

Each subscript of each portion may correspond to different sets of array elements, depending on the nesting level. For example, consider a loop l indexed by j that is nested in a loop k indexed by i, where the lower and upper bounds on i, j are i_1, i_2, j_1, j_2, respectively. Depending on our "point of view" within this loop nest, a portion $x(i,j)$ may correspond to $x(i_1 : i_2, j_1 : j_2)$, $x(i, j_1 : j_2)$, or just $x(i,j)$. At the deepest nesting level, a portion always corresponds to a single datum. This has to be kept in mind when generating the communication statements.

Since sends and receives have to match each other not only in the data that are exchanged but also in the way how these data are packaged into messages, we have to make sure that data are only sent together if they are also received together and vice versa. In the code example, the receive matching "READ Send {[z(1:25)]}" is contained in "READ Recv {w(a(1:25)), [z(1:25)]}." To generate executable code, a heuristic must be applied to split the receive into two individual statements. *(Not implemented yet.)*

3.2 Not-owned references

To combine messages and to hide message latencies, we want to read as early as possible, *i.e.*, READ calls should be ***shifted up*** in the control flow graph. Therefore, generating READ statements is a BEFORE problem. For generating monolithic statements (*i.e.*, a single statement that combines the send and receive), we generate code as indicated by RES^{eager}. Otherwise, RES^{eager} generates the sends and RES^{lazy} generates the matching receives.

We initialize the data flow variables as follows:

$$\mathsf{TAKE}_{init}(n) = \{p \mid n \text{ references } p\}, \tag{16}$$

$$\mathsf{STEAL}_{init}(n) = \begin{cases} \top & \text{if } n = \textsc{Start}, \\ \{p \mid n \text{ redefines the subscript of } p \text{ or partially redefines } p\} & \text{otherwise;} \end{cases} \tag{17}$$

$$\mathsf{GIVE}_{init}(n) = \{p \mid n \text{ defines } p\}. \tag{18}$$

Defining $\mathsf{STEAL}_{init}(\textsc{Start}) = \top$ implies that everything is stolen at the beginning of the program, which is a safeguard against advancing production out of the program.

When treating local definitions of not-owned data as additional producers, we must be sure that these not-owned definitions are unique, *i.e.*, that there is no other processor defining the same elements. This can be a non-trivial problem, especially in the presence of indirection arrays, in which case user assistance (for example by rewriting an enclosing DO loop as a FORALL loop) might be needed.

3.3 Not-owned definitions

To combine messages, we want to write as late as possible, *i.e.*, write calls should be *shifted down* in the control flow graph. Therefore, this is an AFTER problem, and when splitting communication, RES^{eager} generates the receives and RES^{lazy}generates the sends.

We initialize the data flow variables as follows:

$$\mathsf{TAKE}_{init}(n) = \{p \mid n \text{ defines } p\}; \tag{19}$$

$$\mathsf{STEAL}_{init}(n) = \{p \mid n \text{ redefines the subscript of } p, \text{ partially references } p, \text{ or partially reads } p\}; \tag{20}$$

$$\mathsf{GIVE}_{init}(n) = \begin{cases} \top & \text{if } n = \textsc{Stop}, \\ \bot & \text{otherwise.} \end{cases} \tag{21}$$

Defining $\mathsf{GIVE}_{init}(\textsc{Stop}) = \top$ suppresses WRITE's at the very end of the program (by making them available for free), which implies that everything is dead at the end of the program.

```
**program** prog$_{orig}$

**do** $h = 1, 20$
    **do** $i = 1, 100$
        $u(i) = v(a(i)) + v(b(i))$
        $t(b(i)) = u(i)$
    **enddo**

    **if** (mod($h$, 5) .eq. 0) then
        **if** ($h$ .eq. 20) goto 7

        **do** $j = 1, 100$
            $v(j) = w(a(j))$
        **enddo**

        **do** $k = 1, 100$
            $v(k) = u(a(k)) + 2 * v(k)$
        **enddo**
    **endif**

    **do** $l = 1, 100$
        $w(l) = t(a(l)) + u(a(l)) + v(a(l)) + x(a(l))$
    **enddo**
**enddo**

**do** $m = 1, 100$
    $x(b(m)) = v(m) + x(m) + z(a(m))$
**enddo**

7 **do** $n = 1, 100$
    $y(n) = u(a(n)) + w(a(n)) + x(a(n)) + z(n) + t(c(n))$
**enddo**
**end**
```

```
**program** prog$_{trans}$

**READ Snd** {[$z$(1:25)]}
**READ Snd/Rcv** {$v(a$(1:25)), $x(a$(1:25))}
**do** $h = 1, 20$
    **READ Snd/Rcv** {$v(b$(1:25))}
    **do** $i = 1, 25$
        $u(i) = v(a(i)) + v(b(i))$
        $t(b(i)) = u(i)$
    **enddo**
    **WRITE Snd** {$t(b$(1:25))}
    **WRITE Snd/Rcv** {[$u$(1:25)]}
    **READ Snd** {$u(a$(1:25))}
    **if** (mod($h$, 5) .eq. 0) then
        **READ Snd** {$w(a$(1:25))}
        **if** ($h$ .eq. 20) then
            **WRITE Rcv** {$t(b$(1:25))}
            **READ Snd** {$t(c$(1:25))}
            **READ Rcv** {$u(a$(1:25))}
            **goto** 7
        **endif**
        **WRITE Rcv** {$t(b$(1:25))}
        **READ Snd** {$t(a$(1:25))}
        **READ Rcv** {$w(a$(1:25))}
        **do** $j = 1, 25$
            $v(j) = w(a(j))$
        **enddo**
        **READ Rcv** {$u(a$(1:25))}
        **do** $k = 1, 25$
            $v(k) = u(a(k)) + 2 * v(k)$
        **enddo**
        **WRITE Snd/Rcv** {[$v$(1:25)]}
        **READ Snd/Rcv** {$v(a$(1:25))}
    **else**
        **WRITE Rcv** {$t(b$(1:25))}
        **READ Snd** {$t(a$(1:25))}
        **READ Rcv** {$u(a$(1:25))}
    **endif**
    **READ Rcv** {$t(a$(1:25))}
    **do** $l = 1, 25$
        $w(l) = t(a(l)) + u(a(l)) + v(a(l)) + x(a(l))$
    **enddo**
    **WRITE Snd/Rcv** {[$w$(1:25)]}
**enddo**
**READ Snd** {$t(c$(1:25)), $w(a$(1:25))}
**READ Snd/Rcv** {[$v$(1:25)], $z(a$(1:25))}
**do** $m = 1, 25$
    **READ Snd/Rcv** {[$x(m)$]}
    $x(b(m)) = v(m) + x(m) + z(a(m))$
    **WRITE Snd/Rcv** {$x(b(m))$}
**enddo**
**READ Snd/Rcv** {$x(a$(1:25))}
7 **continue**
**READ Rcv** {$t(c$(1:25)), $w(a$(1:25)), [$z$(1:25)]}
**do** $n = 1, 25$
    $y(n) = u(a(n)) + w(a(n)) + x(a(n)) + z(n) + t(c(n))$
**enddo**
**end**
```

Figure 10: Original and transformed code, assuming four processors.

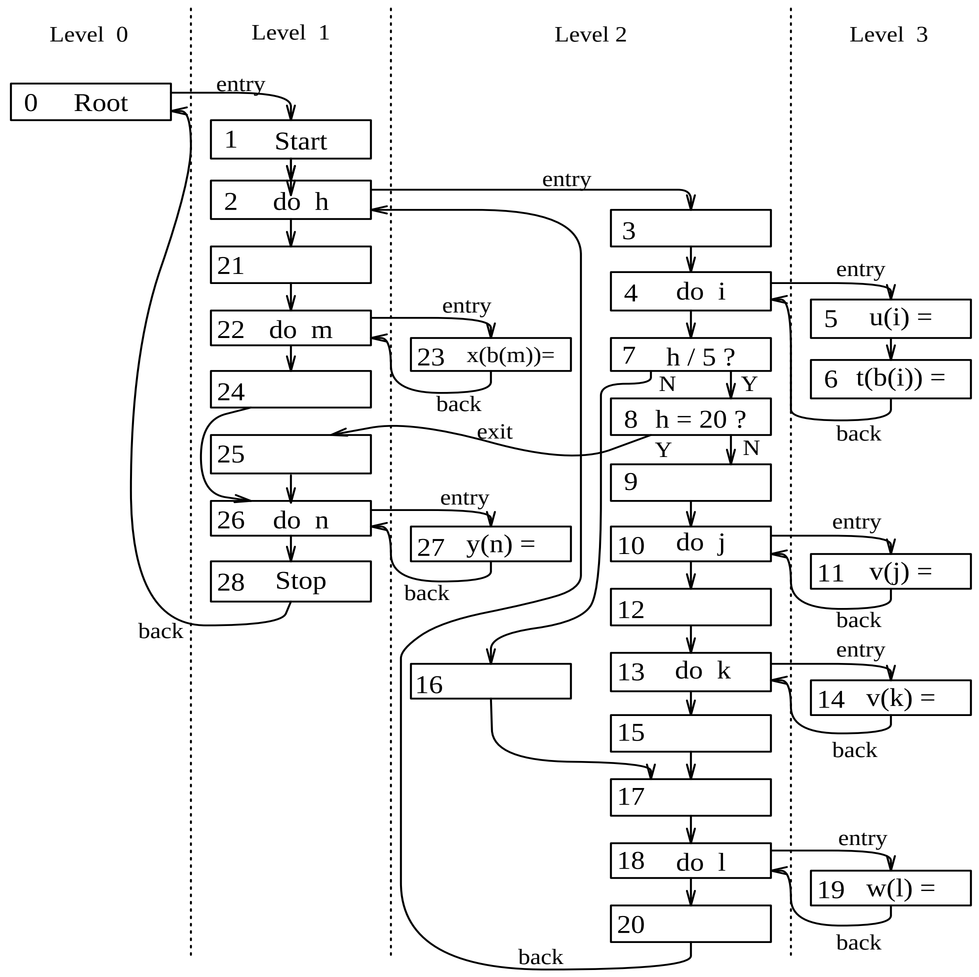

Figure 11: Control flow graph for example program.

3.4 Example

Figure 10 shows a contrived example program, $prog_{orig}$, and the corresponding output of the FORTRAN D compiler, $prog_{trans}$. The corresponding flow graph is in Figure 11. Data declarations, initializations, distribution statements, etc., have been omitted. Communication schedule generation, which is a non-trivial problem in itself [HKK+92], is also excluded. For simplicity, we assume that all arrays are distributed BLOCK-wise and that we use the owner computes rule. In case of an indirect definition (like the $x(b(m))$), we assume that computation is distributed based on the owner of the indirection array. In *prog*, these assumptions result in distributing all inner loops by performing loop bounds reduction (even though the carried dependence on the m loop effectively makes it sequential). Note that *prog* does not have any output dependences that are not already covered by other dependences. For example, the potential output dependence on $w(b(m))$ carried by the m loop is accompanied by a true dependence that also prohibits running the m loop in parallel; therefore, we do not violate the output dependence arising from a sequential interpretation of the m loop. In general, however, indirect array definitions may make parallelization unsafe, unless the compiler recognizes the computation as a reduction operation, it can prove that the indirection array is a permutation, or the loop is explicitly parallel.

The transformed code contains communication statements in a high level format that does not include any schedule parameters, message tags, and so on. The possible communication types are the following.

READ Snd $\{p\}$**:** The owner of p sends p to a processor that will reference p.

READ Rcv $\{p\}$**:** A processor that will reference p receives p from the owner.

WRITE Snd $\{p\}$**:** A processor that has defined p but does not own p sends p to the owner of p.

WRITE Rcv $\{p\}$**:** The owner of p receives p from a processor that has defined p.

"READ Snd/Rcv $\{p\}$" is an abbreviation for a "READ Snd $\{p\}$" followed by a "READ Rcv $\{p\}$," similarly for "WRITE Snd/Rcv $\{p\}$." For simplicity we also do not consider already locally available data, like the $z(1{:}25)$, that are owned by each processor but for which *prog* still has a READ. Furthermore, when reading $p_1 = v(b(1{:}25))$, for example, we know that $p_2 = v(a(1{:}25))$ is already available, so we only need to send data that are in p_1 and not in p_2, see also Section 4. When generating messages in their final form, we also have to take into account that the transformed code has a local name space on each processor, whereas the original code operates with a single global name space.

Some specific comments on the generated communication:

- The "READ Snd $\{[z(1{:}25)]\}$" at the beginning of the program is matched by the "READ Rcv $\{w(a(1{:}25)), [z(1{:}25)]\}$" before the n loop that contains the reference to $z(n)$. Communication is hoisted out of the n loop, and the h and m loops can hide the message latency.
- Reading $v(a(1{:}25))$ is hoisted out of the i and l loops that contain references to $v(a(i))$ and $v(a(l))$, respectively, then it is combined and hoisted out of the h loop as well. However, $v(a(1{:}25))$ gets stolen by the true-branch of the conditional in the h loop, so it has to be read again at the end of that branch.
- Reading $v(b(1{:}25))$ is not hoisted out of the h loop, since unlike the $v(a(1{:}25))$, it is not guaranteed to be available at the end of the h loop.
- There is no READ for the reference to $u(i)$ since it already come for free due to the proceeding definition of $u(i)$.
- $u(a(1{:}25))$ is referenced in the k, l, and n loops. It is read by one send and three receive statements, one for each possible flow of control. Similarly, $t(b(1{:}25))$ is written by one send and three receives. This necessitated introducing an ELSE branch to the IF statement and turning the conditional branch into a conditional block terminated by an unconditional branch.
- $w(a(1{:}25))$ is referenced in the j and n loops. It is read by two sends and two receives, where both receives can match the first send.
- The m loop carries a dependence on x, therefore the READ for $x(m)$ and the WRITE of $x(b(m))$ are not hoisted out of the m loop.
- Reading $z(a(1{:}25))$ is not shifted before the h loop because of the conditional jump out of the h loop that goes around the $z(a(m))$ reference. This was not the case for the $z(n)$ reference that could therefore be communicated at the very beginning of the program.
- $y(1{:}25)$ is recognized as dead at the end of the program and therefore not written.

In general, we assume that aggregate communication primitives capable of handling irregular array accesses are used for communication, like for example the PARTI communications library [SBW90]. However, the compiler recognizes regular subscripts (for example, subscripts linear in loop variables or auxiliary induction variables) and communicates these using simpler, faster communication routines. In *prog*, such simple subscripts are enclosed in brackets (like "$[x(1{:}25)]$").

4 Extensions and Conclusions

After running *GiveNTake*, RES contains a producer placement that satisfies the correctness and optimality constraints. Code can be generated by mapping each n to the source code and inserting communication statements as indicated by RES(n). This, however, turns out to be a non-trivial task in itself, because there may be no location in the source code directly corresponding to a given n. For example, if communication is placed at a synthetic node that was created to break a critical edge, then we might have to create a new basic block, like an additional ELSE branch for an IF statement. Furthermore, we might generate basic blocks unnecessarily when doing this naïvely. For this reason, we supplemented our implementation of the GIVE-N-TAKE framework with a data flow phase that does additional postprocessing on the generated results. This phase shifts results out of synthetic nodes if this can be done without altering the run time behavior of the program.

While this paper focuses on the data flow analysis nature of GIVE-N-TAKE, we also would like to take advantage of dependence analysis that can legitimate some code placements that would otherwise be considered unsafe. For example, assume an i loop enclosing a j loop that in turn encloses a definition S_1 of $p_1 = x(i)$ and a reference S_2 of $p_2 = x(i-1)$, and no other occurrences of x. Dependence analysis discovers that the true dependence $r_1 \delta_i r_2$ is carried by the outer i loop, and therefore it is legal to hoist the communication of p_2 out of the inner j loop. It is relatively straightforward to provide this information to Give-N-Take by shifting STEAL sets out from the original statements. In the example, instead of including p_2 in $\mathsf{STEAL}(S_1)$, we can include p_2 in $\mathsf{STEAL}(l)$, where l is the header of the i loop. Since p_2 is then not stolen any more in the j loop, the communication will be hoisted out as desired.

We can also take advantage of overlapping portions by taking into account what data are already available when generating communication. This information is already contained in GIVEN_{in} (for RES_{in}) and GIVEN_{out} (for RES_{out}). An *incremental schedule* can take advantage of these situations [HKK+92]. A similar effect can be achieved with *mode vectors* [GV91]; however, they do so not by compile time analysis but by keeping track of available data at run time.

To summarize, this paper has outlined a general code generation framework that is based on and subsumes partial redundancy elimination as one of several types of problems (Before/After, Eager/Lazy) that can be solved with the framework. An important property is the Balancedness of the solutions Give-N-Take produces when asked for both an Eager and a Lazy placement. We have proven the framework's correctness and illustrated its efficiency when applying it to generating messages for distributed memory computers. Its flexibility allowed us to apply the same algorithm to very different tasks that traditionally were solved with separate frameworks. This simplified the implementation in the Fortran D compiler significantly. We expect Give-N-Take to have potential use in other areas as well, like general memory hierarchy issues (cache prefetching, register allocation, parallel I/O) and classic partial redundancy elimination applications.

References

[AL93] S. P. Amarasinghe and M. S. Lam. Communication optimization and code generation for distributed memory machines. *ACM SIGPLAN Notices*, 28(6):126–138, June 1993. *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation.*

[All70] F. E. Allen. Control flow analysis. *ACM SIGPLAN Notices*, 5(7):1–19, 1970.

[CK92] S. Carr and K. Kennedy. Scalar replacement in the presence of conditional control flow. Technical Report TR92283, Rice University, CRPC, November 1992. To appear in *Software – Practice & Experience.*

[CM69] J. Cocke and R. Miller. Some analysis techniques for optimizing computer programs. In *Proceedings of the 2nd Annual Hawaii International Conference on System Sciences*, pages 143–146, 1969.

[Coc70] J. Cocke. Global common subexpression elimination. *ACM SIGPLAN Notices*, 5(7):20–24, 1970.

[DGS93] E. Duesterwald, R. Gupta, and M. L. Soffa. A practical data flow framework for array reference analysis and its use in optimizations. *ACM SIGPLAN Notices*, 28(6):68–77, June 1993. *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation.*

[Dha88a] D.M. Dhamdhere. A fast algorithm for code movement optimization. *ACM SIGPLAN Notices*, 23(10):172–180, 1988.

[Dha88b] D.M. Dhamdhere. Register assignment using code placement techniques. *Computer Languages*, 13(2):75–93, 1988.

[Dha91] D.M. Dhamdhere. Practical adaptation of the global optimization algorithm of Morel and Renvoise. *ACM Transactions on Programming Languages and Systems*, 13(2):291–294, April 1991.

[DK83] D.M. Dhamdhere and J.S. Keith. Characterization of program loops in code optimization. *Computer Languages*, 8:69–76, 1983.

[DK93] D. M. Dhamdhere and U. P. Khedker. Complexity of bidirectional data flow analysis. In *Conference Record of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, pages 397–408, Charleston, South Carolina, January 1993.

[DP93] D.M. Dhamdhere and H. Patil. An elimination algorithm for bidirectional data flow problems using edge placement. *ACM Transactions on Programming Languages and Systems*, 15(2):312–336, April 1993.

[DRZ92] D.M. Dhamdhere, B.K. Rosen, and F.K. Zadeck. How to analyze large programs efficiently and informatively. In *Proceedings of the ACM SIGPLAN '92 Conference on Program Language Design and Implementation*, pages 212–223, San Francisco, CA, June 1992.

[DS88] K. Drechsler and M. Stadel. A solution to a problem with Morel and Renvoise's "Global optimization by suppression of partial redundancies". *ACM Transactions on Programming Languages and Systems*, 10(4):635–640, October 1988.

[GS90] T. Gross and P. Steenkiste. Structured dataflow analysis for arrays and its use in an optimizing compiler. *Software—Practice and Experience*, 20(2):133–155, February 1990.

[GS93] M. Gupta and E. Schonberg. A framework for exploiting data availability to optimize communication. In *Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing*, Portland, OR, August 1993.

[GV91] E. Granston and A. Veidenbaum. Detecting redundant accesses to array data. In *Proceedings of Supercomputing '91*, Albuquerque, NM, November 1991.

[HHKT92] M. W. Hall, S. Hiranandani, K. Kennedy, and C. Tseng. Interprocedural compilation of Fortran D for MIMD distributed-memory machines. In *Proceedings of Supercomputing '92*, Minneapolis, MN, November 1992.

[Hig93] High Performance Fortran Forum. High Performance Fortran language specification, version 1.0. Technical Report CRPC-TR92225, Center for Research on Parallel Computation, Rice University, Houston, TX, 1992 (revised Jan. 1993). To appear in *Scientific Programming*, July 1993.

[HKK$^+$92] R. v. Hanxleden, K. Kennedy, C. Koelbel, R. Das, and J. Saltz. Compiler analysis for irregular problems in Fortran D. In *Proceedings of the Fifth Workshop on Languages and Compilers for Parallel Computing*, New Haven, CT, August 1992. Available via anonymous ftp from `softlib.rice.edu` as `pub/CRPC-TRs/reports/CRPC-TR92287-S`.

[HKT92a] S. Hiranandani, K. Kennedy, and C. Tseng. Compiler support for machine-independent parallel programming in Fortran D. In J. Saltz and P. Mehrotra, editors, *Languages, Compilers, and Run-Time Environments for Distributed Memory Machines*. North-Holland, Amsterdam, The Netherlands, 1992.

[HKT92b] S. Hiranandani, K. Kennedy, and C. Tseng. Evaluation of compiler optimizations for Fortran D on MIMD distributed-memory machines. In *Proceedings of the 1992 ACM International Conference on Supercomputing*, Washington, DC, July 1992.

[JD82] S.M. Joshi and D.M. Dhamdhere. A composite hoisting-strength reduction transformation for global program optimization, parts I & II. *International Journal of Computer Mathematics*, 11:21–41, 111–126, 1982.

[Ken71] K. Kennedy. A global flow analysis algorithm. *International Journal of Computer Mathematics*, 3:5–15, 1971.

[KRS92] J. Knoop, O. Rüthing, and B. Steffen. Lazy code motion. In *Proceedings of the ACM SIGPLAN '92 Conference on Program Language Design and Implementation*, San Francisco, CA, June 1992.

[MR79] E. Morel and C. Renvoise. Global optimization by suppression of partial redundancies. *Communications of the ACM*, 22(2):96–103, February 1979.

[SBW90] J. Saltz, H. Berryman, and J. Wu. Multiprocessors and runtime compilation. ICASE Report 90-59, Institute for Computer Application in Science and Engineering, Hampton, VA, September 1990.

[Sor89] A. Sorkin. Some comments on "A solution to a problem with Morel and Renvoise's 'Global optimization by suppression of partial redundancies'". *ACM Transactions on Programming Languages and Systems*, 11(4):666–668, October 1989.

[Tar74] R. E. Tarjan. Testing flow graph reducibility. *Journal of Computer and System Sciences*, 9:355–365, 1974.

A Proof of correctness of the data flow equations

To simplify the discussion of correctness for different cases of control flow, we expand a control flow path into its different *program points* [DK93], namely the entries and exits of nodes passed through. For

$$p : (\textsc{Start} =)\ n_0 \rightarrow n_1 \rightarrow \ldots \rightarrow n_s\ (= \textsc{Stop}),$$

we define the *expanded control flow path* to be

$$E(p) : ((\textsc{Start}, out) =)\ (q_0, r_0) \rightarrow (q_1, r_1) \rightarrow \ldots \rightarrow (q_t, r_t)\ (= (\textsc{Stop}, in)),$$

where $t \geq s$ and $\forall i, 0 \leq i \leq t : q_i \in N, r_i \in \{in, out\}$. (q_i, in) stands for the *entry* of q_i, and (q_i, out) denotes the *exit* of q_i. Of course, paths and expanded paths do not always have to originate at START and terminate at STOP.

How exactly an edge $e = (m, n)$ is expanded depends on the type of edge (Section 2.2.2).

- If e is a forward or exit edge, then $E(e) = (m, out) \rightarrow (n, in)$.
- If e is an entry edge, then $E(e) = (m, in) \rightarrow (n, in)$.
- If e is a back edge, then $E(e) = (m, out) \rightarrow (n, out)$.
- Note that edges of the form $(n, in) \rightarrow (n, out)$ do not correspond to actual edges in the control flow graph; we will refer to such edges as *internal edges.*
- Furthermore, there is no direct expanded equivalent for a control flow path that goes from the end of a loop through the loop header back to the beginning of the loop, since it goes through the header without going through either its entry or its exit. This, however, does not affect our proofs, since we can take the full effect of each loop into account by traversing it once (we might also not execute it at all).

For example, a path (with a loop iteration)

$$p : 1 \rightarrow 2 \rightarrow 3 \rightarrow 2 \rightarrow 4$$

would be expanded into

$$E(p) : \quad (1, out) \rightarrow (2, in) \rightarrow (3, in) \rightarrow (3, out) \rightarrow (2, out) \rightarrow (4, in)$$

Let x be an arbitrary entry in our data flow universe. For a data flow variable VAR and a node $n \in N$, let "$\mathsf{VAR}(n)$" denote $x \in \mathsf{VAR}$, and let "$\overline{\mathsf{VAR}(n)}$" denote $x \notin \mathsf{VAR}$.

A.1 Balancedness

A.1.1 The Balancedness Theorem

Balancedness is equivalent to matching, at run time (*i.e.*, along all possible control flow paths), all EAGER productions with succeeding LAZY productions, without any intervening EAGER production, and vice versa. This has to hold for for BEFORE and AFTER solutions, but with different orientations relative to the flow of control. The following theorem expresses these constraints.

Theorem 1 (C1: Balancedness.) Given: *An expanded path*

$$p : ((\textsc{Start}, \text{out}) =)\ (q_0, r_0) \rightarrow (q_1, r_1) \rightarrow \ldots \rightarrow (q_s, r_s)\ (= (\textsc{Stop}, \text{in}))$$

with an EAGER *production for some e, $0 < e < s$:*

$$\mathsf{RES}^{\text{eager}}_{r_e}(q_e). \tag{22}$$

Claim: *In a* BEFORE *solution, there exists an $q_l \in N, e \leq l < s$, such that*

$$\mathsf{RES}^{\text{lazy}}_{r_l}(q_l), \tag{23}$$

$$\forall q_k \in N, e < k \leq l : \overline{\mathsf{RES}^{\text{eager}}_{r_k}(q_k)}. \tag{24}$$

Similarly, a LAZY *production $\mathsf{RES}^{\text{lazy}}_{r_e}(q_e)$ is balanced by an* EAGER *production. In an* AFTER *solution, the same holds with the flow of control reversed.*

We will show the proof for balancing EAGER production in a BEFORE problem (in our communication generation application, this corresponds to guaranteeing that each READ$_{Send}$ will be matched by a READ$_{Recv}$, without any READ$_{Send}$ in between). The proofs for the other cases are analogous.

The underlying idea is to show that if an EAGER production has been placed on a path but no LAZY production has taken place yet, then this information is propagated forward (for a BEFORE problem) or backward (for an AFTER problem) by both GIVEN^{eager} and TAKEN being true and by GIVEN^{lazy} being false (control flow graph node numbers omitted here). The following contains some lemmata on which the inductive proof of Theorem 1 will be based (Section A.1.4).

A.1.2 Induction base lemmata

Lemma 1 (EAGER implies LAZY.) Claim: *For all $n \in N$, the following holds:*

$$\mathsf{GIVEN}_{\text{in}}^{\text{eager}}(l) \supseteq \mathsf{GIVEN}_{\text{in}}^{\text{lazy}}(l), \tag{25}$$

$$\mathsf{GIVEN}^{\text{eager}}(l) \supseteq \mathsf{GIVEN}^{\text{lazy}}(l), \tag{26}$$

$$\mathsf{GIVEN}_{\text{out}}^{\text{eager}}(l) \supseteq \mathsf{GIVEN}_{\text{out}}^{\text{lazy}}(l). \tag{27}$$

Proof:

The only difference between EAGER and LAZY is in Equation (12), where GIVEN^{eager} gets induced by TAKEN_{in} and GIVEN^{lazy} follows from TAKE. (6) implies that for all $n \in N$:

$$\mathsf{TAKEN}_{in}(n) \supseteq \mathsf{TAKE}(n). \tag{28}$$

PREDS(ROOT) $= \emptyset$ implies by (11) that $\mathsf{GIVEN}_{in}^{eager}(\text{ROOT}) = \mathsf{GIVEN}_{in}^{lazy}(\text{ROOT}) = \perp$, which can serve as a base for a simple induction using (11), (12), (13), and (28) to prove (25), (26), and (27).

□

Lemma 2 (Production at entry.) Given: *A node $n \in N$ such that*

$$\mathsf{RES}_{\text{in}}^{\text{eager}}(n), \tag{29}$$

$$\overline{\mathsf{RES}_{\text{in}}^{\text{lazy}}(n)}. \tag{30}$$

Claim:

$$\mathsf{GIVEN}^{\text{eager}}(n), \tag{31}$$

$$\overline{\mathsf{GIVEN}^{\text{lazy}}(n)}, \tag{32}$$

$$\mathsf{TAKEN}_{\text{in}}(n). \tag{33}$$

Proof:

(Read the following as: "Fact (29) implies by rule (14) that (31) holds, and that (34), *i.e.*, $\overline{\mathsf{GIVEN}_{in}^{eager}(n)}$, holds as well. Fact (34) implies by rule (25) ...")

$$(29) \xrightarrow{(14)} (31),\ \overline{\mathsf{GIVEN}_{in}^{eager}(n),} \tag{34}$$

$$(34) \xrightarrow{(25)} \overline{\mathsf{GIVEN}_{in}^{lazy}(n)}. \tag{35}$$

From (31) and (34) follows (33) by (12). (32) can be derived from (35) and (30) via (14).

□

Lemma 3 (Production at exit.) Given: *A node $n \in N$ such that*

$$\mathsf{RES}_{\text{out}}^{\text{eager}}(n), \tag{36}$$

$$\overline{\mathsf{RES}_{\text{out}}^{\text{lazy}}(n)}. \tag{37}$$

Claim: *For all $s \in Succs(n)$:*

$$\mathsf{GIVEN}_{\text{in}}^{\text{eager}}(s), \tag{38}$$

$$\overline{\mathsf{GIVEN}_{\text{in}}^{\text{lazy}}(s)}, \tag{39}$$

$$\mathsf{TAKEN}_{\text{in}}(s). \tag{40}$$

Proof:

$$(36) \xrightarrow{(15)} \overline{\mathsf{GIVEN}^{eager}_{out}(n)}, \tag{41}$$

$$\exists t \in \textsc{Succs}(n) : \mathsf{GIVEN}^{eager}_{in}(t), \tag{42}$$

$$(41) \xrightarrow{(27)} \overline{\mathsf{GIVEN}^{lazy}_{out}(n)}. \tag{43}$$

Note that (42) implies that $\textsc{Succs}(n) \neq \emptyset$, and therefore n cannot have an outgoing back edge (see Section 2.2.3). This implies that any path originating in n has to go through some $s \in Succs(n)$, which makes this lemma particularly useful. Furthermore, (41), (42) imply by Equation (11) that t has multiple predecessors. Since we exclude critical edges in E (see Section 2.2.2), t must be the only successor of n. Therefore, (42) is equivalent to (38). (37), (43) imply (39) by (15), and (40) follows from (38) and (41) using (11).
□

A.1.3 Induction step lemmata

Lemma 4 (Balancedness entering a node.) **Given:** *A node $n \in N$ such that*

$$\mathsf{GIVEN}^{\mathrm{eager}}_{\mathrm{in}}(n), \tag{44}$$

$$\overline{\mathsf{GIVEN}^{\mathrm{lazy}}_{\mathrm{in}}(n)}, \tag{45}$$

$$\mathsf{TAKEN}_{\mathrm{in}}(n). \tag{46}$$

Claim:

$$\overline{\mathsf{RES}^{\mathrm{eager}}_{\mathrm{in}}(n)}, \tag{47}$$

$$\mathsf{GIVEN}^{\mathrm{eager}}(n), \tag{48}$$

and either

$$\mathsf{RES}^{\mathrm{lazy}}_{\mathrm{in}}(n) \tag{49}$$

or

$$\overline{\mathsf{GIVEN}^{\mathrm{lazy}}(n)}, \tag{50}$$

$$\mathsf{TAKEN}_{\mathrm{out}}(n), \tag{51}$$

$$\overline{\mathsf{TAKE}(n)}. \tag{52}$$

Proof:
(44) implies (47) by (14) and (48) by (12). Assume (49) does not hold. This together with (45) implies by (14) that (50) must hold, which in turn implies (52) by (12). (51) follows by (6) from (46) and (52).
□

Lemma 5 (Balancedness within a node and its interval.) **Given:** *A node $n \in N$ such that*

$$\mathsf{GIVEN}^{\mathrm{eager}}(n), \tag{53}$$

$$\overline{\mathsf{GIVEN}^{\mathrm{lazy}}(n)}, \tag{54}$$

$$\mathsf{TAKEN}_{\mathrm{in}}(n). \tag{55}$$

Claim:

$$\overline{\mathsf{RES}^{\mathrm{eager}}_{\mathrm{out}}(n)}, \tag{56}$$

$$\mathsf{GIVEN}^{\mathrm{eager}}_{\mathrm{out}}(n), \tag{57}$$

$$\overline{\mathsf{GIVEN}^{\mathrm{lazy}}_{\mathrm{out}}(n)}, \tag{58}$$

$$\mathsf{TAKEN}_{\mathrm{out}}(n). \tag{59}$$

Furthermore, for all $m \in T(n)$:

$$\mathsf{GIVEN}^{\mathrm{eager}}(n), \tag{60}$$

$$\overline{\mathsf{GIVEN}^{\mathrm{lazy}}(n)}, \tag{61}$$

$$\overline{\mathsf{RES}^{\text{eager}}_{\text{in}}(m)}, \tag{62}$$

$$\overline{\mathsf{RES}^{\text{lazy}}_{\text{in}}(m)}, \tag{63}$$

$$\overline{\mathsf{RES}^{\text{eager}}_{\text{out}}(m)}, \tag{64}$$

$$\overline{\mathsf{RES}^{\text{lazy}}_{\text{out}}(m)}. \tag{65}$$

Proof:

We first prove the claims for n itself (claims (56) ... (59)). Then we will show that there is no consumption within $T(n)$. From there we will derive that EAGER and LAZY availability stay unchanged throughout $T(n)$ (claims (60), (61)). Proving that there will be no production of either type within $T(n)$ (claims (62) ... (65)) concludes the proof.

$$(54) \xrightarrow{(12)} \overline{\mathsf{TAKE}(n)}, \tag{66}$$

$$(55),(66) \xrightarrow{(6)} (59), \overline{\mathsf{BLOCK}(n)}, \tag{67}$$

$$(67) \xrightarrow{(3)} \overline{\mathsf{STEAL}(n)}, \tag{68}$$

$$\overline{\mathsf{GIVE}(n)}. \tag{69}$$

(58) follows by (13) from (54) and (69). (53),(68) imply via (13) that (57) holds, which in turn implies (56) by (15).

Having proven the claims for n itself, we will inductively show for each node in $T(n)$ that EAGER and LAZY availability stay unchanged (claims (60), (61)). **Induction base**: Let $m = \text{FIRSTCHILD}(n)$. From $n = \text{HEADER}(m)$, it follows for m:

$$(67) \xrightarrow{(3)} \overline{\mathsf{BLOCK}_{loc}(m)}, \tag{70}$$

$$(59),(66),(67) \xrightarrow{(5)} \overline{\mathsf{TAKE}_{loc}(m)}. \tag{71}$$

Induction step: Assume that (70), (71) hold for some arbitrary $m \in N$. We can then also prove for m:

$$(71) \xrightarrow{(8)} \overline{\mathsf{TAKE}(m)}, \tag{72}$$

$$(72),(70) \xrightarrow{(7)} \overline{\mathsf{BLOCK}(m)}, \tag{73}$$

$$(73) \xrightarrow{(3)} \overline{\mathsf{GIVE}(m)}, \tag{74}$$

$$\overline{\mathsf{STEAL}(m)}. \tag{75}$$

Let $s \in \text{SUCCS}^-(m)$ (if there are any such successors), let $c = \text{FIRSTCHILD}(m)$ (if it exists). From (73) follows (70) for c by definition (3). (70) for s follows by definition (7) from (70),(72). (71) for both c and s follows from (71) and (73) by Definition (8).

Let $l = \text{LASTCHILD}(n)$. It follows:

$$\text{SUCCS}^+(l) = \emptyset \xrightarrow{(4)} \overline{\mathsf{TAKEN}_{out}(l)} \tag{76}$$

$$(72),(76) \xrightarrow{(6)} \overline{\mathsf{TAKEN}_{in}(l)}. \tag{77}$$

Let $m \in \text{PREDS}(l) \cap T(n)$. This implies $l \in \text{SUCCS}^+(m)$, and from (77) then follows by Equation (4) that (76) must hold for m as well. This together with (72) implies in turn (77) for m. In this manner we can prove inductively that (76), (77) hold for all $t \in T(n)$, *i.e.*, there is no consumption within $T(n)$.

We proceed to prove inductively that EAGER and LAZY availability stay unchanged throughout $T(n)$ (claims (60), (61)). **Induction base**: Let $m = \text{FIRSTCHILD}(n)$.

$$(53) \xrightarrow{(11)} \mathsf{GIVEN}^{eager}_{in}(m), \tag{78}$$

$$(54) \xrightarrow{(11)} \overline{\mathsf{GIVEN}^{lazy}_{in}(m)}. \tag{79}$$

Induction step for (60): Assume that (78) holds for some arbitrary $m \in T(n)$. Equation (12) implies (60) for m. It follows:

$$(60), (75) \xrightarrow{(13)} \mathsf{GIVEN}_{out}^{eager}(m). \tag{80}$$

Let $s \cap T(n)$ such that (80) holds for all $m \in \textsc{Preds}(s)$. Equation (11) then implies (78) for s, which concludes the induction. **Induction step for (61):** Assume that (79) holds for some arbitrary $m \in T(n)$. (72) then implies via Equation (12) that (61) holds for m. It follows:

$$(61), (74) \xrightarrow{(13)} \overline{\mathsf{GIVEN}_{out}^{lazy}(m)}. \tag{81}$$

As for (80), we can assume to have inductively proven (81) for all $m \in \textsc{Preds}(s)$. Equation (11) then implies (79) for s, which concludes the induction.

It remains to prove that there is no production within $T(n)$) (claims (62) ... (65)). Definition (14) implies (62) from (78), and (63) from (61). From (80) follows (64) via (15). Claim (65) is slightly more complicated, since we have to prove (79) to hold for all $s \in \textsc{Succs}(m)$, including successors connected to m through an exit edge. In other words, we must ensure that exit edges do not push production back into $T(n)$. Let $e = (m, s)$ be an exit edge with $m \in T(n), s \notin T(n)$. From the lack of critical edges follows $\textsc{Preds}(s) = \{m\}$ (see Section 2.2.3). (81) for m then implies by Equation (11) that (79) holds for s. Since (79) was already proven to hold for all $s \in T(n)$, we now know (79) to hold for all $s \in \textsc{Succs}(m)$, from which (65) follows by Equation (15).

□

Lemma 6 (Balancedness along forward/exit edges.) Given: *Nodes* $p, n \in N$ *such that* $p \in \textsc{Preds}(n)$ *and*

$$\mathsf{GIVEN}_{\text{out}}^{\text{eager}}(p), \tag{82}$$

$$\overline{\mathsf{GIVEN}_{\text{out}}^{\text{lazy}}(p)}, \tag{83}$$

$$\mathsf{TAKEN}_{\text{out}}(p), \tag{84}$$

$$\overline{\mathsf{RES}_{\text{out}}^{\text{lazy}}(p)}. \tag{85}$$

Claim:

$$\mathsf{GIVEN}_{\text{in}}^{\text{eager}}(n), \tag{86}$$

$$\overline{\mathsf{GIVEN}_{\text{in}}^{\text{lazy}}(n)}, \tag{87}$$

$$\mathsf{TAKEN}_{\text{in}}(n). \tag{88}$$

Proof:
(88) follows from (84) by (4). (82) and (88) imply (86) by (11). (83), (85) imply (87) by (15).
□

A.1.4 Proof of the Balancedness Theorem

Based on the lemmata developed so far, we now inductively prove Theorem 1 (as mentioned before, we will only show the Eager, Before case, the other cases are analogous).

Induction base.

$\mathsf{RES}_{r_e}^{lazy}(q_e)$ would correspond to (23) for $l = e$. (24) would be vacuously true, and we would be done. Let $n = q_e$. Suppose

$$\overline{\mathsf{RES}_{r_e}^{lazy}(n)}. \tag{89}$$

The following is the induction invariant that the induction base will prove to hold:

$$\mathsf{GIVEN}^{eager}(n), \tag{90}$$

$$\overline{\mathsf{GIVEN}^{lazy}(n)}, \tag{91}$$

$$\mathsf{TAKEN}_{in}(n). \tag{92}$$

(The induction step will then prove that the invariant for some q_i implies that either the invariant holds for q_{i+1} as well, or that (89) cannot hold for q_{i+1}, in which case we would be done.) We have to differentiate between production at entry and production at exit of n. If $r_e = in$, then the invariant follows directly from (22),(89) through Lemma 2. Suppose $r_e = out$; let $s = q_{e+1}$. We cannot prove the invariant for n itself ((22) actually contradicts (90) via (15) and (13)), but we will show that the invariant to hold for s. (22) implies by (15) that $\textsc{Succs}(n) \neq \emptyset$, therefore e cannot be a back edge (see Section 2.2.3). It follows $s \in \textsc{Succs}(n)$. (22),(89) then imply through Lemma 3 that (38), (39), (40) hold for s. This implies (92) ($\equiv$ (40)) and also makes Lemma 4 applicable ((38)$\equiv$(44), (39)$\equiv$(45), (40)$\equiv$(46)). Invariant (90) then corresponds to (48). Assumption (89) contradicts (49), so the last invariant, (91), follows from (50).

Our induction has not only to prove that (23) will eventually come true for some q_l, $e \leq l < s$, (*i.e.*, that (89) will fail for q_l), but it also has to prove that (24) holds for all $q_k \in N, e < k \leq l$. For $r_e = in$, the induction base corresponds to $e = l$ (since we proved the invariant for q_e itself), and (24) is vacuously true again. For $r_e = out$, we have to prove (24) for $s = q_{e+1}$. This, however, is equivalent to result (47) of Lemma 4 whose prerequisites were already fulfilled.

Induction step.

Let $q_k \in N$, $e < k$; let $m = q_k$, $n = q_{k+1}$. Assume (89) ... (92) to hold for m. We want to prove that either (89) does not hold for n, or that (90) ... (92) do hold for n. We also have to show (24) for n. We perform the induction along the edge $(m, r_k) \rightarrow (n, r_{k+1})$, based on the actual type of edge.

Case 1 (internal edge): $r_k = in$, $r_{k+1} = out$. Since this is an internal edge, it is $m = n$, and the invariant to prove is already part of the induction step assumptions. From the invariant also follows via Lemma 5 that (24) ($\equiv$ (56)) holds for n. □

Case 2 (forward/exit edge): $r_k = out$, $r_{k+1} = in$. We can apply Lemma 5 for m ((53)$\equiv$(90), (54)$\equiv$(91), (55)$\equiv$(92)). This lemma, together with (89), implies the prerequisites for Lemma 6 ((57)$\equiv$(82), (58) $\equiv$(83), (59)$\equiv$(84), (89)$\equiv$(85)). Lemma 6 in turn makes Lemma 4 applicable ((86)$\equiv$(44), (87)$\equiv$(45), (88)$\equiv$(46)). As in the induction base, it follows that for n either (89) does not hold or (90), (91), (92) hold; (24) ($\equiv$(47)) follows as well. □

Case 3 (entry edge): $r_k = r_{k+1} = in$. We can apply Lemma 5 for m, which then states in (62) ... (65) that there will be no production anywhere within $T(m)$ (implying (24) for all nodes in $T(m)$). Therefore, we will perform an induction step that leads directly to the first q_f, $f > k$, such that $q_f \in T(m)$, $q_{f+1} \notin T(m)$. Let $p = q_f$, $s = q_{f+1}$. We want to prove the invariant (90), (91), (92) for s. Let $g = (p, s)$; g can be either a back edge or an exit edge.

If g is a back edge, then we are exiting $T(m)$ as we entered it (*i.e.*, through it's header, m). Since in this case the induction invariant is already proven for s ($= m$), we are done and can continue with Case 1 (with $f = k$).

If g is an exit edge, it is $s \in \textsc{Succs}^+(m)$, $r_f = out$, $r_{f+1} = in$. Lemma 5 implies in (59) via (4) that (92) holds for s. From the proof of Lemma 5, we know that (79) holds for s. The same proof states that (80) holds for p, which together with (92) implies via (11) that (78) holds for s. Now we can apply Lemma 4 for s ((78)$\equiv$(44), (79)$\equiv$(45), (92)$\equiv$(46)), and we are done. □

Case 4 (back edge): $r_k = r_{k+1} = out$. In this case, m is the last child of an interval. Since we do not allow critical edges, it is $\textsc{Succs}^+(m) = \emptyset$. From Equation (4) follows that (59) does not hold for n, which together with (92) implies via (6) that (66) does not hold. This, however, is by (12) a contradiction with (91), therefore we do not have to consider this case further. Note the implication that after an EAGER placement has occurred along p within some loop, we can never delay LAZY production past the exit of the loop. □

To conclude the proof of the theorem, we notice that since $\mathsf{TAKEN}_{out}(\textsc{Stop}) = \bot$, the induction invariant (92) eventually leads to a contradiction along p; therefore, (89) cannot hold for all nodes visited after q_e, and (22) becomes true.

□

A.2 Safety

Safety is guaranteed if each production is succeeded by a consumption specified in the initial input for the framework. This is equivalent to the following theorem:

Theorem 2 (C2: Safety.) Given: *An expanded path*

$$p : ((\textsc{Start}, \text{out}) =) \ (q_0, r_0) \rightarrow (q_1, r_1) \rightarrow \ldots \rightarrow (q_s, r_s) \ (= (\textsc{Stop}, \text{in}))$$

such that for some p, $0 < p < s$,

$$\mathsf{RES}_{r_p}. \tag{93}$$

Claim: *there exists in a* BEFORE *solution an* $q_c \in N$, $0 \leq p \leq c$, *such that* $r_c = \text{in}$ *and*

$$\mathsf{TAKE}_{\text{init}}(q_c). \tag{94}$$

In an AFTER *solution, the flow of control is reversed.*

Proof:
Let $m = q_p$. For $r_p = in$, it is

$$\begin{array}{rcll} (93) & \xrightarrow{(14)} & \mathsf{GIVEN}(m), & (95) \\ & \text{and} & \overline{\mathsf{GIVEN}_{in}(m)}, & (96) \\ (95),(96) & \xrightarrow{(12)} & \mathsf{TAKEN}_{in}(n) \qquad \text{for an EAGER Problem,} & (97) \\ & \text{or} & \mathsf{TAKE}(n) \qquad \text{for a LAZY Problem.} & (98) \end{array}$$

The proof of the theorem follows by a straightforward induction from (97) and (98) using Equations (4), (5), (6), and (8).
□

A.3 Sufficiency

A.3.1 The Sufficiency Theorem

Sufficiency requires that each consumption is proceeded by production, without being destroyed before reaching the consumption. Furthermore, there has to be both an EAGER and a LAZY production, in this order. This is implied by the following theorem, which also reflects that EAGER production only occurs on node entries (and not on exits).

Theorem 3 (C3: Sufficiency.) Given: *An expanded path*

$$p : ((\text{START}, \text{out}) =) \ (q_0, r_0) \rightarrow (q_1, r_1) \rightarrow \ldots \rightarrow (q_s, r_s) \ (= (\text{STOP}, \text{in}))$$

such that for some c, $0 < c < s$, $r_c = \text{in}$,

$$\mathsf{TAKE}_{\text{init}}(q_c). \tag{99}$$

Claim: *There exists in a* BEFORE *solution a* q_e, $0 \leq e < c$, *such that* $r_e = \text{in}$ *and*

$$\mathsf{GIVE}_{\text{init}}(q_e). \tag{100}$$

or

$$\mathsf{RES}_{\text{in}}^{\text{eager}}(q_e) \tag{101}$$

In the latter case, there also exists an l, $e \leq l \leq c$, *such that*

$$\mathsf{RES}_{r_l}^{\text{lazy}}(q_l). \tag{102}$$

Furthermore, for all i *with* $r_i = \text{out}$, $e \leq i \leq c$, *it is*

$$\overline{\mathsf{STEAL}_{\text{init}}(q_i)}. \tag{103}$$

In an AFTER *solution, the flow of control is reversed.*

Again, we will show the proof for a BEFORE problem (in our communication generation application, this corresponds to guaranteeing that each reference will be proceeded by a local definition or a READ$_{Send}$ and a READ$_{Recv}$, without any non-local definition in between). The proof for an AFTER problem is analogous.

The basic idea of the proof is to backtrace on p from q_c until we reach a producer. While walking backwards, we keep the invariant GIVEN, and we also ensure $\overline{\mathsf{STEAL}}$ holds along all internal edges crossed. The following contains some lemmata on which an inductive proof of Theorem 3 will be based.

A.3.2 Induction step lemmata

Lemma 7 (Local availability implies global availability.) Claim: *For all $n \in N$, the following holds.*

$$\mathsf{GIVEN}_{\mathrm{out}}(n) \supseteq \mathsf{GIVE}_{\mathrm{loc}}(n). \tag{104}$$

Proof:
If $n = \textsc{Root}$, then $\mathsf{GIVEN}_{out}(\textsc{Root}) = \mathsf{GIVE}_{loc}(\textsc{Root}) = \bot$, and we are done.
For $n \neq \textsc{Root}$, it is

$$\begin{aligned} \mathsf{GIVEN}_{out}(n) &\overset{(13)}{=} (\mathsf{GIVE}(n) \cup \mathsf{GIVEN}(n)) \setminus \mathsf{STEAL}(n) \\ &\overset{(12),(28)}{\supseteq} (\mathsf{GIVE}(n) \cup \mathsf{GIVEN}_{in}(m) \cup \mathsf{TAKE}(n)) \setminus \mathsf{STEAL}(n) \end{aligned} \tag{105}$$

If $n = \textsc{FirstChild}(m)$, it is

$$\textsc{Preds}(n) = \emptyset, \tag{106}$$

$$\begin{aligned} \mathsf{GIVEN}_{out}(n) &\overset{(105)}{\supseteq} (\mathsf{GIVE}(n) \cup \mathsf{TAKE}(n)) \setminus \mathsf{STEAL}(n) \\ &\overset{(9),(106)}{=} \mathsf{GIVE}_{loc}(n). \end{aligned}$$

Otherwise, assume (104) to hold for all $p \in \textsc{Preds}(n)$. It follows inductively:

$$\begin{aligned} \mathsf{GIVEN}_{out}(n) &\overset{(105),(11)}{\supseteq} (\mathsf{GIVE}(n) \cup \bigcap_{p\in\textsc{Preds}(n)} \mathsf{GIVEN}_{out}(p) \cup \mathsf{TAKE}(n)) \setminus \mathsf{STEAL}(n) \\ &\overset{by\ ind.}{\supseteq} (\mathsf{GIVE}(n) \cup \mathsf{TAKE} \cup \bigcap_{p\in\textsc{Preds}(n)} \mathsf{GIVE}_{loc}(p)) \setminus \mathsf{STEAL}(n). \end{aligned}$$

□

Lemma 8 (Items are either propagated or stolen locally.) Given: *A node $n \in N$ such that for all $p \in \textsc{Preds}(n)$:*

$$\mathsf{STEAL}_{\mathrm{loc}}(p) \tag{107}$$
$$or\quad \mathsf{GIVEN}_{\mathrm{out}}(p). \tag{108}$$

Claim:

$$\mathsf{STEAL}_{\mathrm{loc}}(n), \tag{109}$$
$$or\quad \mathsf{GIVEN}_{\mathrm{out}}(n) \tag{110}$$
$$and\quad \mathsf{GIVEN}(n). \tag{111}$$

Proof:
Assume

$$\overline{\mathsf{STEAL}_{loc}(n)}. \tag{112}$$

$$(112) \xrightarrow{(10)} \overline{\mathsf{STEAL}(n)} \tag{113}$$

Case 1: Assume (108) holds for all $p \in \textsc{Preds}(n)$.

$$(108) \xrightarrow{(11)} \mathsf{GIVEN}_{in}(n). \tag{114}$$

(111) follows from (114) via (12), and (111), (113) together result by (13) in (110). □
Case 2: Assume $\exists p \in \textsc{Preds}(n)$ such that (108) does not hold for p; this implies (107) for p.

$$(107), (112) \xrightarrow{(10)} \mathsf{GIVE}_{loc}(p). \tag{115}$$

This, however, implies (108) for p by Lemma 7, which is a contradiction.
□

Lemma 9 (Sufficiency throughout intervals.) Given: *A node $n \in N$ such that*

$$\mathsf{GIVEN}(n), \tag{116}$$

$$\mathsf{GIVEN}_{\mathrm{out}}(n). \tag{117}$$

Claim: *For $l = \textsc{LastChild}(n)$,*

$$\mathsf{GIVEN}_{\mathrm{out}}(l), \tag{118}$$

and for all $c \in \textsc{Children}(n)$:

$$\mathsf{STEAL}_{\mathrm{loc}}(c), \tag{119}$$

$$\textit{or both} \quad \mathsf{GIVEN}(c) \tag{120}$$

$$\textit{and} \quad \mathsf{GIVEN}_{\mathrm{out}}(c). \tag{121}$$

Proof:

The main result of this lemma that we are interested in is (118). However, we first prove inductively that (119) or both (120) and (121) hold for all children. Furthermore, this induction might lead us to deeper nesting levels of $T(n)$ than just the children of n.

Let $f = \textsc{FirstChild}(n)$.

$$(116) \xrightarrow{(11)} \mathsf{GIVEN}_{in}(f), \tag{122}$$

$$(122) \xrightarrow{(12)} \mathsf{GIVEN}(f) \qquad (\equiv(120) \text{ for } f), \tag{123}$$

$$(123) \xrightarrow{(13)} \mathsf{GIVEN}_{out}(f) \qquad (\equiv(121) \text{ for } f), \tag{124}$$

$$\textit{or} \quad \mathsf{STEAL}(f), \tag{125}$$

$$(125) \xrightarrow{(10)} \mathsf{STEAL}_{loc}(f) \qquad (\equiv(119) \text{ for } f). \tag{126}$$

If for all $c \in \textsc{Children}(n)$, $\textsc{Preds}(c) \subseteq \textsc{Children}(n)$ (*i.e.*, no exit edge from any loop nested within n is reaching any child of n), then we can apply Lemma 8 as induction step to prove for all $c \in \textsc{Children}(n)$ that (119) or both (121) and (120) hold.

Otherwise, let c be the first (leftmost) child of n such that there exists an exit edge $e = (p, c)$, and let $h \in \textsc{Children}(n)$ such that $p \in T(h)$ (*i.e.*, h is the header of the outermost loop exited by e). Note that since we do not allow critical edges,

$$h \in \textsc{Preds}^{+}(c) \setminus \textsc{Preds}(c). \tag{127}$$

Since c was the first child of n reached by an exit edge, we can prove inductively by Lemma 8 that (119) or both (120) and (121) hold for all ancestors of c. including h.

If (119) holds for h, then (127) implies (119) for c via (10). Assume (119) does not hold for h. Then (120) and (121) hold for h, and we can apply Lemma 9 for h. In this manner, we can perform an induction over the nesting level; due to the finite nesting depth of $T(n)$, (119) has eventually to hold for some header and we are done. □

This concludes the induction for (119) or both (121) and (120). We also have:

$$(117) \xrightarrow{(13)} \overline{\mathsf{STEAL}(n)} \tag{128}$$

$$(128) \xrightarrow{(1)} \overline{\mathsf{STEAL}_{loc}(l)}. \tag{129}$$

In other words, (119) does not hold for l, which implies (118) for l ($\equiv$ (121)) and concludes the proof of the lemma.

□

A.3.3 Proof of the Sufficiency Theorem

Based on the lemmata developed so far, we now prove Theorem 3 (as mentioned before, we will only show the Eager, Before case, the other cases are analogous).

Induction base. We will prove that for both flavors there exists a producer q_p reaching consumer q_c. Balancedness then implies that they are actually in the right order.

Let $n = q_c$.

$$(99) \xrightarrow{(5)} \mathsf{TAKE}(n), \tag{130}$$
$$(130) \xrightarrow{(6)} \mathsf{TAKEN}_{in}(n). \tag{131}$$

This implies for both EAGER and LAZY:

$$(130),(131) \xrightarrow{(12)} \mathsf{GIVEN}(n) \tag{132}$$
$$(132) \xrightarrow{(14)} \mathsf{RES}_{in}(n) \tag{133}$$
$$\text{or} \quad \mathsf{GIVEN}_{in}(n). \tag{134}$$

If (133) were true, then we would have reached production and would be done. Assume (134); this will be the induction invariant we will use for node entries.

Induction step.

Let $q_i \in N$, $i < c$; let $m = q_{i-1}$, $n = q_i$. We perform the induction along the edge $(m, r_{i-1}) \rightarrow (n, r_i)$, based on the actual type of edge. We have to prove for m that we either reach production ((133) at entry, or (135) at exit), or that availability is preserved (the invariant, (134) at entry and (136) at exit). The exit invariant (136) then implies via (13) and (1) that nothing is stolen ((103) at exit). A special case arises when traversing internal edges of interval headers, which corresponds to skipping the interval (*e.g.*, not executing a loop; see Section 2.2.1).

Case 1 (forward/exit edge): $r_{i-1} = out$, $r_i = in$. From the invariant (134) for n follows for m directly from (15) either

$$\mathsf{RES}_{out}(m) \tag{135}$$

or the invariant

$$\mathsf{GIVEN}_{out}(m). \tag{136}$$

□

Case 2 (entry edge): $r_{i-1} = r_i = in$. Since in this case $m = \text{HEADER}(n)$, it follows from the invariant (134) for n via (11) that (132) must hold for m. We can proceed as in the induction base to prove that at m we either reached production or preserve the invariant. □

Case 3 (internal edge): $r_{i-1} = in$, $r_i = out$. Since this is an internal edge, it is $m = n$. The invariant (136) for n implies by (13) that either (132) must hold for n, in which case we could proceed as in Case 2 and would be done, or that the following holds:

$$\mathsf{GIVE}(n), \tag{137}$$
$$(137) \xrightarrow{(2)} \mathsf{GIVE}_{init}(n), \tag{138}$$
$$\text{or} \quad \mathsf{GIVE}_{loc}(\text{LASTCHILD}(n)). \tag{139}$$

(138) would be equivalent to (100), and we would be done. If (138) does not hold, then we have encountered the special case mentioned above, where data are produced in $T(n)$ and not in n itself or a predecessor of n. As described in Section 2.2.1, $\mathsf{GIVE}(n) \setminus \mathsf{GIVE}_{init}(n)$ are the data for which we rely on them being produced in the loop, or being produced in a separate node that gets executed if the loop does not get executed. □

Case 4 (back edge): $r_{i-1} = r_i = out$. As in Case 3, (138) or (139) must hold. If (138) were true, we would be done. Otherwise, (139) would imply for m ($=$ LASTCHILD(n)) by (104) that the invariant (136) for m, and we would be done in this case as well. □

This completes the proof of the Sufficiency Theorem.

B Proof of correctness of the algorithm

After proving in Appendix A the correctness of the equations form Section 2.3, this section proves the correctness of the *GiveNTake* algorithm presented in Section 2.4 by demonstrating that it computes a fixed point for these equations. We are guaranteed to reach a fixed point if each equation gets evaluated after its right hand side is fully known.

For the discussion of how the equations are linked to each other, we define the $E(e, n)$ as a shorthand the variable defined by Equation e for node n; $E(e_1, n_1) \longleftarrow E(e_2, n_2)$ then expresses that $E(e_1, n_1)$ depends on

$E(e_2, n_2)$. We let $S_i(n)$ denote the evaluation of the equations in set S_i for node n. We assume that each set is evaluated in increasing equation number. Therefore, constraints of the form $E(e_1, n) \longleftarrow E(e_2, n)$ are satisfied for all e_1, e_2 from the same set with $e_1 > e_2$ and will be omitted from further discussion. For example, the dependence of $\mathsf{GIVE}(n)$ on $\mathsf{STEAL}(N)$ (*i.e.*, $E(2, n) \longleftarrow E(1, n)$) is already satisfied under this assumption.

S_1 depends on itself as follows:

- $E(4, n) \longleftarrow E(6, \textsc{Succs}(n))$; $E(7, n) \longleftarrow E(7, \textsc{Succs}(n))$; $E(8, n) \longleftarrow E(8, \textsc{Succs}(n))$.

  This implies that S_1 should be evaluated in BACKWARD order to make sure all the successors of n are processed before n itself.

- $E(3, n) \longleftarrow E(7, \textsc{FirstChild}(n))$; $E(5, n) \longleftarrow E(6, \textsc{FirstChild}(n))$, $E(8, \textsc{FirstChild}(n))$.

  S_1 should be evaluated in an UPWARD fashion to make sure that each header can access the values of its children.

The above constraints alone could be satisfied by evaluating S_1 in REVERSEPREORDER. However, S_1 also depends on S_2 as follows:

- $E(1, n) \longleftarrow E(10, \textsc{LastChild}(n))$; $E(2, n) \longleftarrow E(9, \textsc{LastChild}(n))$.

  This can be satisfied by evaluating $S_1(n)$ after $S_2(\textsc{Children}(n))$.

Furthermore, S_2 depends on S_1:

- $E(9, n) \longleftarrow E(1, n)$, $E(2, n)$, $E(5, n)$. $E(10, n) \longleftarrow E(1, n)$, $E(2, n)$, $E(5, n)$;

  $S_2(n)$ should be computed after $S_1(n)$.

Finally, S_2 depends on itself:

- $E(9, n) \longleftarrow E(9, \textsc{Preds}(n))$. $E(10, n) \longleftarrow E(10, \textsc{Preds}(n))$;

  S_2 should be evaluated in FORWARD order.

The dependencies for S_3 and S_4 are somewhat simpler:

- $E(11, n) \longleftarrow E(6, n)$, $E(12, \textsc{Header}(n))$, $E(13, \textsc{Preds}(n))$; $E(12, n) \longleftarrow E(6, n)$, $E(5, n)$; $E(13, n) \longleftarrow E(2, n)$, $E(1, n)$.

  These constraints are satisfied when evaluating S_3 in FORWARD, DOWNWARD fashion (*i.e.*, PREORDER) after S_1.

- $E(14, n) \longleftarrow E(6, n)$, $E(12, n)$, $E(11, n)$; $E(15, n) \longleftarrow E(4, n)$, $E(11, n)$, $E(11, \textsc{Succs}(n))$, $E(13, n)$.

  S_4 has to be evaluated after S_1 and S_3, in any order.

It can easily be verified that the algorithm in Figure 9 observes all the constraints.