GIVE-N-TAKE — A Balanced Code Placement Framework

Reinhard v. Hanxleden
Ken Kennedy

CRPC-TR94388-S
March, 1994

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

From the *Proceedings of the ACM SIGPLAN '94 Conference on Program Language Design and Implementation*, Orlando, Florida, June 1994.

GIVE-N-TAKE — A Balanced Code Placement Framework*†

Reinhard von Hanxleden Ken Kennedy

Center for Research on Parallel Computation
Department of Computer Science, Rice University
P.O. Box 1892, Houston, TX 77251
E-mail: {reinhard|ken}@rice.edu

Abstract

GIVE-N-TAKE is a code placement framework which uses a general producer-consumer concept. An advantage of GIVE-N-TAKE over existing partial redundancy elimination techniques is its concept of production *regions*, instead of single locations, which can be beneficial for general latency hiding. GIVE-N-TAKE guarantees *balanced* production, that is, each production will be started and stopped once. The framework can also take advantage of production coming "for free," as induced by side effects, without disturbing balance. GIVE-N-TAKE can place production either before or after consumption, and it also provides the option to hoist code out of potentially zero-trip loop (nest) constructs. GIVE-N-TAKE uses a fast elimination method based on Tarjan intervals, with a complexity linear in the program size in most cases.

We have implemented GIVE-N-TAKE as part of a Fortran D compiler prototype, where it solves various communication generation problems associated with compiling data-parallel languages onto distributed-memory architectures.

1 Introduction

Partial Redundancy Elimination (PRE) is a classical optimization framework for moving and placing code in a program. Example applications include common subexpression elimination, loop invariant code motion, and strength reduction. The original dataflow framework for performing PRE was developed by Morel and Renvoise [MR79] and has since then experienced various refinements [JD82, DS88, Dha88a, Dha91, DRZ92, KRS92]. However, the PRE frameworks developed to date still have certain limitations, which become apparent when trying to apply them to more complex code placement tasks.

Atomicity: PRE implicitly assumes that the code fragments it moves, generates, or modifies are atomic in that they need only a single location in the program to be executed. For example, when placing the computation of a common subexpression, PRE will specify only one location in the program, and code will be generated at that location to perform the entire computation. Later optimizations may then reschedule the individual instructions, for example to hide memory access delays, but PRE itself does not provide any such mechanism.

Ignoring side effects: Taking again the example of common subexpression elimination, classical PRE assumes that each common subexpression has to be computed somewhere; *i.e.*, nothing "comes for free." However, there are problems where side effects of other operations can eliminate the need for actual code placement. For example, when placing register loads and stores, certain loads may become redundant with previous definitions. This is generally treated as a special case, for example by developing different, but interdependent sets of equations for loads and stores [Dha88b].

Pessimistic loop handling: One difficulty with flow analysis has traditionally been the treatment of loop constructs that allow zero-trip instances, like a Fortran DO loop. Hoisting code out of such loops is generally considered *unsafe*, as it may introduce statements on paths where they have not existed before. However, unless the computation to be moved may change the meaning of the program (for example by introducing a division by zero), we often would like to hoist computation out of such loops even if the number of iterations is not known at compile time.

*This work is supported by an IBM fellowship, and by the National Aeronautics and Space Administration/the National Science Foundation under grant #ASC-9349459.

†From the *Proceedings of the ACM SIGPLAN '94 Conference on Program Language Design and Implementation*, Orlando, Florida, June 1994. Also available via anonymous ftp from `softlib.rice.edu` as `pub/CRPC-TRs/reports/CRPC-TR94388-S`.

Several techniques exist to handle zero-trip loops, 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 due to explicit control flow graph manipulations, and they do not fully apply to nested loops.

This paper presents a data flow framework, called GIVE-N-TAKE, that aims to overcome these limitations in a general context. It is applicable to a broad class of code generation/placement problems, including the classical domains of PRE techniques as well as memory hierarchy related problems, like prefetching and communication generation. GIVE-N-TAKE is subject to a set of correctness and optimality criteria (see Section 3.2); for example, each consumption *must* be preceded by a production, and 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 (*e.g.*, for fetching an operand), whereas in an AFTER problem, they have to be produced afterwards (*e.g.*, for storing a result). Intuitively, one can think of an AFTER problem as a BEFORE problem with reversed flow of control.

Orthogonally we can classify a problem as EAGER when it asks for production as early as possible (*e.g.*, sending a message), or as LAZY when it wants production as late as possible (*e.g.*, receiving a message); this definition assumes a BEFORE problem. For an AFTER problem, "early" and "late" have to be interchanged. (Classical PRE, for example, can be classified as a LAZY, BEFORE problem.) This means that the same framework can be used for different flavors of problems; there are no separate sets of equations for loads and stores [Dha88b], or for READs and WRITEs [GV91].

The rest of this paper is organized as follows. Section 2 introduces the communication generation problem, which will be used as an illustrating example application of GIVE-N-TAKE. Section 3 provides further intuition for the GIVE-N-TAKE framework and some background on the type of flow graph and neighbor relations used by the GIVE-N-TAKE equations. Section 4 states the actual equations and argues informally for their correctness and efficiency. Section 5 gives an efficient algorithm for solving the GIVE-N-TAKE equations. Section 6 concludes with a brief summary. A discussion of possible extensions, such as the combination of GIVE-N-TAKE with dependence analysis, and formal correctness proofs of GIVE-N-TAKE can be found elsewhere [HK93].

```
do i = 1, N
    y(i) = ...
enddo
if test then
    do j = 1, N
        z(j) = ...
    enddo
    do k = 1, N
        ... = x(a(k))
    enddo
else
    do l = 1, N
        ... = x(a(l))
    enddo
endif
```

Figure 1: An instance of the communication placement problem, where the array x is assumed to be distributed or shared. Each reference to x in the k and l loops necessitates a global READ operation, whereby a processor referencing some element of x receives it from its owner. Possible communication placements are shown in Figure 2.

2 A Code Placement Example: Communication Generation

An example of code placement is the generation of communication statements when compiling data parallel languages, like Fortran D [HKT92a] or HIGH PERFORMANCE FORTRAN [KLS+94]. For example, a processor of a distributed memory machine may reference owned data, which by default reside on the processor, as well as non-owned data, which reside on other processors. Local references to non-owned data induce a need for communication, in this case a READ of the referenced data from other processors. Figure 1 shows an example node code containing references to distributed data.

Since generating an individual message for each datum to be exchanged would be prohibitively expensive on most architectures, optimizations like message vectorization, latency hiding, and avoiding redundant communication are crucial for achieving acceptable performance [HKT92b]. The profitability of such optimizations depends heavily on the actual machine characteristics; however, even for machines with low latencies or shared-memory architectures, the performance can benefit from maximizing reuse and minimizing the total number of shared data accesses.

Figure 2 compares two possible communication placements for the example from Figure 1. Note that the GIVE-N-TAKE solution shown on the right would generally be considered unsafe, since for $N < 1$ the loops would not be executed. In the communication generation problem, however, we generally rather accept the risk of slight overcommunication than not

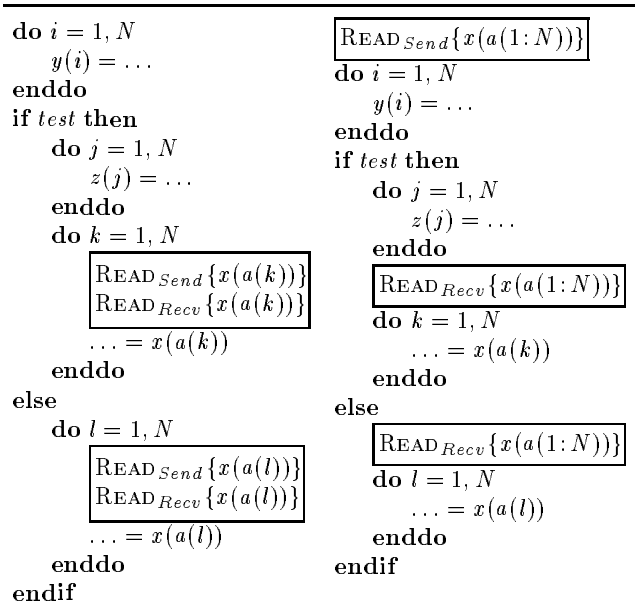

Figure 2: Possible communication placements for the code in Figure 1. A naïve code generation, shown on the left, results in a total of N messages to be exchanged, without any latency hiding. The solution provided by GIVE-N-TAKE, shown on the right, needs just one message and uses the i-loop for latency hiding ($x(a(k))$ and $x(a(l))$ can be recognized as identical based on the subscript value numbers).

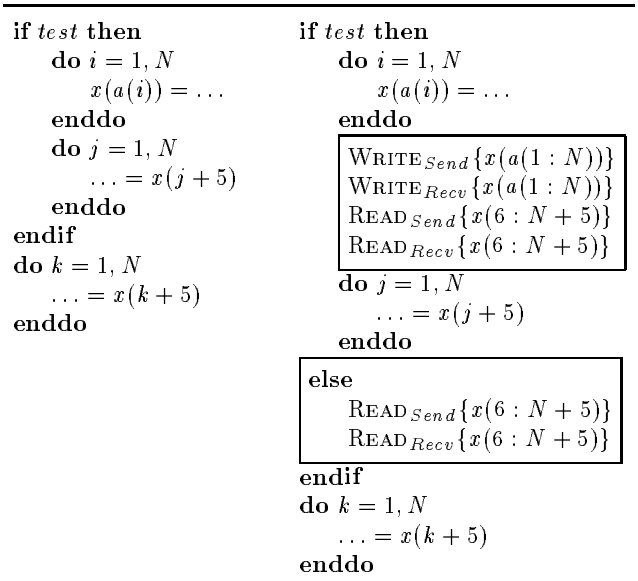

Figure 3: Example of a code with local definitions of potentially non-owned data (left), and a corresponding placement of global WRITEs (right).

hoist communication. Furthermore, it is often the case that non-execution of a loop also means that no communication needs to be performed (in the example, $N < 1$ implies $x(a(1:N)) = \emptyset$).

Note that the examples shown in this paper do not include data declarations, initializations, distribution statements, etc. The communication statements are in a high level format that does not include any schedule parameters, message tags, and so on. Communication schedule generation, which is a non-trivial problem in itself [HKK+92], and the conversion from global to local name space are also excluded. These and other implementation details on the usage of GIVE-N-TAKE for communication generation, like the value number based data flow universe, are described elsewhere [Han93].

If we do not use a strict owner computes rule [CK88], then non-owned data may not only be locally referenced, but also locally defined. We assume that these data have to be written back to their owners before they can be used by other processors, as shown in Figure 3. (An alternative would be the direct exchange between a non-owner that writes data and another non-owner that reads them [GS93]. This could also be accommodated by GIVE-N-TAKE, but especially in the presence of indirect references it would result in more complicated code generation.)

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. Therefore, combinations of dependence analysis and PRE have been used, for example for determining reaching definitions [GS90] or performing scalar replacement [CK92]. Duesterwald et al. incorporate iteration distance vectors (assuming regular array references) into an array reference data flow framework, which is then applied to memory optimizations and controlled loop unrolling [DGS93].

Several researchers have already addressed the communication generation problem, although often restricted to relatively simple array reference patterns. Amarasinghe and Lam optimize communication generation using Last Write Trees [AL93]. They assume affine loop bounds and array indices, they do not allow loops within conditionals (such as in Figure 1). 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. Granston and Veidenbaum combine dependence analysis and

PRE 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 non-owned variables if these variables have already been read or written locally. However, they assume atomicity, and they also assume that the program is already annotated with read/write operations; they do not try to hoist memory accesses to less frequently executed regions.

While these works address many important aspects of communication generation that are outside of the scope of GIVE-N-TAKE itself, such as name space mappings or regular section analysis, they do not seem to be general and powerful enough with respect to communication *placement*. In the following, it is this aspect that we will focus on.

3 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.

Data flow frameworks are commonly characterized by a pair $\langle L, F \rangle$, where L is a meet semilattice and F is a class of functions (see Marlowe and Ryder [MR90] for a discussion of these and other general aspects of data flow frameworks). Roughly speaking, L characterizes the solution space (or universe) of the framework, such as the set of common subexpressions or available constants, and their interrelationships. F contains functions that operate on L and compute the desired information about the program. Together with a flow graph (consisting of nodes, edges, and a root) and a mapping from graph nodes or edges to F, this framework constitutes a data flow problem, which can be solved to analyze and optimize a certain aspect of a specific program. However, since we view GIVE-N-TAKE as a fairly general code placement mechanism, this paper will focus mostly on F, the class of functions that we use to propagate information about consumption and production through a given program.

3.1 Communication Placement with Give-N-Take

The problem of generating READs can be interpreted as a BEFORE problem as follows:

- Each reference to non-owned data *consumes* these data.
- Each READ operation, 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 on another processor) of non-owned data *destroys* these data.

To split each READ into a READ_{Send} (the send issued at the owner) and a READ_{Recv} (the corresponding receive at the referencing processor), we need both the EAGER and the LAZY solution of the framework. We want to send as early as possible and receive as late as possible; since this is a BEFORE problem, the READ_{Send}'s will be given by the EAGER solution, and the READ_{Recv}'s will be the LAZY solution.

For placing global WRITEs, the non-owned definitions can be viewed as consumers, just as non-owned references, and we have to insert producers which in this case communicate data back 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 non-owned references can be modified to take advantage of non-owned definitions if they are later locally referenced; *i.e.*, non-owned definitions can also be viewed as statements that produce non-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 an AFTER problem) and a WRITE_{Recv} (the EAGER solution).

3.2 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 to be correct are the following (with their specific implications when applied to communication generation):

(C1) *Balance*: If we compute both the EAGER and the LAZY solution for a given problem, then these solutions have to match each other; see Figure 4. (For each executed READ_{Send}, exactly one matching READ_{Recv} will be executed, and vice versa; similarly for WRITE_{Send}'s and WRITE_{Recv}'s.)

(C2) *Safety*: Everything produced will be consumed; see Figure 5. (No unnecessary READs or WRITEs. In our specific case, this is more an optimization than a correctness issue.)

A special case are zero-trip loop constructs, like a Fortran DO loop. GIVE-N-TAKE tries to hoist

items out of such loops, unless explicitly told otherwise on a general [HK93] or case-by-case (Section 4.1) basis.

(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; see Figure 6. (All references to non-owned data must be locally satisfiable due to preceding READs or local definitions, without intervening non-local definitions, and all definitions of non-owned data must be brought back to their owners by WRITEs before being referenced non-locally or communicated by a READ.)

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

(O1) Nothing produced already (and not destroyed yet) will be produced again; see Figure 7. (Nothing will be recommunicated, unless it has been non-locally redefined.)

(O2) There are as few producers as possible; see Figure 8. (Communicate as little as possible.)

(O3) Things are produced as early as possible for EAGER, BEFORE and LAZY, AFTER problems; see Figure 9. (Send as early as possible.)

(O3′) Things are produced as late as possible for LAZY, BEFORE and EAGER, AFTER problems; see Figure 10. (Receive as late possible.)

Note that while the correctness criteria are treated as strict requirements that GIVE-N-TAKE must fulfill [HK93], the optimality criteria are viewed more as general guidelines (and are phrased correspondingly vague).

3.3 The Interval Flow Graph

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] and incremental analysis [Bur90]. 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, 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 off the loop. In that sense, Tarjan

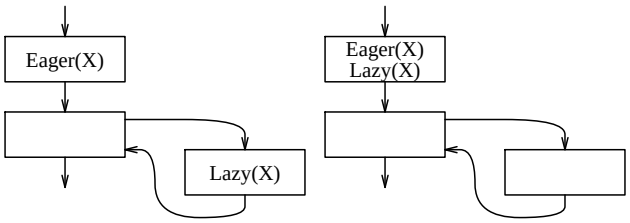

Figure 4: Left: unbalanced production, where one EAGER(X) production is followed by an arbitrary number of LAZY(X) productions. Right: possible solution obeying correctness criterion C1.

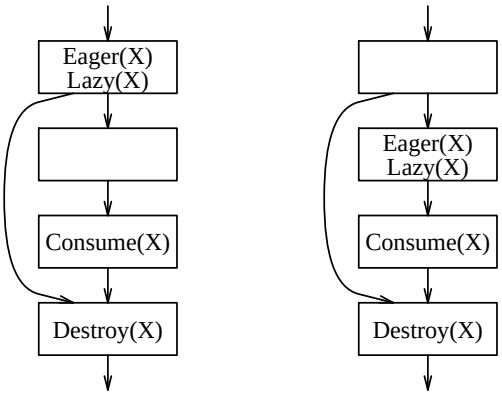

Figure 5: Left: unsafe production. Right: possible solution obeying C2.

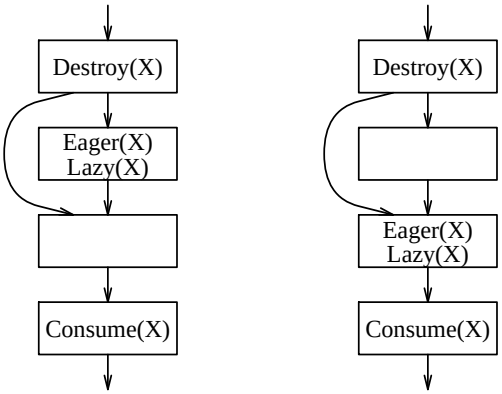

Figure 6: Left: insufficient production. Right: possible solution obeying C3.

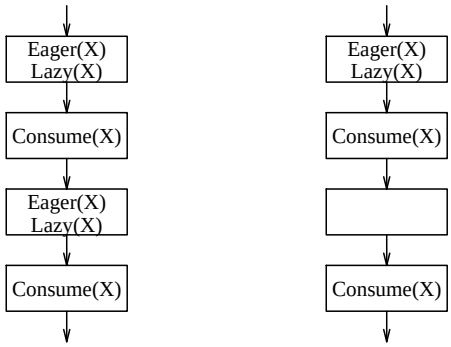

Figure 7: Left: redundant production. Right: possible solution obeying O1.

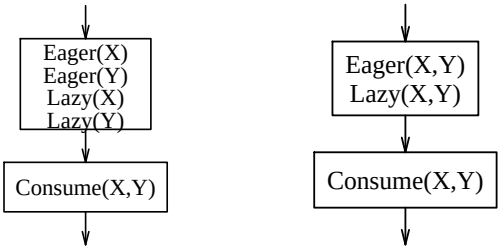

Figure 8: Left: too many producers. Right: possible solution obeying O2.

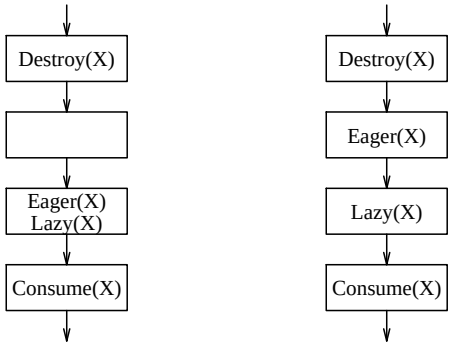

Figure 9: Left: too late production. Right: possible solution obeying O3.

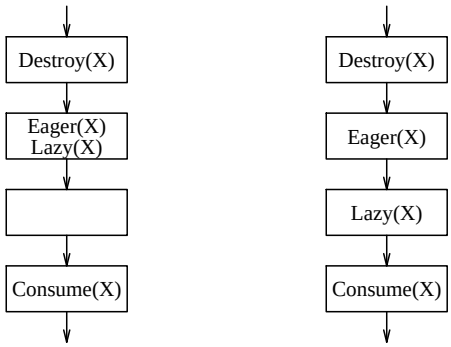

Figure 10: Left: too early production. Right: possible solution obeying O3′.

intervals reflect the loop structure more closely than Allen-Cocke intervals [RP86]. 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. Instead, we operate on one *interval flow graph* $G = (N, E)$, with nodes N and edges E. $\textsc{Root} \in N$ is the unique root of G, which is viewed as a header node for the entire program. For $n \in N$, $\textsc{Level}(n)$ is the loop nesting level of n, counted from the outside in; $\textsc{Level}(\textsc{Root}) = 0$.

We define $T(n) = \emptyset$ for all non-header nodes n, and $T^+(n) = T(n) \cup \{n\}$ for all nodes n. We also define $\textsc{Children}(n)$ to be the set of all nodes in $T(n)$ which are one level deeper than n; $\textsc{Children}(n) = \{c \mid c \in T(n), \textsc{Level}(c) = \textsc{Level}(n) + 1\}$. For each $m \in \textsc{Children}(n)$, we define $J(m)$ to be the immediately enclosing interval, $T(n)$.

One of the main differences between G and a standard control flow graph is the way in which edges $e = (m, n) \in E$ are constructed and classified. In addition to edges that correspond to actual control-flow edges, E may also contain Synthetic edges, which connect the header h of an interval $T(h)$ to all sinks (excluding $T^+(h)$) of edges originating within $T(h)$. Each non-Synthetic edge (m, n) is classified as having one of the following types.

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

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

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

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

We also define $\textsc{Header}(n) = m$ if n is the sink of an Entry edge originating in m (otherwise, $\textsc{Header}(n) = \emptyset$).

Note that Cycle and Jump edges correspond to Tarjan's cycle and cross edges, respectively [Tar74]. However, we divide his forward edges into Forward and Entry edges depending on whether they enter an interval or not (while others divide them into forward and tree edges depending on whether they are part of an embedded tree or not). Note also that for each Jump edge (m, n), G contains $\textsc{Level}(m) - \textsc{Level}(n)$ Synthetic edges.

Give-N-Take requires G to have the following properties:

```
      do i = 1, N
        y(a(i)) = ...
        if test(i) goto 77
      enddo
      do j = 1, N
        ...
      enddo
77    do k = 1, N
        ... = x(k + 10) + y(b(k))
      enddo
```

Figure 11: Example code. We wish to use the j-loop for latency hiding in case the branch out of the i-loop is not taken.

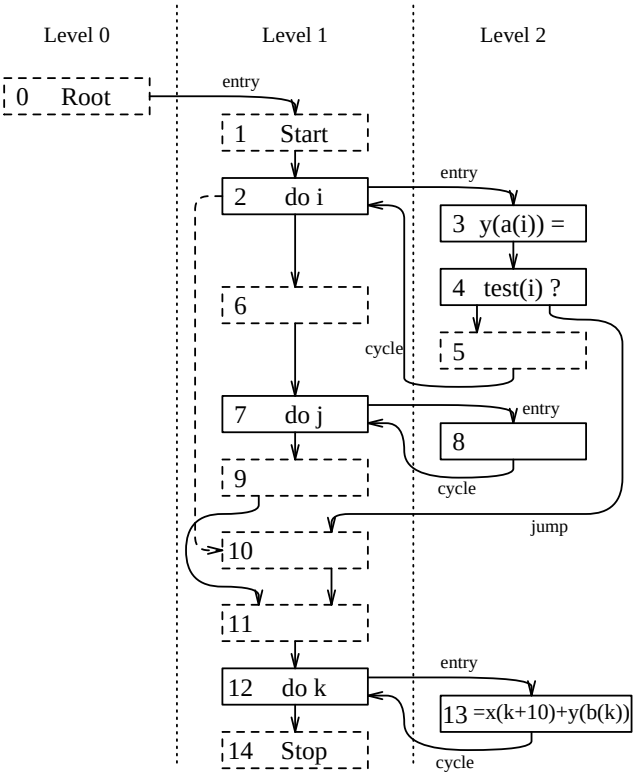

Figure 12: Flow graph for the code from Figure 11. The dashed nodes are synthetic nodes inserted to break critical edges. The dashed edge $(2, 10)$ is a SYNTHETIC edge caused by JUMP edge $(4, 10)$ (since $4 \in T(2)$). All non-FORWARD, non-SYNTHETIC edges are labeled as either ENTRY, CYCLE, or JUMP edges.

- 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 $n \in T(h)$ such that $(n, h) \in E$; *i.e.*, there is only one CYCLE edge out of $T(h)$. We will refer to node n as LASTCHILD(h).
- There are no *critical edges*, which connect a node with multiple outgoing edges to a node with multiple incoming edges. 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.

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. The code shown in Figure 3 is a case of placing production at a synthetic node (the added **else** branch). Note that for the EAGER production on the **else** branch (the "READ$_{Send}\{x(6 : N+5)\}$"), a naïvely placed matching LAZY production (a "READ$_{Recv}\{x(6 : N+5)\}$") might be located right before the k-loop, since LAZY productions are generally delayed as far as possible. This, however, would violate balance, since on the **then** branch the corresponding EAGER production has already been matched by a LAZY production. Therefore, the LAZY production is moved up into the **else** branch.

Each of the requirements above can lead to a growth of G and can therefore slow GIVE-N-TAKE down. (For example, inserting synthetic nodes makes $\mathcal{O}(N) = \mathcal{O}(E)$.) However, it has been noted by several researchers that for typical programs, both the average out-degree of flow graph nodes and the maximal loop nesting depth can be assumed to be bounded by small constant independent of the size of the program [MR90]. Therefore, the increase of G should be fairly small for well structured programs.

Figure 12 shows the interval flow graph for the code in Figure 11. The i-loop, for example, corresponds to the interval $T(2)$ formed by nodes 3, 4, 5, with header 2 (again, the header itself is not part of the interval). Note that FORWARD edges are the only non-SYNTHETIC edges that do not cross nesting level boundaries.

3.4 Traversal Orders and Neighbor Relations

The order in which the nodes of the interval flow 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 being solved (see Section 5). E induces two partial orderings on N:

Vertically: Given a FORWARD/JUMP edge (m, n), a FORWARD order visits m before n, and a BACKWARD order visits m after n.

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

Since these partial orderings are orthogonal, they can be combined into PREORDER (FORWARD and DOWNWARD), POSTORDER (FORWARD and UPWARD), and the corresponding reverse orderings. For example, the nodes in Figure 12 are numbered in PREORDER. Note that in a BEFORE problem, the flow of information is not necessarily in FORWARD order; this will become apparent in the discussion of the algorithm in Section 5.

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 $N - \{n\}$ that has a certain relationship with n. In general, we are interested in information residing at predecessors or successors. However, we are also considering through which type of edge they are connected to n. The edge type carries information about how the neighboring nodes are related to each other (for example, whether moving production from one node to the other constitutes a hoist out of a loop or not). The type also indicates whether this information has already been computed under the current node visiting order or not.

Let TYPE be a set of edge types, where the letters C, E, F, J, and S indicate CYCLE, ENTRY, FORWARD, JUMP, and SYNTHETIC edges, respectively. GIVE-N-TAKE uses the following neighbor relations:

$\text{PREDS}^{\text{TYPE}}(n)$**:** The source nodes of edges reaching n of a type in TYPE.

$\text{SUCCS}^{\text{TYPE}}(n)$**:** The sink nodes of edges originating from n of a type in TYPE.

The conventional "predecessors" and "successors" are then $\text{PREDS}^{\text{CEFJ}}(n)$ and $\text{SUCCS}^{\text{CEFJ}}(n)$, respectively, which we will abbreviate as $\text{PREDS}(n)$ and $\text{SUCCS}(n)$, respectively. We will refer to the transitive closures of $\text{PREDS}^{\text{FJ}}(n)$ and $\text{SUCCS}^{\text{FJ}}(n)$ as the *ancestors* and *descendants* of n, respectively. Note that $\{\text{LASTCHILD}(n)\} = \text{PREDS}^{\text{C}}(n)$, and $\{\text{HEADER}(n)\} = \text{PREDS}^{\text{E}}(n)$. Note also the following implications of the lack of critical edges:

- Let $e = (n, s)$ be a JUMP edge. Then there exists an $h \in N$ with $n \in T(h), s \notin T^+(h)$. Since $T^+(h)$ is by definition strongly connected, n must have successors within $T^+(h)$. Since s is also a successor of n, n must have multiple outgoing edges. However, G does not have critical edges, therefore s has only one predecessor, which is n; *i.e.*, $\text{PREDS}^{\text{CEF}}(s) = \emptyset$. In other words, the sink of a JUMP edge, like node 10 in Figure 12, never has any predecessors besides the source of the JUMP edge.
- Let $e = (n, h)$ be a CYCLE edge. It follows that h 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, it is $h \notin \text{SUCCS}^{\text{EFJ}}(n)$. It follows $\text{SUCCS}^{\text{EFJ}}(n) = \emptyset$ for each source n of an CYCLE 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.

4 Give-N-Take Equations

Given a set of initial variables for each node $n \in N$, which describe consumption, destruction, and side effects at the corresponding location in the program, GIVE-N-TAKE computes the production as a set of result variables for each node. Intermediate stages are the propagation and blocking of consumption, and the placing of production.

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 $\text{PREDS}^{\text{FJ}}(n)$) and there are no such neighbors (such as for a loop entry node), then an empty set results. Subscripts *in*, *out* denote variables for the entry and the exit of a node, respectively (reverse for AFTER problems). 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 13 contains the equations for the data flow variables, which will be introduced in the following sections. We will provide example values from the READ instance for the graph in Figure 12, where x_k, y_a, y_b correspond to references $x(k+10)$, $y(a(i))$, and $y(b(k))$, respectively (values at ROOT are excluded for simplicity).

4.1 Initial Variables

The following variables get initialized depending on the problem to solve, where $\perp$ is the default value.

$\text{STEAL}_{init}(n)$**:** All elements whose production would be voided at n. This can also be used to prevent hoisting productions out of zero-trip loops, if so desired.

In our communication problem, this includes an array portion p if either the contents of this portion get modified at n, or if p itself gets changed, for example if p is an indirect array reference and n modifies the indirection array [HK93].

$$\mathsf{STEAL}(n) = \mathsf{STEAL}_{init}(n) \cup \mathsf{STEAL}_{loc}(\textsc{LastChild}(n)) \quad (1)$$

$$\mathsf{GIVE}(n) = \mathsf{GIVE}_{init}(n) \cup \mathsf{GIVE}_{loc}(\textsc{LastChild}(n)) \quad (2)$$

$$\mathsf{BLOCK}(n) = \mathsf{STEAL}(n) \cup \mathsf{GIVE}(n) \cup \bigcup_{s \in \textsc{Succs}^{\mathrm{E}}(n)} \mathsf{BLOCK}_{loc}(s) \quad (3)$$

$$\mathsf{TAKEN}_{out}(n) = \bigcap_{s \in \textsc{Succs}^{\mathrm{FJS}}(n)} \mathsf{TAKEN}_{in}(s) \quad (4)$$

$$\mathsf{TAKE}(n) = \mathsf{TAKE}_{init}(n) \cup \Big(\bigcup_{s \in \textsc{Succs}^{\mathrm{E}}(n)} \mathsf{TAKEN}_{in}(s) - \mathsf{STEAL}(n) \Big) \cup \Big((\mathsf{TAKEN}_{out}(n) \cap \bigcup_{s \in \textsc{Succs}^{\mathrm{E}}(n)} \mathsf{TAKE}_{loc}(s)) - \mathsf{BLOCK}(n) \Big) \quad (5)$$

$$\mathsf{TAKEN}_{in}(n) = \mathsf{TAKE}(n) \cup (\mathsf{TAKEN}_{out}(n) - \mathsf{BLOCK}(n)) \quad (6)$$

$$\mathsf{BLOCK}_{loc}(n) = \Big(\mathsf{BLOCK}(n) \cup \bigcup_{s \in \textsc{Succs}^{\mathrm{F}}(n)} \mathsf{BLOCK}_{loc}(s)\Big) - \mathsf{TAKE}(n) \quad (7)$$

$$\mathsf{TAKE}_{loc}(n) = \mathsf{TAKE}(n) \cup \Big(\bigcup_{s \in \textsc{Succs}^{\mathrm{EF}}(n)} \mathsf{TAKE}_{loc}(s) - \mathsf{BLOCK}(n)\Big) \quad (8)$$

$$\mathsf{GIVE}_{loc}(n) = (\mathsf{GIVE}(n) \cup \mathsf{TAKE}(n) \cup \bigcap_{p \in \textsc{Preds}^{\mathrm{FJ}}(n)} \mathsf{GIVE}_{loc}(p)) - \mathsf{STEAL}(n) \quad (9)$$

$$\mathsf{STEAL}_{loc}(n) = \mathsf{STEAL}(n) \cup \bigcup_{p \in \textsc{Preds}^{\mathrm{FJ}}(n)} (\mathsf{STEAL}_{loc}(p) - \mathsf{GIVE}_{loc}(p)) \cup \bigcup_{p \in \textsc{Preds}^{\mathrm{S}}(n)} \mathsf{STEAL}_{loc}(p) \quad (10)$$

$$\mathsf{GIVEN}_{in}(n) = \mathsf{GIVEN}(\textsc{Header}(n)) \cup \bigcap_{p \in \textsc{Preds}^{\mathrm{FJ}}(n)} \mathsf{GIVEN}_{out}(p) \cup (\mathsf{TAKEN}_{in}(n) \cap \bigcup_{q \in \textsc{Preds}^{\mathrm{FJ}}(n)} \mathsf{GIVEN}_{out}(q)) \quad (11)$$

$$\mathsf{GIVEN}(n) = \mathsf{GIVEN}_{in}(n) \cup \begin{cases} \mathsf{TAKEN}_{in}(n) & \text{for an } \textsc{Eager} \text{ Problem,} \\ \mathsf{TAKE}(n) & \text{for a } \textsc{Lazy} \text{ Problem.} \end{cases} \quad (12)$$

$$\mathsf{GIVEN}_{out}(n) = (\mathsf{GIVE}(n) \cup \mathsf{GIVEN}(n)) - \mathsf{STEAL}(n) \quad (13)$$

$$\mathsf{RES}_{in}(n) = \mathsf{GIVEN}(n) - \mathsf{GIVEN}_{in}(n) \quad (14)$$

$$\mathsf{RES}_{out}(n) = \bigcup_{s \in \textsc{Succs}^{\mathrm{FJ}}(n)} \mathsf{GIVEN}_{in}(s) - \mathsf{GIVEN}_{out}(n) \quad (15)$$

Figure 13: GIVE-N-TAKE equations.

For Figure 12, we have for example $y_b \in \mathsf{STEAL}_{init}(\{3\})$. (Read as: "For the READ problem, the variable STEAL_{init} at node 3 contains the array portion referenced by $y(b(k))$.")

$\mathsf{GIVE}_{init}(n)$**:** All elements that "come for free," *i.e.*, which are already produced at n.

If we do not use the owner computes rule in communication generation, then this includes local definitions of non-owned data, since a later reference to these data does not need to communicate them in any more.

$y_a \in \mathsf{GIVE}_{init}(\{3\})$.

$\mathsf{TAKE}_{init}(n)$**:** The set of consumers at n.

For communication generation, this is the set of non-owned array references.

$x_k, y_b \in \mathsf{TAKE}_{init}(\{13\})$.

4.2 Propagating Consumption

The following variables, together with the variables defined in Section 4.3, analyze consumption.

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

$y_b \in \mathsf{STEAL}(\{2,3\})$.

$\mathsf{GIVE}(n)$: 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 $T(n)$.

$y_a, y_b \in \mathsf{BLOCK}(\{2,3\})$.

$\mathsf{TAKEN}_{out}(n)$: Things guaranteed to be consumed (before being stolen) on all paths originating in n, excluding n itself. Here we have to consider not only FORWARD and JUMP edges, but also SYNTHETIC edges. (Otherwise we might violate safety by producing something whose only consumer may be skipped due to a jump out of a loop).

$x_k, y_b \in \mathsf{TAKEN}_{out}(\{2,6,7,9\ldots11\})$;
also, $x_k \in \mathsf{TAKEN}_{out}(\{1\})$.

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

$x_k, y_b \in \mathsf{TAKE}(\{12,13\})$.

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

$x_k, y_b \in \mathsf{TAKEN}_{in}(\{6,7,9\ldots13\})$;
also, $x_k \in \mathsf{TAKEN}_{in}(\{1,2\})$.

$\mathsf{BLOCK}_{loc}(n)$: Items blocked by n or by descendants of n within $J(n)$ without being consumed.

$y_a, y_b \in \mathsf{BLOCK}_{loc}(\{1\ldots3\})$.

$\mathsf{TAKE}_{loc}(n)$: 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 successors on ENTRY edges, since they are not guaranteed to be reflected in TAKE (which has to be conservatively small), whereas they will always be considered by $\mathsf{BLOCK}(n)$ (which is conservatively large).

$x_k, y_b \in \mathsf{TAKE}_{loc}(\{6,7,9\ldots13\})$;
also, $x_k \in \mathsf{TAKE}_{loc}(\{1,2\})$.

4.3 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 a production.

$y_a \in \mathsf{GIVE}_{loc}(\{2\ldots7,9\ldots11\})$;
$x_k, y_b \in \mathsf{GIVE}_{loc}(\{12\ldots14\})$.

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

$y_b \in \mathsf{STEAL}_{loc}(\{2\ldots7,9\ldots12,14\})$.

4.4 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. (As described in Section 5, the following variables may differ for the EAGER and for the LAZY solution; this will be indicated in the examples by superscripts.)

$\mathsf{GIVEN}_{in}(n)$: Things that are guaranteed to be available at the entry of n (or, for an AFTER problem, the exit of n.) If n is a first child, then it has everything available that is available at its header (and $\textsc{Preds}^{\mathrm{FJ}} = \emptyset$). 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 variable RES_{out} will ensure that things will be produced also along the paths that originally did not have them available (see Equation 15).

$x_k \in \mathsf{GIVEN}^{eager}_{in}(\{2\ldots14\})$;
$y_a \in \mathsf{GIVEN}^{eager}_{in}(\{4\ldots14\})$;
$y_b \in \mathsf{GIVEN}^{eager}_{in}(\{7\ldots9,11\ldots14\})$.
$x_k, y_b \in \mathsf{GIVEN}^{lazy}_{in}(\{13,14\})$;
$y_a \in \mathsf{GIVEN}^{lazy}_{in}(\{4\ldots14\})$.

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

$x_k \in \mathsf{GIVEN}^{eager}(\{1 \ldots 14\})$;
$y_a \in \mathsf{GIVEN}^{eager}(\{4 \ldots 14\})$;
$y_b \in \mathsf{GIVEN}^{eager}(\{6 \ldots 14\})$.
$x_k, y_b \in \mathsf{GIVEN}^{lazy}(\{12 \ldots 14\})$;
$y_a \in \mathsf{GIVEN}^{lazy}(\{4 \ldots 14\})$.

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

$x_k \in \mathsf{GIVEN}^{eager}_{out}(\{1 \ldots 14\})$;
$y_a \in \mathsf{GIVEN}^{eager}_{out}(\{2 \ldots 14\})$;
$y_b \in \mathsf{GIVEN}^{eager}_{out}(\{6 \ldots 14\})$.
$x_k, y_b \in \mathsf{GIVEN}^{lazy}_{out}(\{12 \ldots 14\})$;
$y_a \in \mathsf{GIVEN}^{lazy}_{out}(\{2 \ldots 14\})$.

4.5 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.

The READ_{Send}'s stem from $x_k \in \mathsf{RES}^{eager}_{in}(\{1\})$ and $y_b \in \mathsf{RES}^{eager}_{in}(\{6, 10\})$; the READ_{Recv}'s are $x_k, y_b \in \mathsf{RES}^{lazy}_{in}(\{12\})$.

$\mathsf{RES}_{out}(n)$**:** 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.

In Figure 12, there is no production needed on exit.

Note that $x \in \mathsf{RES}_{out}(n)$ implies by Equation 15 that $x \notin \mathsf{GIVEN}_{out}(n)$, but that for some $s \in \text{SUCCS}^{\text{FJ}}(n)$ and $p \in \text{PREDS}^{\text{FJ}}(s) - \{n\}$, $x \in \mathsf{GIVEN}_{out}(p)$ must hold. In other words, n must have a successor s which in turn has a predecessor $p \neq n$ that produces an x which is consumed by s and not produced by n. Furthermore, the lack of critical edges implies that s must be the only successor of n, and therefore it does not matter whether we use union or intersection in Equation 15.

Figure 14 shows the code from Figure 11 annotated with communication generation as computed by GIVE-N-TAKE.

5 Solving the Equations

This section presents an algorithm, *GiveNTake*, for solving a code placement problem using the GIVE-N-TAKE framework. Section 4 already listed 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 evaluating these equations.

```
        READ_Send{x(11 : N + 10)}
        do i = 1, N
            y(a(i)) = ...
            if test(i) then
                WRITE_Send{y(a(1 : i))}
                WRITE_Recv{y(a(1 : i))}
                READ_Send{y(b(1 : N))}
                goto 77
            endif
        enddo
        WRITE_Send{y(a(1 : N))}
        WRITE_Recv{y(a(1 : N))}
        READ_Send{y(b(1 : N))}
        do j = 1, N
            ...
        enddo
77      READ_Recv{x(11 : N + 10), y(b(1 : N))}
        do k = 1, N
            ... = x(k + 10) + y(b(k))
        enddo
```

Figure 14: The code from Figure 11 annotated with communication statements.

5.1 The Constraints

The objective of the algorithm is to assign the flow variables at each node a value that is consistent with all equations; *i.e.*, we have to reach a *fixed point*. Note that the number of evaluation iterations to reach a fixed point may be constant, as is usually 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. For GIVE-N-TAKE, there actually 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 (it also implies *fastness* [GW76]). However, since the direction of the flow of information varies across the equations, we still need multiple passes over the control flow graph, solving a different set of equations during each pass.

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 4.2, 4.3, 4.4, and 4.5 defines one set of equations that can be evaluated concurrently. We will re-

fer 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. Since all equations except Equation 12 in S_3 are the same for EAGER and LAZY problems and 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 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*.

To determine an order for solving the GIVE-N-TAKE equations that yields a fixed point after evaluating each equation only once, we have to make sure that an equation is evaluated after the right hand side is fully known. Inspection of the equations yields the following constraints:

- S_1 should be evaluated in BACKWARD order (for example, because Equation 8 defines $\mathsf{TAKE}_{loc}(n)$ in terms of $\mathsf{TAKE}_{loc}(s)$, with $s \in \text{SUCCS}^{\text{EF}}(n)$).
- S_1 should also be evaluated in UPWARD order (*e.g.*, Equation 3)).
- $S_1(n)$ ("the equations from S_1 for node n") should be computed before $S_2(n)$, but after $S_2(\text{CHILDREN}(n))$.
- S_2 should be evaluated in FORWARD order.
- S_3 must be computed in FORWARD, DOWNWARD fashion (*i.e.*, PREORDER), after S_1.
- S_4 has to be evaluated after S_1 and S_3, in any order.

Intuitively, these constraints express that information about consumption is flowing up and back, whereas the availability of production gets propagated forward and down. The production to be inserted at a node, however, again depends on the successors of the node.

5.2 The Algorithm

The resulting algorithm is shown in Figure 15. A formal proof that it does indeed obey all ordering constraints, as well as a proof that GIVE-N-TAKE meets the correctness constraints (C1), (C2), and (C3), can be found elsewhere [HK93].

As already noted, each equation is evaluated only once for each node. Furthermore, each equation depends only on a subset of neighbors. Therefore, the total complexity of GIVE-N-TAKE is $\mathcal{O}(E)$ steps (where the cost of each step depends on the current lattice and its representation, for example bit vectors of a certain length). As already noted in Section 3.3, E can be assumed to be of a size in the order of the program size in most cases; under this assumption, GIVE-N-TAKE as well as other interval-based elimination methods have linear time complexity.

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
  forall $c \in \text{CHILDREN}(n)$, in FORWARD order
    Compute Equations 9, 10
  endforall
  Compute Equations 1...8
endforall
forall $n \in N$, in PREORDER
  Compute Equations 11...13 for EAGER/LAZY
endforall
forall $n \in N$
  Compute Equations 14,15 for EAGER/LAZY
endforall
end

Figure 15: Algorithm *GiveNTake* computing an EAGER/LAZY code placement. RES without subscripts stands for both RES_{in} and RES_{out}.

5.3 BEFORE vs. AFTER Problems

We mentioned earlier that an AFTER problem can essentially be treated as a BEFORE problem with reversed flow of control. However, this also means that the reversed flow graph has to fulfill the same requirements from Section 3.3 as the original graph, which is not trivially the case. For example, ENTRY edges may become CYCLE edges (and vice versa), but each loop may have only one CYCLE edge; this can be satisfied by adding nodes similar to the SYNTHETIC nodes. More severely is the requirement for G to be reducible, which will be violated if the original graph had any JUMP edges, since these will become jumps *into* loops. In fact, this would prevent us from determining a unique set of intervals for the reverse G. For example, consider the flow graph in Figure 16, which may be the result of solving an AFTER problem for a program containing a jump out of a loop. A consumption placed at node 4 might be hoisted into its header (node 3), which would be unsafe (due to the path 1-2-5-3).

In our implementation, we handle this case by using the same interval structure as for the original graph, and preventing hoisting production out of loops that contain JUMP edges. This can be done by either accordingly initializing STEAL_{init} for each header of a loop containing a JUMP edge, or by ignoring for these headers the contributions to TAKE coming from the loop body (see Equation 5).

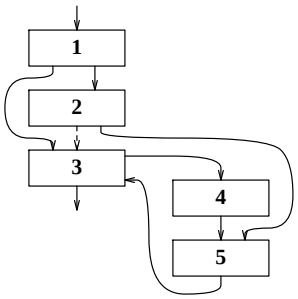

Figure 16: Flow graph containing a jump *into* a loop. Note the synthetic (dashed) edge between nodes 2 and 3.

5.4 A Note on Synthetic Nodes

Having computed the result variables with GIVE-N-TAKE, one still has to perform the actual program optimizations by modifying the analyzed code. This step might be complicated by having production placed at a synthetic node, which would require new basic blocks (see Figure 3). However, it may often be possible to shift production to a neighboring non-synthetic node. This can either be done at code generation time, or by post-processing the results of GIVE-N-TAKE, in a way that is similar to a mechanism employed in edge-placement [Dha88a] for avoiding code proliferation. Our implementation took the latter route, by running a backward pass on G which checks whether these movements can be done without conflicts.

6 Summary

This paper has outlined a general code generation framework, based on Tarjan intervals, that handles several different classes of problems. Unlike previous approaches, it does not assume atomicity. Instead, GIVE-N-TAKE provides both EAGER and LAZY solutions, and it guarantees their balance across arbitrary control flow. Furthermore, GIVE-N-TAKE can be applied to both BEFORE and AFTER problems, and it can take advantage of side effects to further eliminate unnecessary production without affecting balance. Other nice properties of GIVE-N-TAKE include the option to hoist code out of zero-trip loop constructs even for nested loops, and the natural handling of irregular loop bounds and access patterns.

Note, however, that like with code placement strategies in general, there may be conflicting goals in how far to separate production and consumption. Often the computations compete for resources, like registers or message buffers, which could cause some "optimizations" to have a negative effect in practice. While GIVE-N-TAKE does not address this issue directly, certain extensions (such as a heuristic for inserting additional STEAL_{init}'s which blocks production) could help to solve this conflict. Other possible extensions are the combination with dependence analysis (for example by refining the initial assignments to TAKE_{init} and STEAL_{init}), or a more thorough treatment of jumps out of loops for AFTER problems. While our current approach (Section 5.3) prevents unsafe code generation, it may miss some otherwise legal optimizations. Related to that is the issue of analyzing irreducible graphs in general.

We have implemented GIVE-N-TAKE in C++ as part of a Fortran D compiler, where it is used to generate messages for distributed memory machines. We generate READs, WRITEs, and WRITEs combined with different reduction operations (such as summation), all of which can be placed either atomically (for example, for a library call), or divided into sends and receives. The non-atomicity and balance attributes enables message latency hiding and other optimization to be performed across arbitrary control flow. GIVE-N-TAKE's 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 (common subexpression elimination, loop invariant code motion, etc.).

Acknowledgements

We thank Paul Havlak, Chuck Koelbel, and Barbara Ryder for many fruitful data-flow discussions; Paul has also developed much of the symbolic analysis underlying our implementation. Lani Granston, Uli Kremer, Nat McIntosh, and Jerry Roth proofread the paper and were very helpful, especially regarding the illustrating examples. We also thank the PLDI referees, who pointed out several errors and unclarities and aided with the overall presentation.

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.

[Bur90] M. Burke. An interval-based approach to exhaustive and incremental interprocedural data-flow analysis.

ACM Transactions on Programming Languages and Systems, 12(3):341–395, July 1990.

[CK88] D. Callahan and K. Kennedy. Compiling programs for distributed-memory multiprocessors. *Journal of Supercomputing*, 2:151–169, October 1988.

[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.

[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.

[GW76] S. Graham and M. Wegman. A fast and usually linear algorithm for global data flow analysis. *Journal of the ACM*, 23(1):172–202, January 1976.

[Han93] R. v. Hanxleden. Handling irregular problems with Fortran D — A preliminary report. In *Proceedings of the Fourth Workshop on Compilers for Parallel Computers*, Delft, The Netherlands, December 1993. D Newsletter #9, available via anonymous ftp from `softlib.rice.edu` as `pub/CRPC-TRs/reports/CRPC-TR93339-S`.

[HK93] R. v. Hanxleden and K. Kennedy. A code placement framework and its application to communication generation. Technical Report CRPC-TR93337-S, Center for Research on Parallel Computation, Rice University, October 1993. Available via anonymous ftp from `softlib.rice.edu` as `pub/CRPC-TRs/reports/CRPC-TR93337-S`.

[HKK+92] R. v. Hanxleden, K. Kennedy, C. Koelbel, R. Das, and J. Saltz. Compiler analysis for irregular problems in Fortran D. In U. Banerjee et al., editor, *Lecture Notes in Computer Science*, volume 757, pages 97–111. Springer, Berlin, August 1992. From the *Proceedings of the Fifth Workshop on Languages and Compilers for Parallel Computing*, New Haven, CT. 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.

[KLS+94] C. Koelbel, D. Loveman, R. Schreiber, G. Steele, Jr., and M. Zosel. *The High Performance Fortran Handbook*. The MIT Press, Cambridge, MA, 1994.

[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.

[MR90] T. Marlowe and B. Ryder. Properties of data flow frameworks. *Acta Informatica*, 28:121–163, 1990.

[RP86] B.G. Ryder and M.C. Paull. Elimination algorithms for data flow analysis. *ACM Computing Surveys*, 18:77–316, 1986.

[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.