Scalar Queue Conversion: Dynamic Single Assignment for Concurrent Scheduling

Matthew I. Frank
Dept. of Electrical and Computer Engineering
University of Illinois, Urbana-Champaign
mif@uiuc.edu

Saman Amarasinghe
Laboratory for Computer Science
Massachusetts Institute of Technology
saman@lcs.mit.edu

University of Illinois Center for Reliable and High Performance Computing
Technical Report CRHC-03-07
July 18, 2003

Abstract

This paper describes scalar queue conversion, a compiler transformation that makes scalar renaming an explicit operation through a process similar to closure conversion. We demonstrate how to use scalar queue conversion to slice a flow graph into two executable parts. When executed, the backward slice creates queues of suspended computations (continuations). At any point in time execution of the backward slice can be suspended and the queued continuations can be invoked to effect the state transformations of the forward slice. In other words, scalar queue conversion finds the concurrency between the backward and forward slices of a given point in the flow graph. We briefly describe our experience using an implementation of scalar queue conversion as the key subroutine in the SUDS automatic parallelization system for the Raw microprocessor. The SUDS compiler implements a generalized form of loop distribution that can distribute loops that contain inner loops with arbitrary (even irreducible) control flow.

1 Introduction

There are two relatively standard approaches for converting sequential imperative programs into equivalent concurrent programs, Tomasulo's algorithm [36, 19], and compiler based program restructuring based on scalar expansion [24]. Both of these techniques are based on the notion that converting a program to a dynamic single assignment form exposes concurrency.

On the other hand, each of these techniques presents the system designer with a set of tradeoffs. In particular, Tomasulo's algorithm guarantees the elimination of scalar storage (anti- and output-) dependences but schedules locally, across a relatively small window of consecutive instructions, and so partially sequentializes control flow. On the other hand, compiler based restructuring techniques can perform global control dependence analysis, and thus find all of the available flow concurrency in a program, but have not, prior to this work, been capable of eliminating scalar storage dependences across arbitrary unstructured control flow. *Scalar queue conversion eliminates this tradeoff between Tomasulo's algorithm and compiler based program restructuring techniques.*

Informally, renaming turns an imperative program into a functional program. Functional programs have the attribute that every variable is dynamically written at most once. Thus functional programs have no anti- or output- dependences. The cost of renaming is that storage must be allocated for all the dynamically renamed variables that are live simultaneously. The particular problem that any renaming scheme must solve, then, is how to manage the fixed, and finite, storage resources that are available in a real system.

Traditional compiler based renaming techniques, (*e.g.*, scalar expansion), rename only those scalars that are modified in loops with structured control flow and loop bounds that are compile time constants. This enables the compiler to preallocate storage for scalar renaming, but limits the applicability of this technique to structured loops that can be analyzed at compile time. Scalar queue conversion, like modern variants of Tomasulo's algorithm [29, 30], manages scalar renaming resources at runtime with queue data structures. Thus, scalar queue conversion, like Tomasulo's algorithm, is able to rename scalars across arbitrary control

```
sum = 0
i = 0
do
  partial_sum = 0
  j = 0
  use(i, sum)
  do
    use2(sum, partial_sum, i, j)
    partial_sum = partial_sum + 1
    j = next(j)
    c1 = cond1(i, j)
  while c1
  i = i + 1
  sum = sum + partial_sum
  c2 = cond2(i)
while c2
use(sum)
```

Figure 1: An example program with a doubly nested loop.

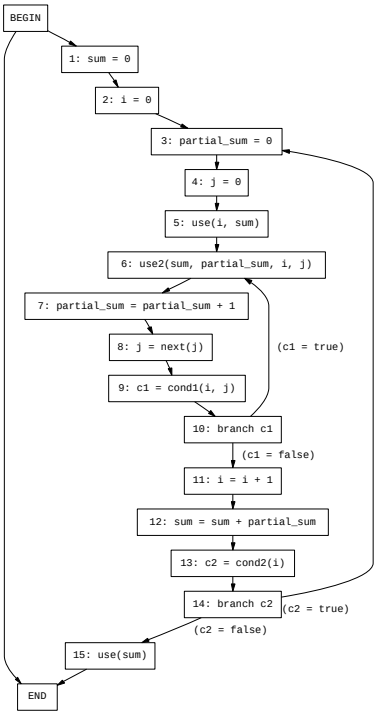

Figure 2: The control flow graph corresponding to the program in Figure 1.

flow.

Tomasulo's algorithm (and trace based scheduling algorithms, in general [15, 18]), however, schedule only locally, and are unable to schedule across the mispredicted branches that exit innermost loops. Thus, Tomasulo's algorithm is unable to exploit concurrency outside of inner loops. Scalar queue conversion performs global control dependence analysis [14, 12], and is thus able to exploit concurrency in outer loops as well.

Because scalar queue conversion both manages scalar renaming resources at runtime, rather than compile time, and does global control dependence analysis, it is able to exploit concurrency in situations where both Tomasulo's algorithm and traditional compiler based restructuring algorithms fail. In particular, scalar queue conversion can exploit the concurrency in *outer* loops of programs with arbitrary *unstructured* control flow. This is the situation that we have found to be the common case in practice [16].

The next section introduces the running example we will use throughout the paper, and defines a few basic terms. Section 3 describes the scalar queue conversion transformation. Section 4 describes unidirectional renaming, the static renaming technique that scalar queue conversion uses to eliminate scalar def-def chains that would otherwise restrict scheduling. Section 5 describes some of the practical issues we encountered in our implementation. Section 6 describes related work. Section 7 concludes.

2 Running Example

The concepts in the rest of this paper are illustrated with respect to an example based on the program shown in Figure 1. We have done our best to choose the example such that it illustrates the relationships between the relevant ideas, but so that it is not so complicated as to overwhelm the reader. The control and data dependence patterns in the example program of Figure 1 are representative of the kind of control and data dependence patterns we found in several sparse matrix creation codes after applying recurrence reassociation. (The variable `i` corresponds to the row number, `j` corresponds to the column number of a non-zero entry, and `sum` and `partial_sum` represent the reassociated index into the array where non-zero entries are being stored).

We will use a standard control flow graph representation of programs. The flow graph for the code in Figure 1 is shown in Figure 2.

The example problem is as follows. Suppose we want to apply loop distribution to the example program. Roughly speaking, the loop distribution algorithm described in Section 5 starts by identifying the loop carried (cyclic) dependences of the loop (in this case, the variables `i` and `sum`), and then creates separate loops for each loop carried dependence. So let us reschedule the loop in Figure 2 into two loops, one that

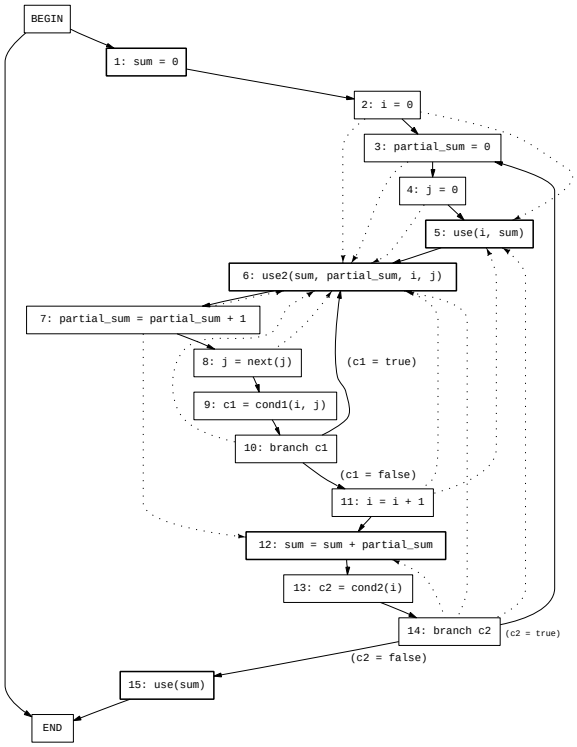

Figure 3: Partitioning the outer loop into the two subsets, 2, 3, 4, 7, 8, 9, 10, 11, 13, 14 and 1, 5, 6, 12, 15 produces a *unidirectional cut* because no dependence edges flow from the second subset into the first. Cut dependence edges are shown in dotted lines. They all flow from the first subset into the second.

does the work corresponding to nodes 2, 3, 4, 7, 8, 9, 10, 11, 13 and 14, and one corresponding to nodes 1, 5, 6, 12 and 15. In particular, nodes 1 and 12 represent all of the definitions of the variable `sum`, while nodes 5, 6, 12 and 15 represent all the uses of variable `sum`.

Is there a legal way to restructure the code to effect this rescheduling? We will demonstrate, in Section 3, that this transformation is legal exactly because the flow of value and control dependences across the partitioning of nodes in the region is *unidirectional*.

To conserve space, we will assume that the reader is familiar with the standard definitions of *dominance*, *postdominance* and *dominance frontiers* [26], *control dependence* [14, 12], *def-use chains*, *use-def chains*, *def-def chains* and *du-webs*. Most of these definitions can be found in a standard undergraduate compiler textbook.

We reserve the terms "def-use, use-def and def-def chains," for dependences between registers (scalars that are provably unaliased). We will also define a particularly conservative set of dependences with respect to memory operations (load and store instructions). We say that any memory operation, x, *reaches* memory operation, y, if there is a path from x to y in the control flow graph. We say there is a *memory dependence* from x to y if at least one of x and y is a store instruction. (That is, we don't care about load-load dependences).

We define the *conservative program dependence graph* as the graph constructed by the following procedure. Take the nodes from the control flow graph. For every pair of nodes, x, y, insert an edge, $x \rightarrow y$, if there is either a def-use-chain from x to y, a use-def-chain from x to y, a def-def-chain from x to y, a memory dependence from x to y or a control dependence from x to y.

We define the *value dependence graph* as the graph constructed by the following procedure. Take the nodes from the control flow graph. For every pair of nodes, x, y, insert an edge, $x \rightarrow y$, if there is either a def-use-chain from x to y, a memory dependence from x to y or a control dependence from x to y. Thus the value dependence graph is the subgraph of the conservative program dependence graph created by removing the use-def and def-def chains from the conservative program dependence graph. A portion of the value dependence graph for our example flow graph is shown in Figure 3.

3 Scalar Queue Conversion

In this section we will show that the compiler can restructure a given flow graph code to eliminate the register storage dependences across a class of flow graph partitionings that we call *unidirectional cuts*. Informally, a unidirectional cut corresponds to a slicing of the

value dependence graph such that information is flowing in only one direction across the slice. The ability to eliminate register storage dependences across unidirectional cuts means that *instruction scheduling algorithms can make instruction ordering decisions irrespective of register storage dependences*. The increased flexibility results in schedules that would otherwise be impossible to construct.

We call this transformation to eliminate register storage dependences *scalar queue conversion*, because it completely generalizes the traditional technique of scalar expansion [24] to arbitrary unstructured (even irreducible) control flow, and provably eliminates all register anti- and output-dependences that would violate a particular static schedule. In Section 5 we show how to use scalar queue conversion as the key subroutine to enable a generalized form of loop distribution. Loop distribution is best viewed as a scheduling algorithm that exposes the available parallelism in a loop [24]. The loop distribution algorithm in Section 5 generalizes previous scheduling techniques by scheduling across code with completely arbitrary control flow, in particular, code with inner loops. This generalization is possible only, and exactly, because scalar queue conversion guarantees the elimination of all register anti- and output-dependences.

The intuition behind the transformation is that *every imperative program is semantically equivalent to some functional program* [25, 20, 1]. Since a functional program never overwrites any part of an object (but rather creates an entirely new object) there are no storage dependences.

Another way to view the transformation is to compare it to the dynamic register renaming performed by Tomasulo's algorithm [36, 19]. Tomasulo's algorithm performs a dynamic mapping of "virtual" register names to "physical" registers, each of which is written only once. After this renaming all register storage dependences are eliminated, because (conceptually) no physical register ever changes its value. Thus, the instruction scheduling algorithm is less constrained by register storage dependences.

More concretely, instead of executing a piece of code, we can defer execution of that piece of code by turning it a closure. A *closure* can be thought of as a suspended computation [25, 31]. It is typically implemented as a data structure that contains a copy of each part of the state required to resume the computation, plus a pointer to the code that will perform the computation. There are then a set of operations that we can perform on a closure:

1. We can *allocate* a closure by requesting a portion of memory from the dynamic memory allocator that is sufficient to hold the required state plus code pointer.

2. We can *fill* a closure by copying relevant portions of the machine state into the allocated memory structure.

3. We can *invoke* a closure by jumping to (calling) the closures code pointer and passing a pointer to the associated data structure that is holding the relevant machine state.

Closures will be familiar to those who have used lexically scoped programming languages. For example, in `C++` and `Java` closures are called *objects*. In these languages closures are *allocated* by calling operator `new`, *filled* by the constructor for the object's class, and *invoked* by calling one of the methods associated with the object's class.

In the general case we can *defer* execution of some subset of the code by creating a closure for each deferred piece of code, and saving that closure on a queue. Later we can *resume* execution of the deferred code by invoking each member of the queue in FIFO order.

3.1 Unidirectional Cuts

Now we define a *cut* of the set of nodes in a region, R, as a partitioning of the set of nodes into two subsets, A, B such that $A \cap B = \emptyset$ and $A \cup B = R$. We say that a cut is *unidirectional* iff there are no edges $x \rightarrow y$ such that $x \in B$ and $y \in A$. That is, all the edges either stay inside A, stay inside B or flow from A to B, and *no* edges flow from B to A. For example, given the region corresponding to the outer loop in Figure 3, the partition $\{2, 3, 4, 7, 8, 9, 10, 11, 13, 14\}$ and $\{1, 5, 6, 12, 15\}$ is a unidirectional cut because there are no def-use chains, memory or control dependences flowing from the second set to the first.

In the following sections we will demonstrate that by the process of *queue conversion* we can always transform a unidirectional cut A-B of a single-entry single-exit region into a pair of single-entry single-exit regions, that produce the same final machine state as the original code, but have the feature that all of the instructions from partition A execute (dynamically) before all the instructions from partition B.

Any particular value dependence graph might have many different unidirectional cuts. The criteria for choosing a specific cut will depend on the reasons for performing the transformation. In Section 5 we will discuss one method for efficiently identifying a useful set of unidirectional cuts for loop distribution.

3.2 Maximally Connected Groups

First we will show that we can create a "reasonable" flow graph that consists only of the nodes from subset A of a unidirectional A-B cut. The property that makes this possible is that every *maximally connected* group of the nodes from subset B will have only a single exit. Thus we can remove a maximally connected subset of nodes from subset B from the region flow graph and "fix-up" the breaks in the flow graph by connecting the nodes that precede the removed set to the (unique) node that succeeds the removed set.

Given a unidirectional cut A-B of a flow graph then we will call a subset of nodes $\beta \subset B$ in the graph a *maximally connected group* iff every node in β is connected in the flow graph only to other nodes of β or to nodes of A. That is, given $\bar{\beta} = B - \beta$ and nodes $b \in \beta$, $\bar{b} \in \bar{\beta}$ there are no edges $b \rightarrow \bar{b}$ or $\bar{b} \rightarrow b$. For example, given the unidirectional cut shown in Figure 3 where $A = \{2, 3, 4, 7, 8, 9, 10, 11, 13, 14\}$ and $B = \{1, 5, 6, 12, 15\}$, the maximally connected groups are the subsets $\{1\}$, $\{5, 6\}$, $\{12\}$ and $\{15\}$ of B.

But now suppose that we are given a unidirectional cut A-B. This means that there can be no control dependences from B to A. Informally, there are no branches in B that can in any way determine when or if a node in A is executed. Now suppose that we are given a maximally connected group $\beta \subset B$. If β has an exit edge $b \rightarrow a$ (an edge where $b \in \beta$, $a \notin \beta$), then, because β is maximally connected it must be the case that $a \in A$. The node a can not be in B because then β would not be maximally connected.

If there are two (or more) such exit edges, $b_0 \rightarrow a_0$ and $b_1 \rightarrow a_1$, where $b_0 \neq b_1$ then it must be the case that there is a branch or set of branches in β that causes the flow graph to fork. In particular, b_0 and b_1 must have different control dependences, and at least one of those control dependences must be on a node inside β. But a_1 and a_0 can not be control dependent on any node inside β, because they are on the wrong side of the A-B cut.

Consider node a_0. There is an edge from b_0 to a_0, thus there is at least one path from b_0 to `exit` that passes through a_0. But a_0 is not control dependent on b_0, so *every* path from b_0 to `exit` must pass through a_0. Thus a_0 postdominates b_0. Similarly, for every node $b_i \in \beta$ such that there is any path from b_i to b_0, it must be the case that a_0 postdominates b_i.

Consider this set of $b_i \in \beta$ that are on a path to b_0. Now, β is connected, thus there must either be a path from b_i to b_1 or there must be a path from b_1 to b_i. If there is a path from b_1 to b_i then there is a path from b_1 to b_0 and thus a_0 also postdominates b_1. Suppose there is no path from b_1 to b_0, then there must be a path from one of the b_i to b_1. But we already know that *every* path from b_i to `exit` goes through a_0, so every path from b_1 to `exit` must go through a_0. Thus a_0 postdominates both b_0 and b_1.

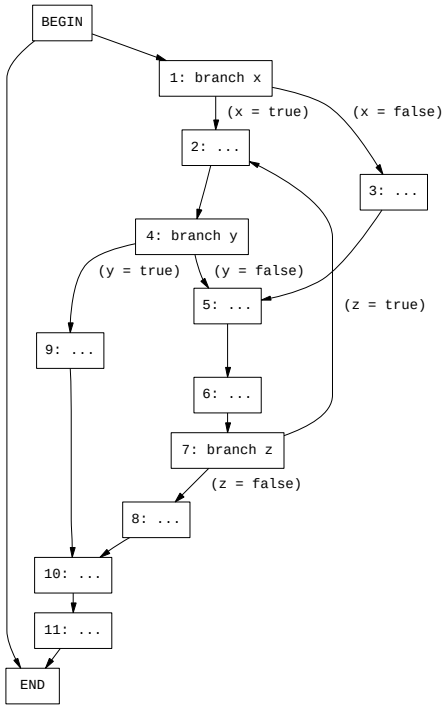

Figure 4: Any maximally connected subset of nodes from the bottom of a unidirectional cut *always* exits to a single point. In this case (an irreducible loop) if either node 4 or 7 is in the bottom of a unidirectional cut then so must all the nodes 2, 4, 5, 6, 7, 8 and 9. Thus a maximally connected subset containing node 4 or node 7 will exit to node 10.

By a similar argument a_1 postdominates both b_1 and b_0. More specifically, a_1 *immediately* postdominates b_1, because there is a flow graph edge $b_1 \rightarrow a_1$. Thus a_0 must postdominate a_1 if it is to also postdominate b_1. A similar argument shows that a_1 must postdominate a_0. Postdominance is a partial order, thus $a_0 = a_1$. So the maximally connected group β exits to a unique node in A.

As an example, consider Figure 4. This figure shows a flow graph containing an irreducible loop. Suppose that we would like to include node 4 (a branch instruction) in set B of a unidirectional A-B cut. We will demonstrate that any maximally connected group $\beta \subset B$ that contains node 4 must also contain nodes 8 and 9, and will, therefore, exit through node 10. We can see this by examining Figure 5, which shows control dependence graph corresponding to the flow graph in Figure 4. There is a cycle in the control dependence graph between the two exit branches in nodes 4 and 7. Thus if either of the exit branches for the irreducible

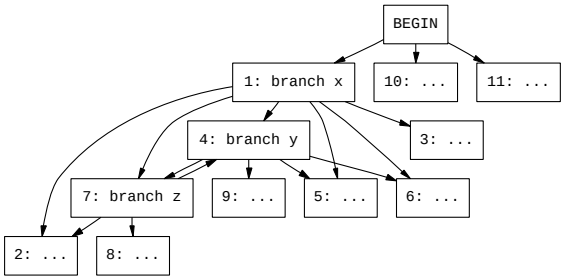

Figure 5: The control dependence graph for the flow graph in Figure 4 has a cycle between nodes 4 and 7. Thus both nodes must be on the same side of a unidirectional cut of the flow graph.

loop is included on one side of the unidirectional cut, then the other must as well, because we require that no control dependences in a unidirectional cut flow from B to A.

Given a unidirectional cut A-B of a flow graph we can efficiently find all the maximally connected groups $\beta \subset B$ as follows. First we scan the edges of the flow graph to find all the edges $b_j \rightarrow a_i$ where $b_j \in B$ and $a_i \in A$. By the argument above the set of nodes a_i found in this manner represent the set of unique exits of maximal groups $\beta_i \subset B$. Then for each a_i we can find the associated maximally connected group β_i by performing a depth first search (backwards in the flow graph by following predecessor edges) starting at a_i, and where we follow only edges that lead to nodes in B.

For example, recall that in Figure 3 the maximally connected subgroup $\{5, 6\}$ exits to node 7. A backwards search from node 7 finds nodes 5 and 6 from set B but does not find node 12, because that would require traversing intermediate nodes (*e.g.*, node 4) that are in set A.

Now we can create a flow graph that performs exactly the work corresponding to part A of the unidirectional A-B cut by removing each of the maximally connected groups of B one by one. Given a maximally connected group $\beta_i \subset B$ with entry edges $a^y_{i_0} \rightarrow b^y_{i_0}, \ldots, a^y_{i_n} \rightarrow b^y_{i_n}$ and exits $b^x_{i_0} \rightarrow a^x_i, \ldots, b^x_{i_n} \rightarrow a^x_i$ to the unique node a^x_i, then we can remove β_i from the flow graph by removing all the nodes of β_i from the flow graph, and inserting the edges $a^y_{i_0} \rightarrow a^x_i, \ldots, a^y_{i_n} \rightarrow a^x_i$. We call the resulting flow graph the *sliced flow graph for partition* A.

Figure 6 shows the sliced flow graph for the partition $\{2, 3, 4, 7, 8, 9, 10, 11, 13, 14\}$. The maximal groups in the original flow graph (Figure 3) were the sets $\{5, 6\}$, and $\{12\}$. The entry edges to $\{5, 6\}$ were $\{4 \rightarrow 5\}$ and $\{10 \rightarrow 6\}$, while the exit edge was $\{6 \rightarrow 7\}$. Thus in the sliced flow graph we remove nodes 5 and 6 and insert edges $\{4 \rightarrow 7\}$ and $\{10 \rightarrow 7\}$. Node 12 is removed and the edge $\{11 \rightarrow 13\}$ is inserted. Similarly, nodes 1 and 15 have been removed and edges connecting their entries to their exits have been inserted.

3.3 The Deferred Execution Queue

In addition to creating a flow graph that performs exactly the work corresponding to part A of a unidirectional A-B cut, we can also annotate the flow graph so that it keeps track of exactly the order in which the maximal groups $\beta_i \subset B$ will be executed. We do this by creating a queue data structure at the entry point of the region flow graph. We call this queue the *deferred execution queue*.

Every edge $a^y_{i_j} \rightarrow b^y_{i_j}$, $a^y_{i_j} \in A$, $b^y_{i_j} \in \beta_i$ in the flow graph represents a point at which control would have entered the maximal group β_i. Likewise, every edge $b^x_{i_k} \rightarrow a^x_i$, $b^x_{i_k} \in \beta_i$, $a^x_i \in A$, represents exactly the points at which control would have returned to region A.

Thus, after creating the sliced flow graph for partition A, by removing the regions β_i from the flow graph (as described in the previous section), we can place an instruction along each edge $a^y_{i_j} \rightarrow a^x_i$ that *pushes* the corresponding code pointer for the node $b^y_{i_j}$ on to the deferred execution queue. The edges $a^y_{i_j} \rightarrow a^x_i$ execute in exactly the order in which the β_is would have executed in the original flow graph. Thus after execution of the sliced flow graph for partition A, the deferred execution queue will contain all of the information we need to execute the code from partition B in exactly the correct order and exactly the correct number of times.

We can accomplish this by converting each β_i into a procedure that contains a flow graph identical to the flow graph that corresponds to the original β_i, but *returns* at each exit point of β_i.[1] Then we can recreate the original execution sequence of partition B by *popping* each code pointer $b^y_{i_j}$ off the front of the deferred execution queue and calling the corresponding procedure.

The queue conversion of our example program is shown in Figure 6. Push instructions for the appropriate maximal group entry points have been inserted along the edges `begin` $\rightarrow$ 2, 4 $\rightarrow$ 7, 10 $\rightarrow$ 7, 11 $\rightarrow$ 13 and 14 $\rightarrow$ `end`. The maximal groups $\{1\}$, $\{5, 6\}$, $\{12\}$ and $\{15\}$ are each converted into a procedure.

[1] If the underlying infrastructure does not support multiple-entry procedures, then each maximal group β_i can be further partitioned into a set of subprocedures, each corresponding to a maximal basic block of β_i. Each subprocedure that does not exit β_i tail calls [31] its successor(s) from β_i.

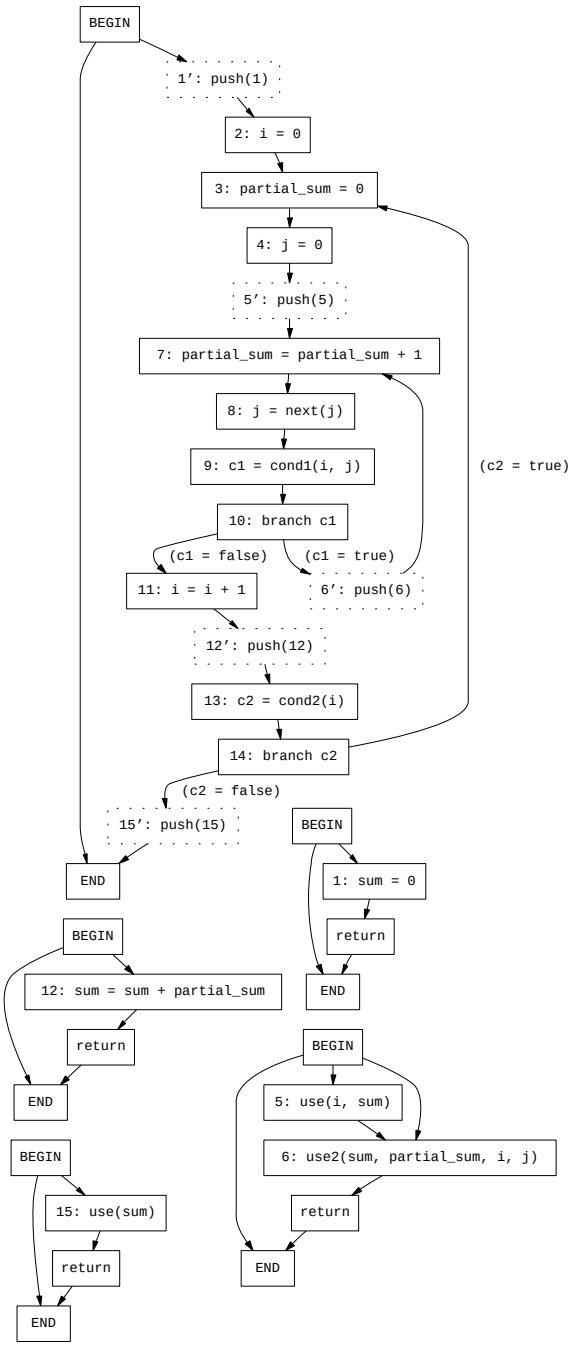

Figure 6: The sliced flow graph for partition A, consisting of nodes 2, 3, 4, 7, 8, 9, 10, 11, 13 and 14. For example, nodes 4 and 10 (the entries to the maximal group consisting of nodes 5 and 6) are connected to node 7, (the single exit node for group 5, 6). Queue conversion annotates the sliced flow graph for A with instructions that record which maximal groups of B would have executed, and in what order. Each maximal group of B is converted into its own procedure.

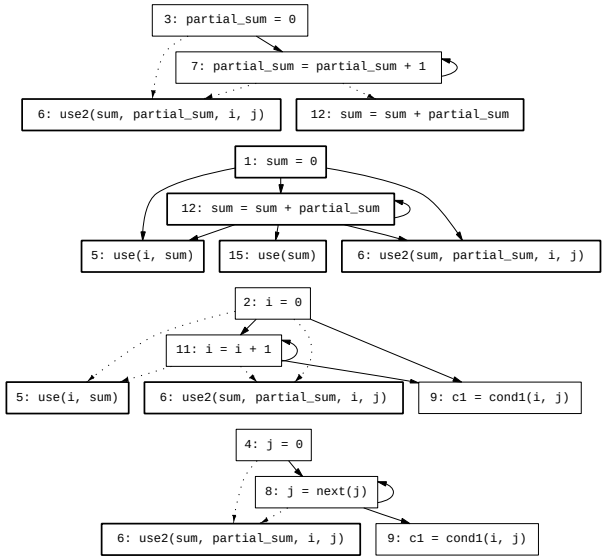

Figure 7: Cuts in the du-webs for variables `i`, `j`, `sum` and `partial_sum` given the cut from nodes 2, 3, 4, 7, 8, 9, 10, 11, 13, 14 to nodes 1, 5, 6, 12, 15 (shown in bold). Def-use chains that cross the cut are shown as dotted edges.

Closure Conversion

If it were the case that there were no register storage dependences flowing from B to A then the deferred execution queue would be sufficient. Our definition of a unidirectional A-B cut did not, however, exclude the existence of use-def or def-def chains flowing from region B to region A. Thus, we must solve the problem that partition A might produce a value in register x that is used in region B but then might overwrite the register with a new value before we have a chance to execute the corresponding code from partition B off the deferred execution queue.

The problem is that the objects we are pushing and popping on to the deferred execution queue are merely code pointers. Instead, we should be pushing and popping closures. A closure is an object that consists of the code pointer together with an *environment* (set of name-value pairs) that represents the saved machine state in which we want to run the corresponding code. Thus a closure represents a suspended computation.

Consider the registers (variables) associated with the set of def-use chains that reach into a maximal group $\beta_i \subset B$. If we save a copy of the values associated with each of these registers along with the code pointer, then we can eliminate all the use-def chains that flow from B to A, and replace them, instead, with use-def chains that flow only within partition A.

To convert each maximal group $\beta_i \subset B$ into a closure we transform the code as follows.

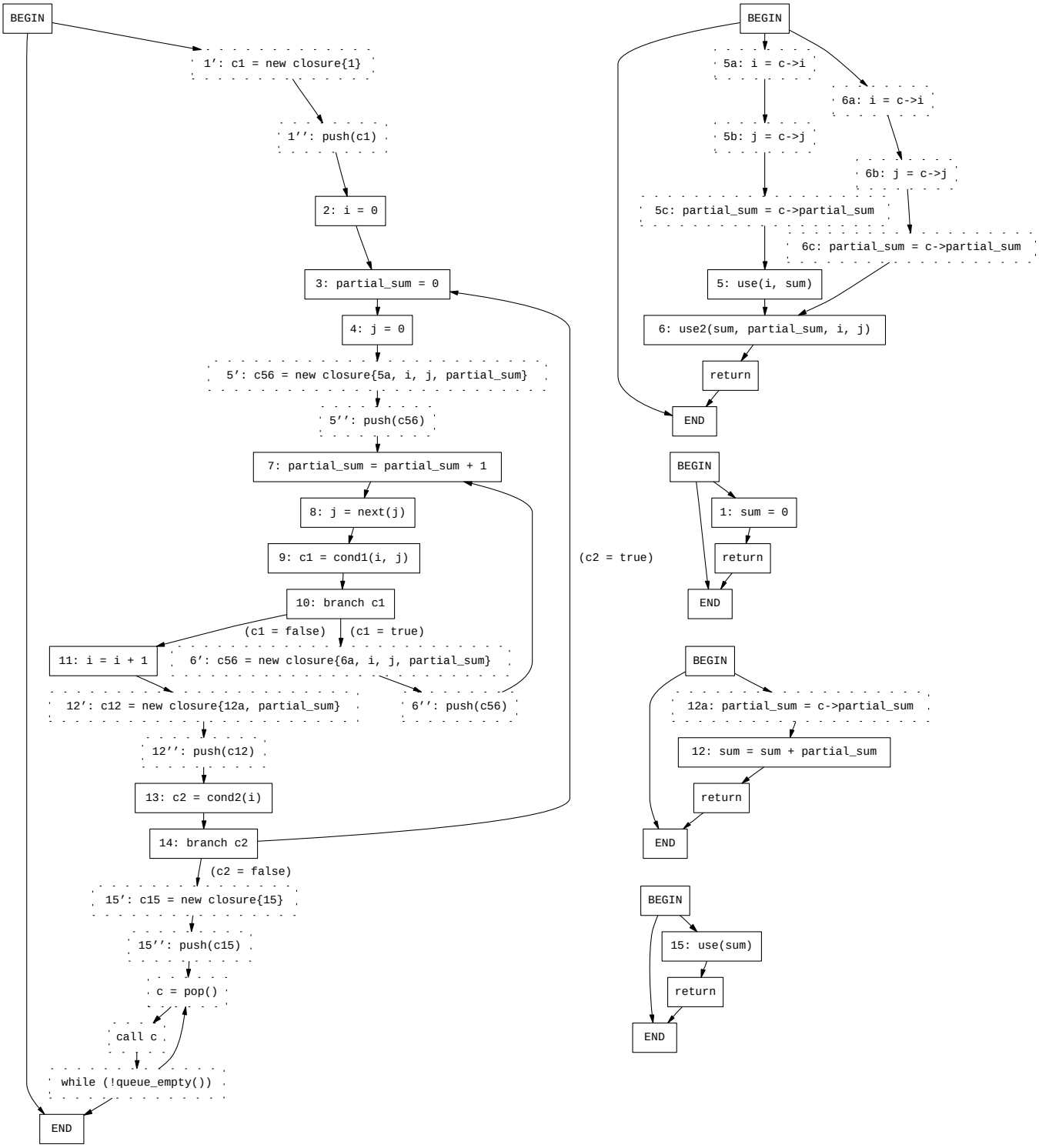

Figure 8: Closure conversion ensures that each value crossing the cut gets copied into a dynamically allocated structure before the corresponding register gets overwritten.

1. Consider the graph of nodes corresponding to β_i. For each of the entry nodes $b^y_{i_j}$ of this graph find the set of nodes $\beta_{i_j} \subset \beta_i$ reachable from $b^y_{i_j}$. For each set β_{i_j} find the set of variables, $V_{ij} = \{v_{ijk}\}$ such that there is a def-use chain flowing from partition A into β_{i_j}. (That is, there is a definition of v_{ijk} somewhere in A and a use of v_{ijk} somewhere in β_{i_j}). Figure 7 shows that this set can be easily derived from the du-webs corresponding to the flow graph. For example, $V_{\{12\}} = \{\texttt{partial\_sum}\}$ and $V_{\{15\}} = \emptyset$. The maximal group $\beta_{\{5,6\}}$ has two entry points, (at 5 and 6). In this case it happens that $V_{\{5,6\},5} = V_{\{5,6\},6} = \{\texttt{i}, \texttt{j}, \texttt{partial\_sum}\}$.

2. Consider each edge $a^y_{i_j} \rightarrow a^x_i$ in the sliced flow graph for partition A that corresponds to entry point $b^y_{i_j}$ of maximal group β_i. Along this edge we place an instruction that dynamically allocates a structure with $|V_{ij}|+1$ slots, then copies the *values* $\langle b^y_{i_j}, v_{ij1}, \ldots, v_{ij|V_{ij}|}\rangle$ into the structure, and then pushes a pointer to this structure onto the deferred execution queue. Figure 8 demonstrates this process. For example, along the edge $4 \rightarrow 7$ we have placed instructions that allocate a structure containing the values of the code pointer, "5", and the copies of the values contained in variables, `i`, `j` and `partial_sum`.

3. For each β_i we create a procedure that takes a single argument, `c`, which is a pointer to the structure representing the closure. The procedure has the same control flow as the original subgraph for β_i except that along each entry we place a sequence of instructions that copies each entry from each slot of the closure into the corresponding variable v_{i_k}. Figure 8 shows that the two entries to the procedure corresponding to the maximal group $\{5,6\}$ have been augmented with instructions that copy the values of variables `i`, `j` and `partial_sum` out of the corresponding closure structure.

4. To invoke a closure from the deferred execution queue we pop the pointer to the closure off the front of the queue. The first slot of the corresponding structure is a pointer to the code for the procedure corresponding to β_i. Thus we call this procedure, passing as an argument the pointer to the closure itself. In Figure 8 this process is shown towards the bottom of the original procedure, where we have inserted a loop that pops closures off the deferred execution queue, and invokes them.

This completes the basic scalar queue conversion transformation. Because a copy of each value reaching a maximal group β_i is made just before the point in the program when it would have been used, the correct set of values reaches each maximal group, even when execution of the group is deferred. Additionally, since the copy is created in partition A, rather than partition B, we have eliminated any use-def chains that flowed from partition B to partition A. In the next section we will demonstrate how to generalize the result to eliminate def-def chains flowing from B to A.

4 Unidirectional Renaming

In the previous section we demonstrated that we could transform a unidirectional A-B cut on a single-entry single-exit region into an equivalent piece of code such that all the instructions in partition A run, dynamically, before all the instructions in partition B. Further we demonstrated that we could do this even in the presence of use-def chains flowing from partition B to partition A. In this section we will show that the result can be generalized, in a straightforward way, to A-B cuts where there are additionally def-def chains flowing from partition B to partition A.

The result depends on the fact that given a unidirectional A-B cut, we can insert a new instruction anywhere in the flow graph, and that if we give that instruction a labeling that includes it in partition B, then we will not introduce any new control dependences that flow from partition B to partition A. (The opposite is not true. That is, if we place a new instruction in partition A at a point that is control dependent on an instruction in partition B, then we will introduce a control dependence edge that will violate the unidirectionality of the cut.)

For the remainder of the paper we will assume that each du-web in the program has been given a unique name. This transformation is already done by most optimizing compilers because it is so common for programmers to reuse variable names, even when the variables are completely independent. For example, many programmers reuse the variable name `i` for the index of most loops. Once the du-webs are calculated we iterate through the set of du-webs for each variable x, renaming all the uses and definitions in each node in the ith web to x_i. Thus we can, without loss of generality, talk about *the* du-web for a particular variable.

Now consider the du-web for variable x on a unidirectional cut A-B where some of the definitions of x are in A and some of the uses of x are in B. Thus, there is a value dependence flowing from A to B. It may be the case that there are definitions of x in B and uses of x in A, but, because A-B is a unidirectional cut, it cannot be the case that there are any def-use chains reaching from B to A. Thus the du-web has a unidirectional structure, just as the value dependence graph did. (In fact,

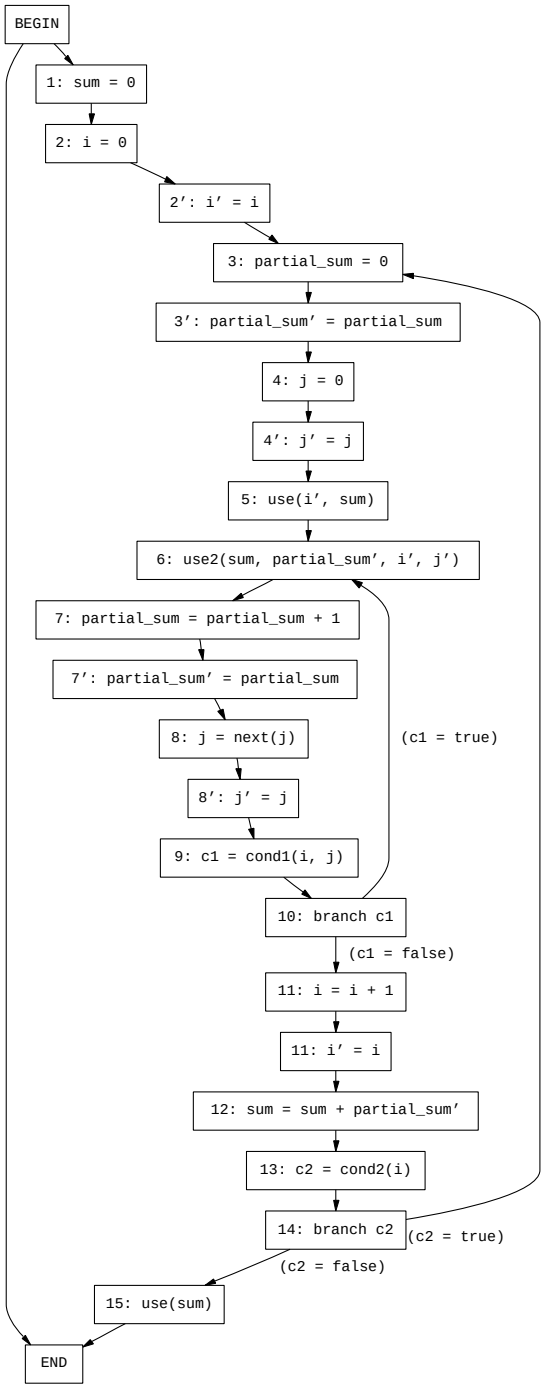

Figure 9: An example of statically renaming the variables `i`, `j` and `partial_sum`.

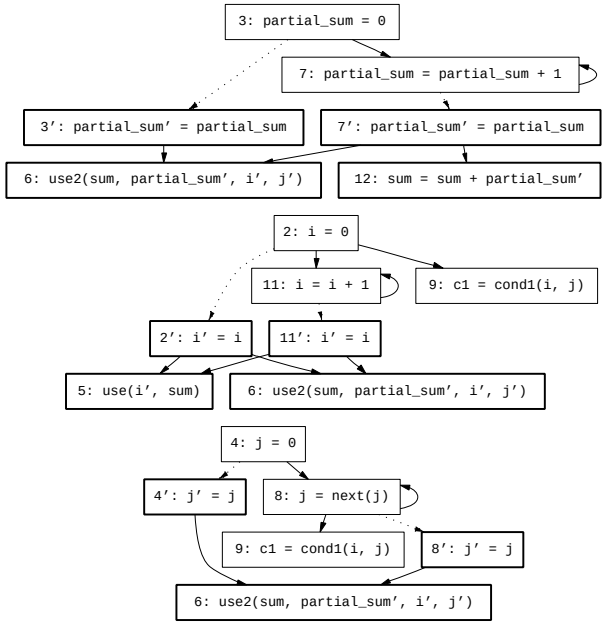

Figure 10: The unidirectionally renamed du-webs for variables `i`, `j` and `partial_sum`.

another way of seeing this is to observe that each du-web is an induced subgraph of the value dependence graph). For example, in the du-webs shown in Figure 7 one can observe that the def-use chains crossing the cut (shown with dotted edges) all flow in one direction.

The du-web for variable x thus has a structure that is *almost* renameable, except for those edges in the web that cross the cut. Suppose, however that we were to place a copy instruction "`x′ = x`" directly after each of the definitions of x from A that reach a use in B. Then we could rename all the definitions and uses of x in B to x′. The program will have exactly the same semantics, but we will have eliminated all of the def-def chains flowing from B to A. We will call such a renaming of of a du-web that crosses a unidirectional cut a *unidirectional renaming*.

An example of a unidirectional renaming is shown in Figure 9. Each time one of the variables `i`, `j` and `partial_sum` is modified it is copied to a corresponding variable `i′`, `j′` or `partial_sum′`. The uses of `i`, `j` and `partial_sum` in partition B are then renamed to `i′`, `j′` and `partial_sum′`. The du-webs for this unidirectional renaming are shown in Figure 10.

To see how unidirectional renaming eliminates backwards flowing def-def chains, consider Figure 11. We examine the cut from the set of nodes $\{1, 2, 3, 4, 6, 7\}$ to the set $\{5, 8\}$. This is a unidirectional cut because all of the value and control dependences flow from the first set to the second. Figure 12 shows the corresponding du-web for variable x. There is, however, a def-def chain flowing from node 5 to node 7 (against the cut

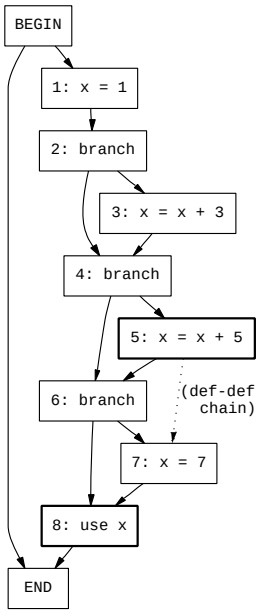

Figure 11: The cut separating nodes 1, 2, 3, 4, 6 and 7 from nodes 5 and 8 is unidirectional because all the value and control dependences flow unidirectionally. The def-def chain flowing from node 5 to node 7 does *not* violate the unidirectionality of the cut.

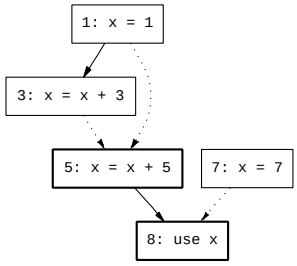

Figure 12: The du-web for variable x from the flow graph in Figure 11. The cut is unidirectional because all the def-use chains flow in one direction across the cut. Dotted edges show cut edges.

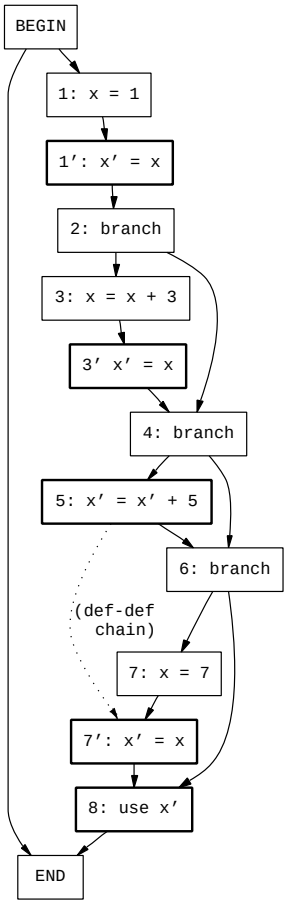

Figure 13: After unidirectionally renaming the variable x the def-def chain between nodes 5 and 7 is eliminated, and replaced instead with a def-def chain from node 5 to node 7'. The new def-def chain does not cross the cut because node 5 and 7' are both in the same partition (indicated by nodes with a bold outline).

direction).

Unidirectionally renaming the flow graph, as shown in Figures 13 and 14 solves this problem. After placing copy instructions "x′ = x" after the definitions that reach across the cut, and renaming x to x′ in nodes 5 and 7, *all* of the definitions of x are on one side of the cut while *all* of the definitions of x′ are on the other side of the cut. Thus there are *no* def-def chains flowing across the cut. All the def-def chains are now contained within one partition or the other.

Placing the copy instructions for the unidirectional renaming directly after the corresponding definition of each variable produces a correct result, but, in fact, we can do better. We can maintain the program semantics and eliminate the output dependences if we place the copy instructions along *any* set of edges in the program that have the property that they cover all the paths leading from definitions of x in A that reach uses of

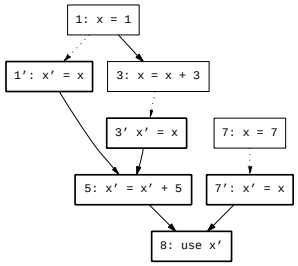

Figure 14: The du-web for variables x and x′ from the flow graph in Figure 13. The cut is still unidirectional because all the def-use chains flow in one direction across the cut. Dotted edges show cut edges. Now, however, there is no def-def chain crossing the cut because definitions of variable x happen in one partition, while definitions of variable x′ happen in the other.

x in B and are not reached by any of the definitions of x in B. Frank's dissertation describes how to derive such a set of edges that is optimal, in the sense that they will execute only as often as the innermost loop that contains both the definitions and the uses [16].

In this Section we have argued that given any unidirectional cut A-B we can insert copy instructions into each du-web that has edges flowing from A to B and derive a semantically equivalent flow graph with the property that there are no def-def chains flowing from B to A. There is a second benefit of performing unidirectional renaming on the du-webs that cross the cut. This is that after renaming, closure conversion and a single pass of local copy propagation, all the *uses* of a variable will be entirely contained on one side of the cut or the other. That is, all communication across the cut will occur through the deferred execution queue. There will be no "shared" scalar variables. Because of this property we perform unidirectional renaming on *all* du-webs that cross the cut, even when there are no def-def chains that need to be broken.

5 Implementation Experience

We have implemented scalar queue conversion in the context of SUDS [16], our research system for automatically parallelizing C programs running on the Raw microprocessor [35]. In this section we briefly describe how we used scalar queue conversion as the key subroutine of a generalized form of loop distribution that can reschedule *any* region of code with arbitrary control flow, including arbitrary looping control flow. In addition, we describe a set of other practical problems that needed to be solved in order to effectively find concurrency. Due to space limitations, the discussion is brief. The intent of this section is to complement our somewhat abstract description of scalar queue conversion with a discussion of its use in a practical setting. For details, refer to Frank's dissertation [16].

5.1 Generalized Loop Distribution

The goal of loop distribution is to transform the chosen region so that *any externally visible changes to machine state will occur in minimum time*. Roughly speaking, then, we begin by finding externally visible state changes for the region in question, which we call critical definitions. We then find the smallest partition of the value dependence graph that includes the critical node, yet still forms a unidirectional cut with its complement. Finally we apply scalar queue conversion to create a minimal (and hopefully small) piece of code that performs *only* the work that cyclically depends on each critical definition.

Consider again the example flow graph from Figure 2 used throughout Section 3. Roughly speaking, this loop has two loop carried dependences, on the variables i and sum. The other variables, (*e.g.*, j, partial_sum, c1 and c2) are private to each loop iteration, and thus are not part of the state changes visible external to the loop.

Following this intuitive distinction, we more concretely identify the *critical definitions* of a region by finding all uses (anywhere in the program) such that at least one definition d_R within the region R reaches the use and at least one definition from outside the region $d_{\bar{R}}$ reaches the use. Then we call the definition d_R (the one inside region R) a critical definition. To reiterate, intuitively, the critical definitions represent changes to the part of the state that is visible from outside the region. Critical definitions represent points inside the region at which that visible state is changed. (As opposed to region (loop) invariant and externally invisible (private) state).

For the region corresponding to the outer loop in Figure 2 the critical definitions are the nodes 11 and 12. Nodes 5, 6, 9 and 11, for example, are reached both by node 11 (inside the loop) and node 2 (outside the loop), so node 11 is a critical definition for the loop. Likewise, nodes 5, 6 and 12 are reached both by node 12 (inside the loop) and node 1 (outside the loop), so node 12 is also a critical definition for the loop.

Next, for each critical node we find all nodes in the value dependence graph that have a *cyclic dependence* with the critical node. That is, given critical node d and node n, if there is a path from d to n in the value dependence graph and a path from n to d in the value dependence graph, then we give n the same priority as d. For example, in the loop in Figure 2 the cyclic path 11 $\rightarrow$ 13 $\rightarrow$ 14 $\rightarrow$ 11 in the value dependence graph indicates that nodes 13 and 14 form a cycle with

the critical node 11. We assign the nodes in each cyclic critical dependence path to the same partition.

All remaining nodes will be assigned to a partition *between* two critical node partitions. That is, for each node n find the critical node d_{below} with the highest priority, such that there is a path from n to d_{below} in the value dependence graph. Then assign n to a partition between d_{below} and d_{below}'s parent.

For example, in Figure 2 node 12 depends on node 7. Node 7, in turn, is dependent on nodes 3, 4, 7, 8, 9 and 10. (There exists, for example, the dependence path $4 \rightarrow 8 \rightarrow 9 \rightarrow 10 \rightarrow 7$.) None of these nodes has a path in the value dependence graph leading to any of nodes 11, 13 or 14. Thus we give nodes 3, 4, 7, 8, 9 and 10 a priority between the priority of node 11 and the priority of node 12.

For each partition we have a unidirectional cut from the higher priorities to this partition and those below. Thus we perform scalar queue conversion on each partition (from the bottom up) to complete our code transformation.

5.2 Queue Management

The register renaming resources in any real system are of finite size, and thus need to be managed carefully so that they do not overflow. This problem appears in any system where queues are used, and there are several approaches to handling the problem. One approach to managing the deferred execution queues is to alternate execution between operations that increase the queue length (code from the backward slice) until the queue is nearly full, and then run code that decreases the queue length (closures from the forward slice) until the queue is empty. This is similar to the approach used to handle reading and writing from pipes in most UNIX file systems. In fact, in the degenerate case that the total space available for deferred execution queues is only as large as a single closure, this management scheme is equivalent to running the code in the order specified by the original flow graph. The SUDS compiler also strip mines the loop being distributed, which tends to reduce the probability of filling the deferred execution queues.

Generalized loop distribution introduces an additional subtlety, in that multiple queues need to be managed simultaneously. The basic idea of falling back to the sequential schedule given by the original flow graph still works, however, and thus the queues can be managed both correctly and efficiently [7, 6, 16].

5.3 Extensions

The transformation described in Section 3 applies only to single exit regions of a flow graph. Scalar queue conversion can be extended to work on multiple exit regions of a flow graph. Interestingly this extension can be effected by applying scalar queue conversion, itself, to separate the multi-exit region from its successors in the flow graph. The generalized loop distribution transformation described above, was also extended with a recurrence reassociation transformation [24].

5.4 Memory Dependences

Scalar queue conversion takes a conservative view of memory dependences by inserting edges in the value dependence graph for *all* load-after-store, store-after-load and store-after-store dependences. These, extra, conservative dependences may restrict the applicability of scalar queue conversion because they might create cycles in the value dependence graph across what would otherwise be unidirectional cuts. In practice we found it necessary to implement four additional transformations to reduce the impact of memory dependences. These include transforming the code by register promotion [10] and apply the transformation that Barua has called "equivalence class unification," [32, 4, 8]. In addition we do a simple form of array privatization, based on the scope information that is typically available in C programs.

Additionally, the SUDS runtime system implements a memory checkpoint repair mechanism, and a concurrency control mechanism based on basic timestamp ordering [5] that is able to detect memory references that violate the expected sequential order. Because of this support, we are able to *speculatively* eliminate many memory dependence back edges that would otherwise create cycles in the value dependence graph.

6 Related Work

The idea of renaming to reduce the number of storage dependences in the dependence graph has long been a goal of parallelizing and vectorizing compilers for Fortran [24]. The dynamic closure creation done by the queue conversion algorithm in Section 3 can be viewed as a generalization of earlier work in *scalar expansion* [24, 11]. Given a loop with an index variable and a well defined upper limit on trip count, scalar expansion turns each scalar referenced in the loop into an array indexed by the loop index variable. The queue conversion algorithm works in any code, even when there is no well defined index variable, and no way to statically determine an upper bound on the number of times the loops will iterate. Moreover, earlier methods of scalar expansion are heuristic. Queue conversion is the first compiler transformation that *guarantees* the elimination of all register storage dependences that

create cycles across what would otherwise be a unidirectional cut.

Given a loop containing arbitrary forward control flow, *loop distribution* [24] can reschedule that graph across a unidirectional cut [21, 17], but since loop distribution does no renaming, the unidirectional cut must be across the *conservative* program dependence graph (*i.e.*, including the register storage dependences). Queue conversion works across any unidirectional cut of the *value* dependence graph. Because scalar queue conversion always renames the scalars that would create register storage dependences, those dependences need not be considered during analysis or transformation. It is sometimes possible to perform scalar expansion before loop distribution, but loop distribution must honor any register storage dependences that are remaining.

Moreover, existing loop distribution techniques only handle arbitrary *forward* control flow inside the loop, and do so by creating arrays of predicates [21, 17]. The typical method is to create an array of three valued predicates for each branch contained in the loop. Then on each iteration of the top half of the loop a predicate is stored for each branch (*i.e.*, "branch went left", "branch went right" or "branch was not reached during this iteration"). Any code distributed across the cut tests the predicate for its closest containing branch. This can introduce enormous numbers of useless tests, at runtime, for predicates that are almost never true.

Queue conversion, on the other hand, creates and queues closures if and *only if* the dependent code is guaranteed to run. Thus, the resulting queues are (dynamically) often much smaller than the corresponding set of predicate arrays would be. More importantly, queue conversion works across inner loops. Further, because queue conversion allocates closures dynamically, rather than creating static arrays, it can handle arbitrary *looping* control flow, in either the outer or inner loops, even when there is no way to statically determine an upper bound on the number of times the loops will iterate.

Feautrier has generalized the notion of scalar expansion to the notion of array expansion [13]. As with scalar expansion, Feautrier's array expansion works only on structured loops with compile time constant bounds, and then only when the array indices are affine (linear) functions of the loop index variables. Feautrier's technique has been extended to the non-affine case [22], but only when the transformed array is not read within the loop (only written). The equivalence class unification and register promotion techniques mentioned in Section 5.4 extend scalar queue conversion to work with structured aggregates (*e.g.*, C `structs`), but not with arrays. Instead, our implementation of scalar queue conversion relies on the memory dependence speculation system mentioned in Section 5.4 to parallelize across array references (and even arbitrary pointer references).

The notion of a unidirectional cut defined in Section 3.1 is similar to the notion, from software engineering, of a static program slice. A *static program slice* is typically defined to be the set of textual statements in a program upon which a particular statement in the program text depends [38]. Program slices are often constructed by performing a backward depth first search in the value dependence graph from the nodes corresponding to the statements of interest[27]. This produces a unidirectional cut.

In Section 3.2 we proved that we could produce an executable control flow graph that includes exactly the nodes from the top of a unidirectional cut of the value dependence graph. Yang has proved the similar property, in the context of *structured* code, that an executable slice can be produced by eliding all the statements from the program text that are not in the slice [39]. Apparently it is unknown, given a program text with unstructured control flow, how to produce a control flow graph from the text, elide some nodes from the graph and then *accurately* back propagate the elisions to the program text [3]. Generalizations of Yang's result to unstructured control flow work only by inserting additional dependences into the value dependence graph [3, 9], making the resulting slices larger and less accurate. The proof in Section 3.2 demonstrates that when working directly with control flow graphs (rather than program texts) this extra work is unnecessary, even when the control flow is irreducible.

Further, executable program slicing only produces the portion of the program corresponding to partition A of a unidirectional cut A-B (that is, it only produces the *backward* executable slice). In Sections 3.3 and 4 we demonstrated how to queue and then resume a set of closures that reproduce the execution of partition B as well (the set of state transitions corresponding to the *forward* executable slice.

The reason queue conversion generalizes both loop distribution and executable program slicing is that *queue conversion makes continuations [34, 31, 2] explicit*. That is, any time we want to defer the execution of a piece of code, we simply create, and save, a closure that represents that code, plus the suspended state in which to run that code. It is standard to compile functional languages by making closures and continuations explicit [31, 2], but this set of techniques is relatively uncommon in compilers for imperative languages.

In fact, scalar queue conversion was anticipated by work from formal programming language semantics that demonstrates that continuation passing style rep-

resentations and SSA form flow graphs of imperative programs are semantically equivalent [20]. Based on this work, Appel has suggested that a useful way of viewing the ϕ nodes at the join points in SSA flow graphs is as the point in the program at which the actual parameters should be copied into the formal parameters of the closure representing the code dominated by the ϕ node [1]. This roughly describes what scalar queue conversion does.

That is, given a maximal group β containing a use of variable x for which we are going to create a closure, we rename x to x' (which can be viewed as the formal parameter). Then we introduce a new closure, containing the instruction $x' = x$, at the ϕ point which *shares* an environment containing x' with β. It is useful to view the new closure as simply copying the actual parameter, x, to the formal parameter x'.

A transformation similar to loop distribution, called *critical-path reduction* has been applied in the context of thread-level speculative systems [37, 33, 40]. Rather than distribute a loop into multiple loops, critical-path reduction attempts to reschedule the body of the loop so as to minimize the amount of code executed during an update to a critical node. While the transformation is somewhat different than that performed by loop distribution, loop distribution and critical-path reduction share the goal of trying to minimize the time observed to update state visible outside the loop body.

Schlansker and Kathail [28] have a critical-path reduction algorithm that optimizes critical paths in the context of superblock scheduling [18], a form of trace scheduling [15]. Vijaykumar implemented a critical-path reduction algorithm for the multiscalar processor that moves updates in the control flow graph [37]. Steffan *et al* have implemented a critical-path reduction algorithm based on Lazy Code Motion [23] that moves update instructions to their optimal point [33, 40]. As with previous loop distribution algorithms, none of these critical-path reduction algorithms can reschedule loops that contain inner loops.

7 Conclusion

This paper has given an informal description of the scalar queue conversion transformation. We have argued that scalar queue conversion can restructure any unidirectional cut of the true scalar dependences in any flow graph, and reschedule the code so that all of the instructions in the top half of the cut run (dynamically) before all of the instructions in the bottom half. Scalar queue conversion completely eliminates scalar anti- and output-dependences that might otherwise make this rescheduling impossible.

We have described the use of scalar queue conversion in one practical setting, as a subroutine for a generalized form of loop distribution that can reschedule loops with arbitrary control flow, including irreducible control flow and inner loops with trip counts that can not be determined until the loop exits. We believe that scalar queue conversion has many other applications as well. Our ongoing work involves applying scalar queue conversion to automate the process of hiding program slices for the purposes of software security (as in [41]), and applying scalar queue conversion as the basis of a code generator to convert imperative programs to executable data-flow graphs.

Acknowledgments

The authors thank the members of the Computer Architecture Group at MIT for creating an environment particularly conducive to research, and Anant Agarwal, Jon Babb, Sam Larsen, Walter Lee, Michael Taylor, and Bill Thies for a variety of conversations, observations, and suggestions directly relevant to this work. This work was financially supported by NSF and Darpa grants to the Fugu and Raw projects, the Industrial Technology Research Institute/Raw Project Collaboration, and the Electrical and Computer Engineering Department at the University of Illinois.

References

[1] A. W. Appel. SSA is functional programming. *SIGPLAN Notices*, 33(4), 1998.

[2] A. W. Appel and T. Jim. Continuation-passing, closure-passing style. In *Symp. Principles of Programming Languages*, 1989.

[3] T. Ball and S. Horwitz. Slicing programs with arbitrary control flow. In *Int'l. Workshop on Automated and Algorithmic Debugging*, 1993.

[4] R. Barua, W. Lee, S. P. Amarasinghe, and A. Agarwal. Maps: A compiler-managed memory system for Raw machines. In *Int'l. Symp. Computer Arch.*, 1999.

[5] P. A. Bernstein and N. Goodman. Timestamp-based algorithms for concurrency control in distributed database systems. In *Int'l. Conf. Very Large Data Bases*, 1980.

[6] G. E. Blelloch, P. B. Gibbons, and Y. Matias. Provably efficient scheduling for languages with fine-grained parallelism. *J. ACM*, 46(2):281–321, 1999.

[7] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. In *Symp. on Foundations of Computer Science*, 1994.

[8] M. Budiu and S. C. Goldstein. Optimizing memory accesses for spatial computation. In *Int'l. Symp. on Code Generation and Optimization*, Mar. 2003.

[9] J.-D. Choi and J. Ferrante. Static slicing in the presence of GOTO statements. *ACM Trans. Prog. Lang. Syst.*, 16(4):1097–1113, 1994.

[10] K. D. Cooper and J. Lu. Register promotion in C programs. In *Conf. Programming Language Design and Implementation*, 1997.

[11] R. Cytron and J. Ferrante. What's in a name? The value of renaming for parallelism detection and storage allocation. In *Int'l. Conf. Parallel Processing*, 1987.

[12] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Prog. Lang. Syst.*, 13(4):451–490, 1991.

[13] P. Feautrier. Array expansion. In *Int'l. Conf. on Supercomputing*, pages 429–441, July 1988.

[14] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Prog. Lang. Syst.*, 9(3):319–349, July 1987.

[15] J. A. Fisher. Trace Scheduling: A technique for global microcode compaction. *IEEE Trans. Comput.*, C-30(7):478–490, July 1981.

[16] M. I. Frank. *SUDS: Automatic Parallelization for Raw Processors.* PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, June 2003.

[17] B.-M. Hsieh, M. Hind, and R. Cryton. Loop distribution with multiple exits. In *Supercomputing*, 1992.

[18] W. W. Hwu, R. E. Hank, D. M. Gallagher, S. A. Mahlke, D. M. Lavery, G. E. Haab, J. C. Gyllenhaal, and D. I. August. Compiler technology for future microprocessors. *Proceedings of the IEEE*, 83(12):1625–1640, Dec. 1995.

[19] R. M. Keller. Look-ahead processors. *ACM Computing Surveys*, 7(4):177–195, Dec. 1975.

[20] R. A. Kelsey. A correspondence between continuation passing style and static single assignment form. In *ACM Workshop on Intermediate Representations*, 1995.

[21] K. Kennedy and K. S. McKinley. Loop distribution with arbitrary control flow. In *Supercomputing*, 1990.

[22] K. Knobe and V. Sarkar. Array SSA form and its use in parallelization. In *Symp. Principles of Programming Languages*, Jan. 1998.

[23] J. Knoop, O. Ruthing, and B. Steffen. Lazy code motion. In *Conf. Programming Language Design and Implementation*, 1992.

[24] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *Symp. Principles of Programming Languages*, 1981.

[25] P. J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6(4):308–320, 1964.

[26] T. Lengauer and R. E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Prog. Lang. Syst.*, 1(1):121–141, 1979.

[27] K. J. Ottenstein and L. M. Ottenstein. The program dependence graph in a software development environment. In *Symp. on Practical Software Development Environments*, 1984.

[28] M. Schlansker and V. Kathail. Critical path reduction for scalar programs. In *Int'l. Symp. on Microarchitecture*, 1995.

[29] J. E. Smith and A. R. Pleszkun. Implementing precise interrupts in pipelined processors. *IEEE Trans. Comput.*, 37(5):562–573, May 1988.

[30] G. S. Sohi. Instruction issue logic for high-performance, interruptible, multiple functional unit, pipelined computers. *IEEE Trans. Comput.*, 29(3):349–359, Mar. 1990.

[31] G. L. Steele. RABBIT: A compiler for Scheme. Technical Report AITR-474, MIT Artificial Intelligence Laboratory, May 1978.

[32] B. Steensgaard. Sparse functional stores for imperative programs. In *ACM Workshop on Intermediate Representations*, Jan. 1995.

[33] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. Improving value communication for thread-level speculation. In *Int'l. Symp. High Performance Computer Arch.*, Feb. 2002.

[34] C. Strachey and C. P. Wadsworth. Continuations: A mathematical semantics for handling full jumps. *Higher-Order and Symbolic Computation*, 13(1):135–152, Apr. 2000. (Republication of Oxford University Computing Laboratory Technical Monograph PRG-11, 1974).

[35] M. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, J.-W. Lee, P. Johnson, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal. The Raw microprocessor: A computational fabric for software circuits and general purpose programs. *IEEE Micro*, Mar. 2002.

[36] R. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development*, 11(1):25–33, Jan. 1967.

[37] T. N. Vijaykumar. *Compiling for the Multiscalar Architecture.* PhD thesis, University of Wisconsin-Madison Computer Sciences Department, Jan. 1998.

[38] M. Weiser. Program slicing. *IEEE Trans. Softw. Eng.*, 10(4):352–357, July 1984.

[39] W. Yang. *A New Algorlthm for Semantics-Based Program Integration.* PhD thesis, University of Wisconsin-Madison Computer Sciences Department, Aug. 1990.

[40] A. Zhai, C. B. Colohan, J. G. Steffan, and T. C. Mowry. Compiler optimization of scalar value communication between speculative threads. *Conf. Arch. Support for Programming Languages and Operating Systems*, 2002.

[41] X. Zhang and R. Gupta. Hiding program slices for software security. In *Int'l. Symp. on Code Generation and Optimization*, 2003.