Global Value Numbering

Taylor Simpson

August 18, 1994

[0]This work has been supported by ARPA, through ONR grant N00014-91-J-1989.

Contents

List of Figures

Chapter 1

Introduction

1.1 Overview

Value numbering is an optimization used to eliminate redundant computations from a routine. This document provides an implementation of the global value numbering algorithm described by Alpern, Wegman, and Zadeck [3]. It operates on the static single assignment (SSA) form of the routine [7]. In contrast to hash-based approaches, this technique partitions values into congruence classes. Two values are *congruent* if they are

1. Computed by the same opcode, and
2. Each of the corresponding operands are congruent.

Since the definition of congruence is recursive, there will be routines where the solution is not unique. A trivial solution would be to set each value in the routine to be congruent only to itself. However, the solution we seek is the *maximal fixed point* – the solution that contains the most congruent values.

We partition the values in the routine into congruence classes using a variation of the algorithm by Hopcroft for minimizing a finite-state machine [1]. After the partitioning is complete, we renumber the registers in the routine based on their class number. After the registers have been renumbered, there are three possibilities for removing redundant computations:

Dominator-Based Removal The technique suggested by Alpern, Wegman, and Zadeck is to remove computations that are dominated by another member of the congruence class [3]. This algorithm can be performed efficiently by bucket sorting based on the preorder index in the dominator tree and comparing adjacent pairs of members.

AVAIL-Based Removal The classical approach is to remove computations that are in the set of available expressions (AVAIL) at the point where they appear in the routine [2]. AVAIL is the set of expressions available along all paths from the start of the routine.

Partial Redundancy Elimination PRE is an optimization introduced by Morel and Renvoise [8]. Partially redundant computations are redundant along some, but not necessarily all, execution paths. In general, PRE moves code upward in the routine if at least one execution path profits from the move and no path suffers from it.

Notice that the options are presented in increasing order of effectiveness. It can be proved that each one is never worse than its predecessor.

- A computation that is dominated by another member of the congruence class must be in the AVAIL set because the dominating node is contained in all paths from the start of the routine. However, there are other ways for a computation to appear in the AVAIL set. If a computation appears in both branches of an if-then-else statement, then it will be in the AVAIL set at the join point even though neither of the branches dominates the join.

- A computation removed by the AVAIL method must be redundant along all execution paths; therefore, it will be removed by PRE. However, PRE can also move computations if at least one path profits and no path suffers.

1.2 Usage

gval can be invoked from the command line as follows:

`gval` [`-dhcparxst`] [*filename*]

gval reads `ILOC` input from *filename* or from `stdin` if no *filename* is specified and prints its output to `stdout`. The following options are available:

`-d` Increase the debug level. The default level is zero meaning no debugging output. The following levels are supported:

1. Indicate when the major phases of the program begin
2. Indicate when the minor phases of the program begin
3. Warn the user when there is a use of an uninitialized register
4. Show the routine in SSA form
5. Display the initial and final partitions

All debugging output is to `stderr`.

`-h` Print a histogram of the classes.

`-c` Remove comments from the `ILOC` file.

`-p` Prepare the output for partial redundancy elimination.

`-a` Remove operations based on AVAIL.

`-r` Remove operations based on dominators.

`-x` Handle commutative operations.

`-s` Eliminate redundant stores.

`-t` Print timing information for the major phases.

1.3 Performance

We ran **gval** using AVAIL-based and dominator-based removal on five routines from three programs in the SPEC benchmark. For each routine, we recorded the amount of time **gval** took in each of its major phases. We ran all of the experiments on an IBM RS/6000 Model 540, running at 30MHz, with a 64KB data cache. Figure 1.1 shows the number of basic blocks, registers and operations in each routine, as well as the time (in seconds) required by each phase. In all cases, partitioning was the most time comsuming phase of the program. Notice that the *Total* line is a bit deceiving because it includes time for both dominator and AVAIL-based removal.

Program	fpppp		tomcatv	doduc	
Routine	fpppp	twldrv	tomcatv	repvid	iniset
Blocks	2	333	91	127	446
Registers	9214	5902	946	831	2283
Operations	22351	16205	2690	1884	6731
Converting to SSA	0.40	0.71	0.06	0.04	0.23
Partitioning	1.24	9.78	0.08	0.05	0.28
Renumbering	0.04	0.04	0.01	0.00	0.02
Dominator-Based Removal	0.04	0.06	0.02	0.00	0.02
AVAIL-Based Removal	0.08	0.14	0.01	0.02	0.08
Converting from SSA	0.03	0.09	0.01	0.01	0.03
Total	1.83	10.82	0.19	0.12	0.66

Figure 1.1: Performance Data for **gval**.

1.4 Overall Structure

This is a simple boilerplate that specifies how the major components of the C program will fit together. We need include files for the shared library code and for the routines to handle SSA form.

"gval.c" 3 ≡
```
#include <Shared.h>
#include "SSA.h"

⟨Type Declarations 11a, ... ⟩
⟨Macros 5b, ... ⟩
⟨Global Variables 5a, ... ⟩
⟨Prototypes 85a, ... ⟩
⟨Functions 58c, ... ⟩
⟨The main routine 4⟩
◇
```

Option	Variable
-d	debug
-h	hist
-c	keep_comments (defined in the shared library)
-p	prepare_for_partial
-a	do_avail
-r	do_dominators
-x	do_commute
-s	elim_stores
-t	time_print (defined in the shared library)

Figure 1.2: Global Variables to Support Command Line Options

1.5 The Main Routine

The `main` routine shows the major steps of the program. The steps involved in reading in the `ILOC` file, converting to and from SSA form, and printing the optimized program will be explained in this section. The steps that are directly involved in global value numbering will be given detailed explanations in the remaining chapters of the document. The printing of the timing information is controlled by the `time_print` flag which will be turned on during parsing of the command line if the user specified the `-t` option. The variable `time` keeps track of the time spend along the way. The `Time_Dump` routine prints the timing information.

⟨The `main` routine 4⟩ ≡

```
Void main(Unsigned_Int argc, Char *argv[])
{
    Timer time = Time_Start();

    ⟨Parse the command line 5d⟩
    ⟨Read the ILOC file and convert to SSA form 7a⟩
    Time_Dump(time, "Converting to SSA required %1.2f seconds\n");
    ⟨Partition all the values in the routine into congruence classes 13c⟩
    Time_Dump(time, "Partitioning required %1.2f seconds\n");
    ⟨Optionally perform dominator-based removal 57a⟩
    Time_Dump(time, "Dominator-based removal required %1.2f seconds\n");
    ⟨Renumber the φ-nodes and registers based on the congruence classes 48⟩
    Time_Dump(time, "Renumbering required %1.2f seconds\n");
    ⟨Convert the routine out of SSA form 7c⟩
    Time_Dump(time, "Converting out of SSA required %1.2f seconds\n");
    ⟨Optionally perform AVAIL-based removal 62⟩
    Time_Dump(time, "AVAIL-based removal required %1.2f seconds\n");
    ⟨Print only those operations that have been marked critical 7d⟩
    Time_Dump(time, "Printing required %1.2f seconds\n");
    free(time);

    exit(0);
}
◇
```

Macro referenced in scrap 3.

We'll need some global variables to handle the command-line options. Figure 1.2 shows the variable that corresponds to each option.

⟨Global Variables 5a⟩ ≡

```
Unsigned_Int debug = 0;
static Boolean hist = FALSE;
static Boolean prepare_for_partial = FALSE;
static Boolean do_avail = FALSE;
static Boolean do_dominators = FALSE;
static Boolean do_commute = FALSE;
static Boolean elim_stores = FALSE;
```
◇

Macro defined by scraps 5ac, 6d, 7b, 11c, 13b, 28a, 36, 57c, 58a.
Macro referenced in scrap 3.

The debug variable has five significant values.

⟨Macros 5b⟩ ≡

```
#define MAJOR_PHASES          1
#define MINOR_PHASES          2
#define UNINITIALIZED_REG     3
#define SHOW_SSA              4
#define PARTITION             5
```
◇

Macro defined by scraps 5b, 6b, 12, 29c, 43d.
Macro referenced in scrap 3.

The global variable file_name will store the name of the input file. If the user specifies no input file, file_name will be left NULL, which tells Block_Init to use stdin.

⟨Global Variables 5c⟩ ≡

```
static Char *file_name = NULL;
```
◇

Macro defined by scraps 5ac, 6d, 7b, 11c, 13b, 28a, 36, 57c, 58a.
Macro referenced in scrap 3.

We parse the command line by looping over the entries in argv. The variable i is the current index into argv, and c_ptr is a pointer into argv[i]. We assume that any input beginning with the character '-' is a list of flags, and all others are file names. If more than one file name is entered, we assume the user entered something wrong. In this case, we print out the usage message and halt.

⟨Parse the command line 5d⟩ ≡

```
{
    Unsigned_Int i;
    for (i = 1; i < argc; i++)
    {
        Char *c_ptr = argv[i];

        if (*c_ptr++ == '-')
            ⟨Parse a list of flags 6a⟩
        else if (!file_name)
            file_name = argv[i];
        else
            ⟨Print the usage message and halt 6c⟩
    }
}
```
◇

Macro referenced in scrap 4.

We parse a list of flags by looking at one character at a time. For each valid flag, there is a corresponding global variable. If we hit a character which is not in the set of recognizable flags, we print out the usage message and halt.

⟨Parse a list of flags 6a⟩ ≡
```
while (*c_ptr)
    switch (*c_ptr++)
    {
        case 'd':    debug++;                         break;
        case 'h':    hist = TRUE;                     break;
        case 'c':    keep_comments = FALSE;           break;
        case 'p':    prepare_for_partial = TRUE;      break;
        case 'a':    do_avail = TRUE;                 break;
        case 'r':    do_dominators = TRUE;            break;
        case 'x':    do_commute = TRUE;               break;
        case 's':    elim_stores = TRUE;              break;
        case 't':    time_print = TRUE;               break;
        default:
            ⟨Print the usage message and halt 6c⟩
    }
◇
```
Macro referenced in scrap 5d.

If we encounter an error parsing the command line, we will print the usage string and halt the program. We use the `ABORT` macro, which is set up to dump core if we are running in debug mode and exit with an error code otherwise.

⟨Macros 6b⟩ ≡
```
#define USAGE_STRING          "Usage: %s [-dhcparxst] [filename]\n"
◇
```
Macro defined by scraps 5b, 6b, 12, 29c, 43d.
Macro referenced in scrap 3.

⟨Print the usage message and halt 6c⟩ ≡
```
{
    fprintf(stderr, USAGE_STRING, argv[0]);
    ABORT;
}
◇
```
Macro referenced in scraps 5d, 6a.

Once we have parsed the command line, we are ready to read in the `ILOC` file and convert it to SSA form. This conversion renames all the items in the routine so that there is a unique definition point for each item in the routine. During this phase, we also construct a table containing the definition points and a table of uses for each definition. The `ConvertToSSA` routine must have an `Arena` in which to allocate these tables and other data structures.

⟨Global Variables 6d⟩ ≡
```
static Arena SSA_arena;
◇
```
Macro defined by scraps 5ac, 6d, 7b, 11c, 13b, 28a, 36, 57c, 58a.
Macro referenced in scrap 3.

The Block_Init routine provided by the shared library will read the ILOC file and build a control-flow graph (CFG) [4]. The ConvertToSSA routine is provided in a companion to this document.

⟨Read the ILOC file and convert to SSA form 7a⟩ ≡
```
Block_Init(file_name);
Time_Dump(time, "Parsing required %1.2f seconds\n");
SSA_arena = Arena_Create();
ConvertToSSA(SSA_arena);
if (debug >= SHOW_SSA) ShowSSA();
◇
```
Macro referenced in scrap 4.

After we have renumbered the ϕ-nodes and registers based on the congruence classes, we will convert the routine out of SSA form. During this phase, we restore the original names to all the tags in the routine and insert copies for ϕ-nodes. The ConvertFromSSA routine will need to know the number of registers in the renumbered version of the routine in case it needs to create new register names. Notice that we do not use the register_count variable provided by the parser because the renumbering process invalidates this number. We'll keep track of the maximum register number using a variable called max_register. The actual value of max_register will be computed when we renumber the registers based on congruence classes. Since registers are numbered starting at zero, the num_registers variable will be one more than max_register.

⟨Global Variables 7b⟩ ≡
```
static Unsigned_Int max_register = 0;
static Unsigned_Int num_registers;
◇
```
Macro defined by scraps 5ac, 6d, 7b, 11c, 13b, 28a, 36, 57c, 58a.
Macro referenced in scrap 3.

The ConvertFromSSA routine is used to restore the original names to the tags and insert copies for ϕ-nodes. During the process of inserting copies for ϕ-nodes, some new register names may need to be invented. If so, register numbers will be allocated starting at num_registers, and the new value of num_registers will be returned. Once we have converted the routine out of SSA form, we no longer need the data structures so we destroy the arena that contains them. The new value of num_registers will be used as the size of the bit vectors used in data-flow analysis when we perform AVAIL-based removal of operations.

⟨Convert the routine out of SSA form 7c⟩ ≡
```
num_registers = max_register + 1;
num_registers = ConvertFromSSA(SSA_arena, num_registers);
Arena_Destroy(SSA_arena);
◇
```
Macro referenced in scrap 4.

The user can specify options that cause operations to be removed from the routine. During an early pass over the CFG, we will set the critical flag for all operations to TRUE. Later, when we discover that an operation can be removed, we will set its critical flage to FALSE. Setting the print_all_operations flag to FALSE causes the shared routine Block_Put_All to only print operations whose critical flag is set.

⟨Print only those operations that have been marked critical 7d⟩ ≡
```
print_all_operations = FALSE;
Block_Put_All(stdout);
◇
```
Macro referenced in scrap 4.

Chapter 2

Partitioning

We partition the values in the routine into congruence classes using a variation of the algorithm by Hopcroft for minimizing a finite-state machine [1]. In this document, we use the term *partition* to refer to a set of *congruence classes* such that each value (SSA name) in the routine is in exactly one class. The partitioning of values is accomplished by starting with an initial partition and iteratively refining the partition until it stabilizes. In the initial partition, all values defined by the same opcode are in the same congruence class. Of course, the set of values defined by some opcodes must be divided immediately. For example, the **FRAME** opcode defines a set of distinct values that are passed to the routine in registers. Therefore, all the values defined by the **FRAME** opcode are placed in different classes in the initial partition.

The partition is refined using a worklist algorithm. The variable called *worklist* will contain classes that might cause other classes to split. When a class is removed from *worklist*, we visit the uses of all its members. The *touched* set records the values defined by the uses that are visited at each iteration. Notice that we only visit uses in the same position because we must treat all opcodes as if they were not commutative. Commutative operations are handled by an extension to the partitioning algorithm described in Section 2.5. Any class s (s stands for split) with a proper subset of its members in *touched* ($\emptyset \subset (s \cap touched) \subset s$) must be split into two classes. We create a new class n containing the touched members of s, and we remove the members of n from s. Splitting class s may cause other classes in the partition to split, so we must update *worklist*. There are two cases to consider:

Case 1 If s was already in *worklist*, this means that the partition was not stable with respect to s, so we must leave s in *worklist* and also add n to *worklist*.

Case 2 If s was not in *worklist*, this means that the partition was stable with respect to s before it was split. Thus, any class that would be split as a result of removing s from *worklist* would be split the same way as a result of removing n from *worklist*. Since we are free to choose between s and n, we will select the one that can be processed in a smaller amount of time. Therefore, we add the smaller of s and n to *worklist*. As we shall see, this step is fundamentally important in achieving our desired running time.

Once we have refined the partition with respect to class c, we will not examine c again unless it is split. Pseudo-code for the partitioning algorithm is shown in Figure 2.1.

```
Place all values computed by the same opcode in the same congruence classes
worklist ← the classes in the initial partition
while worklist ≠ ∅
    Select and delete an arbitrary class c from worklist
    for each position p of a use of x ∈ c
        touched ← ∅
        for each x ∈ c
            Add all uses of x in position p to touched
        for each class s such that ∅ ⊂ (s ∩ touched) ⊂ s
            Create a new class n ← s ∩ touched
            s ← s − n
            if s ∈ worklist
                Add n to worklist
            else
                Add smaller of n and s to worklist
```

Figure 2.1: Partitioning Algorithm

$$\begin{aligned} A &\leftarrow X - Y \\ B &\leftarrow Y - X \\ C &\leftarrow A - B \\ D &\leftarrow B - A \end{aligned}$$

Figure 2.2: Partitioning Example

To illustrate how the algorithm operates, consider the code fragment in Figure 2.2. Figure 2.3 shows the initial partition if X is not congruent to Y ($X \ncong Y$). Notice that all names defined by the subtraction opcode are initially in the same class. Let the class containing X be the first removed from *worklist*. Touching the uses of X in position 1 will result in A being removed from its class. Touching the uses of X in position 2 will result in B being removed from its class. The partition after one iteration is shown in Figure 2.3. Let the class containing Y be the next one removed from *worklist*. Touching the uses of Y in position 1 will result in C being removed from its class. Touching the uses of Y in position 2 will not cause any classes to be split because the *touched* set is the entire class. Since all values are now in different classes, no further splitting can occur. However, since the algorithm has no way to determine this, it will not terminate until *worklist* is empty. The final partition is also shown in Figure 2.3.

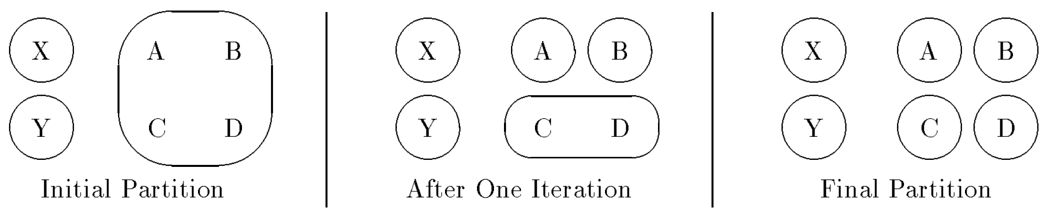

Figure 2.3: Partitioning Steps for Example Program

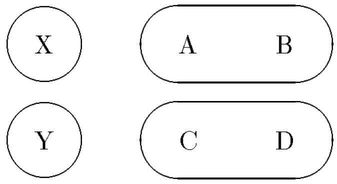

Figure 2.4: Incorrect Partition When Positions Are Ignored

To understand the importance of touching only uses in the same position, consider again the code fragment in Figure 2.2 and its initial partition in Figure 2.3. Touching all uses of X in any position will result in A and B being removed from their class and put into a new class together. If the algorithm continues in this fashion, no further splitting will occur. The final partition is shown in Figure 2.4. Notice that we have erroneously proven that $A \cong B$ and $C \cong D$.

2.1 Complexity

From the pseudo-code given in Figure 2.1, it is not clear what the running time of the algorithm will be. We claim that the partition can be computed in $O(E \log N)$ time, where E is the number of edges and N is the number of nodes in the routine's SSA graph[1]. However, only a careful implementation of the data structures will result in the desired time bound. To discover the operations our data structure must support and their complexities, we will analyze the algorithm from the bottom up. The statement in line 15 can be performed in constant time if we can determine the size of a class in constant time. Therefore, the **if** statement in lines 12–15 will require constant time. The splitting of a class in lines 10 and 11 can be performed in $O(\|n\|)$ time if we can remove an element from class s and insert it into class n in constant time[2]. The crucial part of the implementation is to perform the **for** loop in lines 9–15 in $O(\|touched\|)$ time. This will be discussed in detail in Section 2.4. The **for** loop in lines 7 and 8 can be performed in $O(\|c\|)$. We assume that the number of uses in any operation is bounded by a constant so the maximum position of a use is also bounded by a constant. Therefore, we can ignore the **for** loop starting on line 5.

Now we are ready to analyze the running time of the entire algorithm. Consider the number of times a class containing a particular element x can be removed from *worklist*. Each time such a class is chosen, it must be smaller than half the size of the last class to contain x removed from *worklist*. This is because we only add the smaller of n and s in line 15. Therefore, a class containing x can be removed from *worklist* at most $O(\log N)$ times. Suppose the cost of each execution of the **for** loop in lines 9–15 is charged to each x in c according to the number of uses of x in position p. Then the cost for each x is $O(\|\text{USES}(x)\| \log N)$, where $\text{USES}(x)$ is the set of uses of x. Since E is the set of all uses of any x, the total cost of the algorithm is $O(E \log N)$. Figure 2.5 summarizes the time complexity required for the various operations that will be performed on the partition.

2.2 Data Structures to Support Partitioning

To support the complexity requirements given in Figure 2.5, for each class we will maintain the number of members and a circular, doubly-linked list of members. We will also maintain a lookup table that will have a pointer to the list node and the class number of each element in the partition. This will enable us to locate and remove an element from a class in constant time. Figure 2.6 shows how the data structure would look assuming that the item with SSA name x_0 is in class 5.

[1] The SSA graph has a node for each definition and edges flow from definitions to uses.
[2] We use the notation $\|n\|$ to represent the size of set n.

Operation	Complexity
Determine the number of elements in a class	O(1)
Determine which class contains a name	O(1)
Remove a member of a class	O(1)
Add a member to a class	O(1)
Add a new class to the partition	O(1)
Iterate through the members of class c	O($\|\|c\|\|$)

Figure 2.5: Operations Supported by the Partition

Each node in the doubly-linked lists is represented by a `Member_Node` structure. Each one will contain `prev` and `next` pointers used by the list and the SSA name of the item. Other fields are used to implement extensions to the partitioning algorithm. They will be explained in Sections 2.5 and 2.6.

⟨Type Declarations 11a⟩ ≡

```
typedef struct member_node
{
    struct member_node *prev;
    struct member_node *next;
    Unsigned_Int2 item;
    ⟨Other Member_Node fields 43b⟩
} Member_Node;
◇
```

Macro defined by scraps 11ab, 13a, 22c, 29b, 43a, 57b, 63a.
Macro referenced in scrap 3.

The partition will be represented by an array indexed by class numbers. Each element will contain a list of the members of the class and the number of members in the class.

⟨Type Declarations 11b⟩ ≡

```
typedef struct
{
    Member_Node *members;
    Unsigned_Int2 num_members;
} Class;
◇
```

Macro defined by scraps 11ab, 13a, 22c, 29b, 43a, 57b, 63a.
Macro referenced in scrap 3.

The array is called `classes`. The variable `num_classes` keeps track of how many classes are currently in the partition. We use zero as a special value, so class numbers will start with one.

⟨Global Variables 11c⟩ ≡

```
static Class *classes;
static Unsigned_Int num_classes = 1;
◇
```

Macro defined by scraps 5ac, 6d, 7b, 11c, 13b, 28a, 36, 57c, 58a.
Macro referenced in scrap 3.

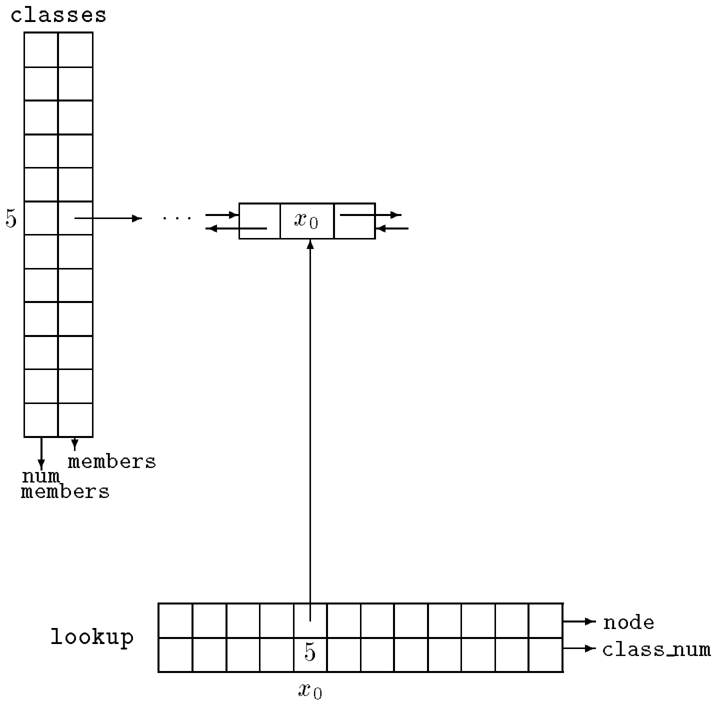

Figure 2.6: Data Structures for Representing the Partition

Now that we have the declared the type of a class, we can define a macro to iterate through the **members** list. It takes two arguments:

1. A pointer to a **Member_Node** structure (the iterator variable), and
2. A list of members of the class.

Each **members** list will be allocated with a “dummy” node to avoid having to check for **NULL** pointers at the end of the list. The **item** field of this node will be set to zero. Therefore, we start iterating with **(members)->next** and continue until we find a node containing a zero.

⟨Macros 12⟩ ≡

```
#define Class_ForAllMembers(node, members)      \
    for (node = (members)->next;                \
         node->item;                            \
         node = node->next)
```

◇

Macro defined by scraps 5b, 6b, 12, 29c, 43d.
Macro referenced in scrap 3.

In order to quickly locate a member of a class, we use a lookup table. The lookup table is an array indexed by SSA names. Each element contains a pointer to the `Member_Node` containing the item and the index of the class containing the item. Other fields are used to implement extensions to the partitioning algorithm. They will be explained in Sections 2.5 and 2.6.

⟨Type Declarations 13a⟩ ≡
```
typedef struct
{
    Member_Node *node;
    Unsigned_Int2 class_num;
    ⟨Other Lookup fields 35a, ... ⟩
} Lookup;
◇
```
Macro defined by scraps 11ab, 13a, 22c, 29b, 43a, 57b, 63a.
Macro referenced in scrap 3.

⟨Global Variables 13b⟩ ≡
```
static Lookup *lookup;
◇
```
Macro defined by scraps 5ac, 6d, 7b, 11c, 13b, 28a, 36, 57c, 58a.
Macro referenced in scrap 3.

Partitioning is a two step process.

1. Initialize the partition by placing all items defined by the same opcode into the same class.
2. Iteratively refine the partition until it stabilizes.

Along the way, we will print any debugging information requested by the user. Notice that we use two arenas for partitioning. The `partition_arena` will be used to hold data structures that live throughout the partitioning process. The `temp_arena` will be used to hold short lived data structures that will be freed using `Arena_Mark` and `Arena_Release`.

⟨Partition all the values in the routine into congruence classes 13c⟩ ≡
```
{
    Arena partition_arena = Arena_Create();
    Arena temp_arena = Arena_Create();

    if (debug >= MAJOR_PHASES)
        fprintf(stderr, "Partition all the values in the routine\n");

    ⟨Create the initial partition 14a⟩
    ⟨Optionally print the initial partition 74c⟩
    ⟨Refine the partition 27a⟩
    ⟨Optionally print the final partition 74d⟩
    ⟨Optionally print a histogram 72⟩

    Arena_Destroy(partition_arena);
    Arena_Destroy(temp_arena);
}
◇
```
Macro referenced in scrap 4.

2.3 Initializing the Partition

The process of initializing the partition has four steps:

1. Initialize the partition by allocating the `classes` and `lookup` arrays.
2. Create a class for each of the pseudo-definitions created for the tags during the conversion to SSA form. We must assume that each tag has a different value at the start of the routine, so each one is given its own congruence class.
3. Create a class for the ϕ-nodes in each block. The semantics of a ϕ-node in block b is to select the argument that corresponds to the predecessor of b from which control is transferred to b. Thus, ϕ-nodes in different blocks can never be congruent. However, ϕ-nodes in the same block with congruent arguments should be considered congruent. Therefore, for each block in the routine, we create a class for its ϕ-nodes.
4. Create classes for the items defined by operations. We initially make the optimistic assumption that all items defined by the same opcode are congruent. However, some types of opcodes cause more than one class to be added to the initial partition.

⟨Create the initial partition 14a⟩ ≡

```
if (debug >= MINOR_PHASES)
    fprintf(stderr, "    Create the initial partition\n");

⟨Initialize the partition 14b⟩
⟨Create a class for each of the pseudo-definitions 16a⟩
⟨Create a class for the φ-nodes in each block 16b⟩
⟨Create classes for the items defined by operations 17c⟩
◇
```

Macro referenced in scrap 13c.

The `classes` array is indexed by class numbers, so we must allocate enough entries for the maximum number of classes. This number will be reached if every SSA name in the routine is in a separate class. Therefore, the `classes` array will have `defCount` entries. The `lookup` array is indexed by SSA names, so it will also have `defCount` entries.

⟨Initialize the partition 14b⟩ ≡

```
classes = Arena_GetMem(partition_arena, defCount*sizeof(Class));
lookup = Arena_GetMem(SSA_arena, (defCount + 1)*sizeof(Lookup));
⟨Fill in lookup[0] 14c⟩
◇
```

Macro referenced in scrap 14a.

During the conversion to SSA form, we use the value zero to represent undefined arguments to ϕ-nodes. When we renumber the registers in the routine, we must maintain this convention. Therefore, we will force zero to map to zero.

⟨Fill in `lookup[0]` 14c⟩ ≡

```
lookup[0].class_num = 0;
lookup[0].node = NULL;
◇
```

Macro referenced in scrap 14b.

We create a new class containing a single element by adding a list of length one to the `classes` array and updating the appropriate `lookup` array entry. Recall that there are other fields in the `Lookup` structure used for extensions to the partitioning algorithm. They will be explained in Sections 2.5 and 2.6.

⟨Create a new class for `node` 15a⟩ ≡
```
{
    Unsigned_Int item = node->item;
    Member_Node *members;

    ⟨Create a new members list 15b⟩
    ⟨Append node to the members list 15c⟩
    classes[num_classes].members = members;
    classes[num_classes].num_members = 1;
    lookup[item].class_num = num_classes++;
    lookup[item].node = node;
    ⟨Initialize other Lookup fields 35b, ... ⟩
}
◇
```
Macro referenced in scraps 16a, 20a, 23a, 25a, 26b, 46b.

Each `members` list is a circular, doubly-linked list. We allocate a dummy node for each list to avoid having to check for `NULL` pointers later.

⟨Create a new `members` list 15b⟩ ≡
```
{
    Member_Node *node = members = Arena_GetMem(partition_arena, sizeof(Member_Node));

    node->prev = node;
    node->next = node;
    node->item = 0;
    ⟨Initialize other Member_Node fields 43c⟩
}
◇
```
Macro referenced in scraps 15a, 16b, 18b, 32b, 38b, 39b.

Since each list is circular, we can append to the list by putting the new node before the dummy node.

⟨Append `node` to the `members` list 15c⟩ ≡
```
node->next = members;
node->prev = members->prev;
members->prev = node;
node->prev->next = node;
◇
```
Macro referenced in scraps 15a, 17a, 24b, 32c, 39ac.

During conversion into SSA form, we created pseudo-definitions for all the tags in the routine. They are numbered from one to `tag_count`. We place each of them in a different congruence class by allocating a `Member_Node`, initializing it, and creating a new class for it. Recall that there are other fields in the structure used for extensions to the partitioning algorithm. They will be explained in Sections 2.5 and 2.6.

⟨Create a class for each of the pseudo-definitions 16a⟩ ≡

```
{
    Unsigned_Int i;
    for (i = 1; i <= tag_count; i++)
    {
        Member_Node *node = Arena_GetMem(partition_arena, sizeof(Member_Node));

        node->item = i;
        ⟨Initialize other Member_Node fields 43c⟩
        ⟨Create a new class for node 15a⟩
    }
}
◇
```

Macro referenced in scrap 14a.

In the initial partition, all items defined by ϕ-nodes in the same block are put into the same congruence class. For each block, we create a list of the items defined by ϕ-nodes. We then add this list to the partition.

⟨Create a class for the ϕ-nodes in each block 16b⟩ ≡

```
{
    Block *block;
    ForAllBlocks(block)
        if (Block_HasPhiNodes(block))
        {
            Member_Node *members;
            Unsigned_Int num_members = 0;
            PhiNode *phi_node;

            ⟨Create a new members list 15b⟩
            Block_ForAllPhiNodes(phi_node, block)
            {
                Unsigned_Int item = phi_node->newName;
                ⟨Add item to the members list 17a⟩
            }
            ⟨Add this class to the partition 17b⟩
        }
}
◇
```

Macro referenced in scrap 14a.

To add an item to a list, we create a node for it and append the node to the list. We must also set the fields of the `lookup` array. Recall that the `Member_Node` and `Lookup` structures have other fields used for extensions to the partitioning algorithm. These will be explained in Sections 2.5 and 2.6.

⟨Add `item` to the `members` list 17a⟩ ≡

```
{
    Member_Node *node = Arena_GetMem(partition_arena, sizeof(Member_Node));

    node->item = item;
    ⟨Initialize other Member_Node fields 43c⟩
    ⟨Append node to the members list 15c⟩
    num_members++;
    lookup[item].class_num = num_classes;
    lookup[item].node = node;
    ⟨Initialize other Lookup fields 35b, ... ⟩
}
◇
```

Macro referenced in scraps 16b, 19b, 20a.

Once the `members` list has been built, we can add the class to the partition.

⟨Add this class to the partition 17b⟩ ≡

```
classes[num_classes].members = members;
classes[num_classes++].num_members = num_members;
◇
```

Macro referenced in scrap 16b.

Creating classes for the items defined by operations is a two step process:

1. Find the names defined by each type of opcode in the routine. We iterate over the operations in the routine and keep a `Class` containing the items defined by each opcode. The variable `op_classes` records the set of names defined by each type of opcode.
2. Add one or more classes for each type of opcode found in the routine. Once we have a list of items defined by each opcode, we can add the appropriate classes to the initial partition. Some of the lists constitute a class while others must be further subdivided.

We use `Arena_Mark` and `Arena_Release` to reclaim the space allocated for `op_classes`.

⟨Create classes for the items defined by operations 17c⟩ ≡

```
{
    Class *op_classes;

    Arena_Mark(temp_arena);
    ⟨Find the names defined by each opcode in the routine 18a⟩
    ⟨Add a class for each type of opcode found in the routine 20b⟩
    Arena_Release(temp_arena);
}
◇
```

Macro referenced in scrap 14a.

Now we are ready to iterate over all the operations in the routine and build the list of items defined by each opcode. We'll use an array indexed by the opcode to hold the lists. It is allocated using `Arena_GetMemClear` so that all the members lists will be `NULL` and `num_members` will be zero.

Since we may delete operations later by marking them not `critical`, we must first mark all operations `critical`. This is a convenient place to do it.

⟨Find the names defined by each opcode in the routine 18a⟩ ≡
```
op_classes = Arena_GetMemClear(temp_arena, number_of_opcodes*sizeof(Class));
{
    Block *block;
    ForAllBlocks(block)
    {
        Inst *inst;
        Block_ForAllInsts(inst, block)
        {
            Operation **oper_ptr;
            Inst_ForAllOperations(oper_ptr, inst)
            {
                Operation *oper = *oper_ptr;
                Opcode_Names opcode = oper->opcode;

                oper->critical = TRUE;

                ⟨If there is not already a class for opcode, create one 18b⟩
                ⟨Add all the items defined to op_classes[opcode] 19a⟩
            }
        }
    }
}
◇
```
Macro referenced in scrap 17c.

We must create a list for `op_classes[opcode]` before we can begin inserting items into it.

⟨If there is not already a class for `opcode`, create one 18b⟩ ≡
```
if (!op_classes[opcode].members)
{
    Member_Node *members;

    ⟨Create a new members list 15b⟩
    op_classes[opcode].members = members;
}
◇
```
Macro referenced in scrap 18a.

Notice the special handling of scalar loads and stores. This is an extension to the partitioning algorithm that will be explained in Section 2.6.

⟨Add all the items defined to `op_classes[opcode]` 19a⟩ ≡
```
{
    Unsigned_Int details = opcode_specs[opcode].details;

    if (elim_stores &&
        details & LOAD && details & SCALAR)
        ⟨Add a scalar load operation to the initial partition 45a⟩
    else if (elim_stores &&
             details & STORE && details & SCALAR)
        ⟨Add a scalar store operation the the initial partition 45c⟩
    else
        ⟨Handle an operation that is not under redundant-store elimination 19b⟩
}
◇
```
Macro referenced in scrap 18a.

Even if the user has not requested redundant-store elimination, we still must treat stores specially to handle aliasing. For operations that are not stores, we put all the defined registers and tags in `op_classes[opcode]`. We count how many items were added by this operation and update the `num_members` field at the end.

⟨Handle an operation that is not under redundant-store elimination 19b⟩ ≡
```
if (details & STORE)
    ⟨Add a store to the initial partition 20a⟩
else
{
    Member_Node *members = op_classes[opcode].members;
    Unsigned_Int num_members = 0;
    Unsigned_Int2 *item_ptr;

    Operation_ForAllDefs(item_ptr, oper)
    {
        Unsigned_Int item = *item_ptr;
        ⟨Add item to the members list 17a⟩
    }
    Operation_ForAllDefTags(item_ptr, oper)
    {
        Unsigned_Int item = *item_ptr;
        ⟨Add item to the members list 17a⟩
    }

    op_classes[opcode].num_members += num_members;
}
◇
```
Macro referenced in scrap 19a.

In `ILOC`, stores have a list of defined tags. The first tag in this list is the tag that is definitely being stored by the operation. All others are possible aliases of the first tag. Therefore, the first tag is handled normally by the partitioning algorithm, but the aliases must be placed into separate congruence classes.

⟨Add a store to the initial partition 20a⟩ ≡

```
{
    Member_Node *members = op_classes[opcode].members;
    Unsigned_Int num_members = 0;
    Unsigned_Int2 *item_ptr;
    Boolean first = TRUE;

    Operation_ForAllDefTags(item_ptr, oper)
    {
        Unsigned_Int item = *item_ptr;
        if (first)
        {
            first = FALSE;
            ⟨Add item to the members list 17a⟩
        }
        else
        {
            Member_Node *node = Arena_GetMem(partition_arena, sizeof(Member_Node));

            node->item = item;
            ⟨Initialize other Member_Node fields 43c⟩
            ⟨Create a new class for node 15a⟩
        }
    }

    op_classes[opcode].num_members += num_members;
}
◇
```

Macro referenced in scrap 19b.

Once we have found the names defined by each opcode and stored this information in the `op_classes` array, we are ready to add a class to the partition for each type of opcode. Notice that only those operations that define values (*i.e.,* `op_classes[opcode].num_members` $\neq 0$) are given a partition.

⟨Add a class for each type of opcode found in the routine 20b⟩ ≡

```
{
    Opcode_Names opcode;
    for (opcode = 0; opcode < number_of_opcodes; opcode++)
        if (op_classes[opcode].num_members)
        {
            Member_Node *members = op_classes[opcode].members;
            Unsigned_Int num_members = op_classes[opcode].num_members;

            ⟨Add the set of items defined by opcode to the partition 21a⟩
        }
}
◇
```

Macro referenced in scrap 17c.

Some opcodes require special handling which may further subdivide the set.

⟨Add the set of items defined by `opcode` to the partition 21a⟩ ≡
```
switch (opcode)
{
    ⟨Add the set of items defined by load immediate, add immediate, or shift immediate 21b⟩
    ⟨Add the set of items defined by loads and stores from memory 22a⟩
    ⟨Add the set of items defined by FRAME and JSR 22b⟩
    default:
        ⟨Add all the members to the same congruence class 26c⟩
        break;
}
◇
```
Macro referenced in scrap 20b.

Load immediate, add immediate, and shift immediate operations have a single constant argument found in position 0 of their **arguments** array. The class must be subdivided according to the value. The scrap that handles this situation will be used more than once for arguments in different positions. Therefore, we use the **arg_pos** variable to parametrize the scrap so it can find the position of the argument.

⟨Add the set of items defined by load immediate, add immediate, or shift immediate 21b⟩ ≡
```
case iLDI: case fLDI: case dLDI: case cLDI: case qLDI:
case iADDI: case iSUBI:
case iSLI: case iSRI: case lSLI: case lSRI:
    {
        Unsigned_Int arg_pos = 0;
        ⟨Handle an opcode with one constant argument 23a⟩
    }
    break;
◇
```
Macro referenced in scrap 21a.

Loads and stores from memory come in two flavors. The "`or`" addressing mode defines the address as the sum of a base register and a constant offset. A constant alignment value is also specified, so these operations contain two constant arguments. The "`rr`" addressing mode defines the address as the sum of a base register and an index register. A constant alignment value is specified, so these operations contain one constant argument located in position 1 of the `arguments` list. The class must be subdivided according to the value. The scrap that handles this situation will be used more than once for arguments in different positions. Therefore, we use the `arg_pos` variable to parametrize the scrap so it can find the position of the argument.

⟨Add the set of items defined by loads and stores from memory 22a⟩ ≡

```
case bCONor: case iCONor: case fCONor: case dCONor: case cCONor: case qCONor:
case bSLDor: case iSLDor: case fSLDor: case dSLDor: case cSLDor: case qSLDor:
case bSSTor: case iSSTor: case fSSTor: case dSSTor: case cSSTor: case qSSTor:
case bLDor: case iLDor: case fLDor: case dLDor: case cLDor: case qLDor:
case bSTor: case iSTor: case fSTor: case dSTor: case cSTor: case qSTor:
    ⟨Handle an opcode with two constant arguments 25a⟩
    break;

case bSLDrr: case iSLDrr: case fSLDrr: case dSLDrr: case cSLDrr: case qSLDrr:
case bSSTrr: case iSSTrr: case fSSTrr: case dSSTrr: case cSSTrr: case qSSTrr:
case bLDrr: case iLDrr: case fLDrr: case dLDrr: case cLDrr: case qLDrr:
case bSTrr: case iSTrr: case fSTrr: case dSTrr: case cSTrr: case qSTrr:
    {
        Unsigned_Int arg_pos = 1;
        ⟨Handle an opcode with one constant argument 23a⟩
    }
    break;
◇
```

Macro referenced in scrap 21a.

The `FRAME` and `JSR` opcodes define a set of distinct values. Therefore, each item defined by one of these opcodes must be placed into a separate congruence class.

⟨Add the set of items defined by `FRAME` and `JSR` 22b⟩ ≡

```
case FRAME:
case JSRr: case iJSRr: case fJSRr: case dJSRr: case cJSRr: case qJSRr:
case JSRl: case iJSRl: case fJSRl: case dJSRl: case cJSRl: case qJSRl:
    ⟨Place each item into a separate congruence class 26b⟩
    break;
◇
```

Macro referenced in scrap 21a.

Operations with one or more constant arguments require that their congruence classes be subdivided according to the values of these arguments. We use a linked list to keep track of which constants are referenced in operations. If we expect alot of constants to appear in the same operation type, we might switch to a more sophisticated data structure.

⟨Type Declarations 22c⟩ ≡

```
typedef struct const_node
{
    Expr value;               /*  The constant value represented  */
    Expr value2;              /*  A possible second value  */
    Unsigned_Int class_num;   /*  The class number for items that use value  */
    struct const_node *next;  /*  A pointer to the next element in the list  */
} Const_Node;
◇
```

Macro defined by scraps 11ab, 13a, 22c, 29b, 43a, 57b, 63a.
Macro referenced in scrap 3.

This scrap is used to add operations with one constant argument (e.g., `LDI`, `SSTrr`) to the partiton. We must further subdivide the set by the constant values being used. The `arg_pos` variable tells us which element of the defining operation's arguments array to look at. We use a different `constant_list` each time because the same value can be used in more than one type (e.g., integer and double). We use `Arena_Mark` and `Arena_Release` to reclaim the memory allocated for `constant_list`.

⟨Handle an opcode with one constant argument 23a⟩ ≡

```
{
    Const_Node *constant_list = NULL;

    Arena_Mark(temp_arena);

    while (num_members)
    {
        Member_Node *node = members->next;
        Expr value = DefiningOper(node->item)->arguments[arg_pos];
        Boolean found = FALSE;

        ⟨Delete node from its current list 23b⟩
        num_members--;

        ⟨Search constant_list for value 24a⟩
        if (!found)
        {
            ⟨Add value to constant_list 24c⟩
            ⟨Create a new class for node 15a⟩
        }
    }

    Arena_Release(temp_arena);
}
◇
```

Macro referenced in scraps 21b, 22a.

To delete an element e from a doubly-linked list, we change the `prev` pointer of e's `next` element, and we change the `next` pointer of e's `prev` element.

⟨Delete node from its current list 23b⟩ ≡

```
node->next->prev = node->prev;
node->prev->next = node->next;
◇
```

Macro referenced in scraps 23a, 25a, 26b, 32c, 39ac.

Since we are handling operations with a single constant argument, we only consider the **value** field of the **Const_Node** structure. We ignore the **value2** field. If **value** is found in the list, we will add the node to the class number indicated.

⟨Search constant_list for value 24a⟩ ≡

```
{
    Const_Node *const_node = constant_list;
    while (const_node)
    {
        if (const_node->value == value)
        {
            Unsigned_Int class_num = const_node->class_num;

            ⟨Add node to classes[class_num] 24b⟩
            found = TRUE;
            break;
        }
        const_node = const_node->next;
    }
}
◇
```

Macro referenced in scrap 23a.

When adding a member to a class, we must also update the **lookup** entry.

⟨Add node to classes[class_num] 24b⟩ ≡

```
{
    Member_Node *members = classes[class_num].members;

    ⟨Append node to the members list 15c⟩
    classes[class_num].num_members++;
    lookup[node->item].class_num = class_num;
}
◇
```

Macro referenced in scraps 24a, 25b.

If **value** is not found in **constant_list** we will add it. The item will be given **num_classes** as its class number.

⟨Add value to constant_list 24c⟩ ≡

```
{
    Const_Node *const_node = Arena_GetMem(temp_arena, sizeof(Const_Node));

    const_node->next = constant_list;
    const_node->value = value;
    const_node->class_num = num_classes;
    constant_list = const_node;
}
◇
```

Macro referenced in scrap 23a.

This scrap is used to add operations with two constant arguments (e.g., `CONor`, `SSTor`) to the partiton. We must further subdivide the set by the constant values being used. All operations with two constant arguments store them in positions one and two of their `arguments` array. We use a different `constant_list` each time because the same values can be used in more than one type (e.g. integer and double). We use `Arena_Mark` and `Arena_Release` to reclaim the memory allocated for `constant_list`.

⟨Handle an opcode with two constant arguments 25a⟩ ≡

```
{
    Const_Node *constant_list = NULL;

    Arena_Mark(temp_arena);

    while (num_members)
    {
        Member_Node *node = members->next;
        Unsigned_Int item = node->item;
        Expr value1 = DefiningOper(item)->arguments[1];
        Expr value2 = DefiningOper(item)->arguments[2];
        Boolean found = FALSE;

        ⟨Delete node from its current list 23b⟩
        num_members--;

        ⟨Search constant_list for value1 and value2 25b⟩
        if (!found)
        {
            ⟨Add (value1, value2) to constant_list 26a⟩
            ⟨Create a new class for node 15a⟩
        }
    }

    Arena_Release(temp_arena);
}
◇
```

Macro referenced in scrap 22a.

⟨Search `constant_list` for `value1` and `value2` 25b⟩ ≡

```
{
    Const_Node *const_node = constant_list;
    while (const_node)
    {
        if (const_node->value == value1 && const_node->value2 == value2)
        {
            Unsigned_Int class_num = const_node->class_num;

            ⟨Add node to classes[class_num] 24b⟩
            found = TRUE;
            break;
        }
        const_node = const_node->next;
    }
}
◇
```

Macro referenced in scrap 25a.

⟨Add (value1, value2) to constant_list 26a⟩ ≡

```
{
    Const_Node *const_node = Arena_GetMem(temp_arena, sizeof(Const_Node));

    const_node->next = constant_list;
    const_node->value = value1;
    const_node->value2 = value2;
    const_node->class_num = num_classes;
    constant_list = const_node;
}
◇
```

Macro referenced in scrap 25a.

Certain `ILOC` opcodes define more than one value (e.g., `FRAME`, `JSR`). When an operation of this type appears in the routine, we must assume that all the defined items are different. Therefore, they are all placed into a separate congruence class. This is accomplished by removing them from the `members` list one at a time and creating a new congruence class for each one.

⟨Place each item into a separate congruence class 26b⟩ ≡

```
while (num_members)
{
    Member_Node *node = members->next;

    ⟨Delete node from its current list 23b⟩
    num_members--;
    ⟨Create a new class for node 15a⟩
}
◇
```

Macro referenced in scrap 22b.

For the majority of operations in the routine, we will put all the defined items into the same congruence class. To accomplish this, we must put the class number in the `lookup` entry for each of the members.

⟨Add all the members to the same congruence class 26c⟩ ≡

```
{
    Member_Node *node;

    classes[num_classes].members = members;
    classes[num_classes].num_members = num_members;

    Class_ForAllMembers(node, members)
        lookup[node->item].class_num = num_classes;

    num_classes++;
}
◇
```

Macro referenced in scrap 21a.

2.4 Refining the Partition

We use a worklist algorithm to iteratively refine the partition. Since **worklist** is indexed by class numbers, we create a set with universe size equal to the maximum number of classes. This number will be reached if every SSA name in the routine is in a separate class. Therefore, **worklist** will have a universe size of **defCount**. Initially, **worklist** contains the classes in the initial partition. At each step in the process, a class is removed from **worklist**, and all uses of members of the class are touched. Any class with a proper subset of its members touched must be split. Finally, we must prepare the partition for redundant-store elimination. This will be explained in Section 2.6.

⟨Refine the partition 27a⟩ ≡
```
{
    SparseSet worklist = SparseSet_Create(partition_arena, defCount);

    if (debug >= MINOR_PHASES)
        fprintf(stderr, "    Refine the partition\n");

    ⟨Insert all the initial classes into worklist 27b⟩
    ⟨Initialize the data structures for refining the partition 28b⟩
    while (SparseSet_Size(worklist))
    {
        Unsigned_Int class_num = SparseSet_ChooseMember(worklist);
        Member_Node *members = classes[class_num].members;
        Unsigned_Int num_members = classes[class_num].num_members;

        SparseSet_Delete(worklist, class_num);
        ⟨Touch all uses of elements in members 29a⟩
    }
    ⟨Prepare the partition for redundant-store elimination 47b⟩
}
◇
```
Macro referenced in scrap 13c.

⟨Insert all the initial classes into **worklist** 27b⟩ ≡
```
{
    Unsigned_Int i;
    for (i = 1; i < num_classes; i++)
        SparseSet_Insert(worklist, i);
}
◇
```
Macro referenced in scrap 27a.

Recall that in our initial presentation of the partitioning algorithm (Figure 2.1) we used a set called *touched*. This set contained all uses of members of a class in some position. We then searched for classes with a proper subset of their members in *touched* and split them[3]. An implementation of the algorithm must employ an efficient technique for determining which classes to split and which members to remove at each iteration. In Section 2.2, we claimed that this step could be performed in $O(\|touched\|)$ time. To accomplish this, we do not represent the *touched* set itself. Instead, we keep the intersection of *touched* and each class in the partition. This information is kept in an array of **Class** structures called **intersections**. Each element of **intersections** will contain the intersection of *touched* with the corresponding element of **classes**. We also use a **SparseSet** called **classes_to_split** to keep track of which classes have a non-empty **intersections** entry. When an item is touched during the process of refining the partition, it is moved out of its class and into the **intersections** array. Recall that we are only interested in classes with a proper subset of their members touched. Therefore, if all members of a class are touched, they will be returned to their original class and the class will be deleted from **classes_to_split**.

[3] In other words, we split all classes s such that $\emptyset \subset (s \cap touched) \subset s$.

Figure 2.7 depicts an example partition with SSA name x_0 is in class 5. If x_0 is touched, it will be removed from the **members** list of **classes[5]** and appended to the **members** list of **intersections[5]**. If x_0 is the first item placed in **intersections[5]**, then 5 will be inserted into the **classes_to_split** set. Once all the uses in position p have been touched, we can determine which classes must be split by iterating through each class s in the **classes_to_split** set. We have already removed the necessary members from **classes**[s] and placed them in **intersections**[s]. Thus, we can simply create a new class containing the members of **intersections**[s] and iterate through the members list, updating the **class_num** field of the **lookup** array. The entire process requires O($\|touched\|$) time.

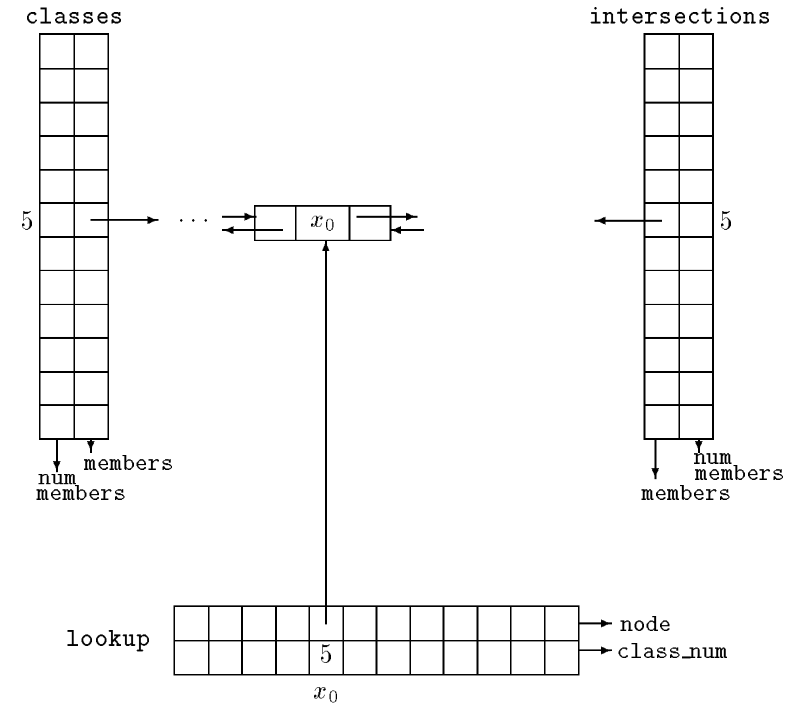

Figure 2.7: Data Structures for Refining the Partition

⟨Global Variables 28a⟩ ≡
```
static Class *intersections;
static SparseSet classes_to_split;
```
◇

Macro defined by scraps 5ac, 6d, 7b, 11c, 13b, 28a, 36, 57c, 58a.
Macro referenced in scrap 3.

Since the array and set are indexed by class numbers, they are allocated with **defCount** entries. We will also initialize the data structures for handling commutative operations. This will be explained in Section 2.5.

⟨Initialize the data structures for refining the partition 28b⟩ ≡
```
intersections = Arena_GetMem(partition_arena, defCount*sizeof(Class));
classes_to_split = SparseSet_Create(partition_arena, defCount);
```
⟨Initialize the data structures for commutative operations 37a⟩
◇

Macro referenced in scrap 27a.

Now we are ready to look at the heart of the partitioning algorithm. The first step is to bucket sort the uses of members of the class by their position. For each position, we touch the uses in that position by moving them out of the original class and into the corresponding element of `intersections`. Any class with a proper subset of its members in `intersections` will have its class number in `classes_to_split`. These classes must be split. Commutative operations are handled differently. This will be explained in Section 2.5.

Since the `buckets` array and the lists that it contains are only used for one iteration of the algorithm, we allocate them in `temp_arena`, and we use `Arena_Mark` and `Arena_Release` to free the memory they occupy.

⟨Touch all uses of elements in `members` 29a⟩ ≡
```
⟨Reset the data structures for commutative operations 37b⟩
Arena_Mark(temp_arena);
{
    Unsigned_Int max_pos = 0;
    Use_Bucket **buckets;
    Unsigned_Int pos;

    ⟨Bucket sort the uses by position 30a⟩
    for (pos = 0; pos <= max_pos; pos++)
    {
        ⟨Touch all uses in position pos 31b⟩
        ⟨Split the members of classes_to_split 33c⟩
    }
}
⟨Split classes representing commutative operations 40b⟩
Arena_Release(temp_arena);
◇
```
Macro referenced in scrap 27a.

When we bucket sort the uses of all members of a class by the position of the use, we will keep a list of uses for each position. Each element contains a pointer to a use of the item (The uses are built during SSA construction) and a pointer to the next element in the list.

⟨Type Declarations 29b⟩ ≡
```
typedef struct use_bucket
{
    UseNode *use;
    struct use_bucket *next;
} Use_Bucket;
◇
```
Macro defined by scraps 11ab, 13a, 22c, 29b, 43a, 57b, 63a.
Macro referenced in scrap 3.

This macro will iterate through all the uses in a particular position. It takes two arguments:

1. A pointer to a `Use_Bucket` structure (the iterator variable), and
2. A list of uses in a particular position.

⟨Macros 29c⟩ ≡
```
#define Position_ForAllUses(bucket, list)    \
    for (bucket = (list);                    \
         bucket;                             \
         bucket = bucket->next)
◇
```
Macro defined by scraps 5b, 6b, 12, 29c, 43d.
Macro referenced in scrap 3.

The use positions are recorded in the use lists during the conversion to SSA form. We look at the use list for each member of the class to determine the maximum position of a use, then we allocate the **buckets** array accordingly. Next, we iterate through all the uses and add them to the corresponding use bucket. Notice the special handling of the redundant-store information. This will be explained in Section 2.6.

```
⟨Bucket sort the uses by position 30a⟩ ≡
    ⟨Find the maximum position of a use of an item in members 30b⟩
    buckets = Arena_GetMemClear(temp_arena, (max_pos + 1)*sizeof(Use_Bucket *));
    {
        Member_Node *node;
        Class_ForAllMembers(node, members)
        {
            Unsigned_Int item = node->item;

            ⟨Bucket sort the uses of item 31a⟩
            ⟨Bucket sort the redundant-store information of node 47a⟩
        }
    }
    ◇
```

Macro referenced in scrap 29a.

```
⟨Find the maximum position of a use of an item in members 30b⟩ ≡
    {
        Member_Node *node;
        Class_ForAllMembers(node, members)
        {
            Unsigned_Int item = node->item;

            ⟨Check the uses of item 30c⟩
            ⟨Check the uses of all the redundant-store information 46c⟩
        }
    }
    ◇
```

Macro referenced in scrap 30a.

```
⟨Check the uses of item 30c⟩ ≡
    {
        UseNode *use;
        Item_ForAllUses(use, item)
        {
            Unsigned_Int pos = UsePos(use);
            if (pos > max_pos) max_pos = pos;
        }
    }
    ◇
```

Macro referenced in scraps 30b, 46c.

For each use of an item, we add a new list element to the `buckets` element corresponding to the use position.

⟨Bucket sort the uses of `item` 31a⟩ ≡

```
{
    UseNode *use;
    Item_ForAllUses(use, item)
    {
        Unsigned_Int pos = UsePos(use);
        Use_Bucket *bucket = Arena_GetMem(temp_arena, sizeof(Use_Bucket));

        bucket->use = use;
        bucket->next = buckets[pos];
        buckets[pos] = bucket;
    }
}
◇
```

Macro referenced in scraps 30a, 47a.

Once we have sorted the uses of all members, we iterate through the use positions and touch all the uses of members of the class in position `pos`. Any classes with a proper subset of their members touched must be split. Those members that were touched must be in a different class from those not touched. The `intersections` array keeps track of the items in each class that must moved to a new class. Notice the special handling of commutative opcodes. This will be explained in Section 2.5.

⟨Touch all uses in position `pos` 31b⟩ ≡

```
SparseSet_Clear(classes_to_split);
{
    Use_Bucket *bucket;
    Position_ForAllUses(bucket, buckets[pos])
    {
        UseNode *use = bucket->use;

        if (UseIsPhiNode(use))
        {
            Unsigned_Int item = UsePhiNode(use)->newName;
            ⟨Touch the defined item 32a⟩
        }
        else
        {
            Operation *oper = UseOper(use);

            if (do_commute && opcode_specs[oper->opcode].details & COMMUTE)
                ⟨Touch the items defined by a commutative operation 37c⟩
            else
                ⟨Touch the items defined by a non-commutative operation 33b⟩
        }
    }
}
◇
```

Macro referenced in scrap 29a.

Update the intersections array by moving item from its class into the intersections array. Redundant-store elimination is an extension to the partitioning algorithm. It will be explained in Section 2.6.

⟨Touch the defined item 32a⟩ ≡

```
{
    Unsigned_Int class_num = lookup[item].class_num;
    Member_Node *node = lookup[item].node;

    if (!⟨This item is handled by redundant-store elimination 44b⟩)
    {
        ⟨Make sure intersections[class_num] is initialized 32b⟩
        ⟨Move node to intersections[class_num] 32c⟩
        ⟨See if intersections[class_num] contains the entire class 33a⟩
    }
}
◇
```

Macro referenced in scraps 31b, 33b.

If this is the first time a member of class_num is touched, we must initialize intersections[class_num]. This is done by creating an empty members list and adding class_num to the classes_to_split set.

⟨Make sure intersections[class_num] is initialized 32b⟩ ≡

```
if (!SparseSet_Member(classes_to_split, class_num))
{
    Member_Node *members;

    ⟨Create a new members list 15b⟩
    intersections[class_num].members = members;
    intersections[class_num].num_members = 0;

    SparseSet_Insert(classes_to_split, class_num);
}
◇
```

Macro referenced in scrap 32a.

Because we are using circular, doubly-linked lists, we can delete a node from its current list and append it to a new list in constant time. We also keep a count of the number of members of intersections[class_num].

⟨Move node to intersections[class_num] 32c⟩ ≡

```
{
    Member_Node *members = intersections[class_num].members;

    ⟨Delete node from its current list 23b⟩
    ⟨Append node to the members list 15c⟩
    intersections[class_num].num_members++;
}
◇
```

Macro referenced in scrap 32a.

If the size of intersections[class_num] reaches the size of the original class, then this class is wholly contained in the *touched* set; therefore, it will not be split.

Notice that we simply copy the pointer to the start of the `intersections` list into the `classes` array. We know that all the members of the class have been moved into the `intersections` list, but there is still the dummy node in the `classes` list. This memory will not be recovered until the arena is destroyed. Perhaps if we were willing to sacrifice some execution time, we could avoid losing this memory.

⟨See if `intersections[class_num]` contains the entire class 33a⟩ ≡

```
if (intersections[class_num].num_members == classes[class_num].num_members)
{
    SparseSet_Delete(classes_to_split, class_num);
    classes[class_num].members = intersections[class_num].members;
}
◇
```

Macro referenced in scrap 32a.

When we touch a non-commutative operation, we move all the defined items to the `intersections` array.

⟨Touch the items defined by a non-commutative operation 33b⟩ ≡

```
{
    Unsigned_Int2 *def_item;
    Operation_ForAllDefs(def_item, oper)
    {
        Unsigned_Int item = *def_item;
        ⟨Touch the defined item 32a⟩
    }
    Operation_ForAllDefTags(def_item, oper)
    {
        Unsigned_Int item = *def_item;
        ⟨Touch the defined item 32a⟩
    }
}
◇
```

Macro referenced in scrap 31b.

Once we have touched all the uses in a position, the `classes_to_split` set contains all the classes with non-empty `intersections` entries. The members of `intersections[class_num]` will become a new class.

⟨Split the members of `classes_to_split` 33c⟩ ≡

```
{
    Unsigned_Int class_num;
    SparseSet_ForAll(class_num, classes_to_split)
    {
        Unsigned_Int num_members = intersections[class_num].num_members;
        Member_Node *members = intersections[class_num].members;

        ⟨Remove members from classes[class_num] 34a⟩
    }
}
◇
```

Macro referenced in scrap 29a.

Since the items were removed from their original class as they were touched, all we need to do is add the new class to the partition and decrease the size of the original class by the number of members removed. We must also add one of the classes to the `worklist`.

⟨Remove `members` from `classes[class_num]` 34a⟩ ≡
```
    ⟨Add the new class to the partition 34b⟩
    classes[class_num].num_members -= num_members;
    ⟨Add the appropriate class to worklist 34c⟩
    num_classes++;
    ◇
```
Macro referenced in scraps 33c, 41ab.

When we add a new class, we must update the `lookup` table entry for all the members.

⟨Add the new class to the partition 34b⟩ ≡
```
    classes[num_classes].members = members;
    classes[num_classes].num_members = num_members;
    {
        Member_Node *node;
        Class_ForAllMembers(node, members)
        {
            Unsigned_Int item = node->item;

            lookup[item].class_num = num_classes;
            ⟨Initialize other Lookup fields 35b, ... ⟩
        }
    }
    ◇
```
Macro referenced in scrap 34a.

If `class_num` is already in the `worklist`, we must add the new class. Otherwise, we add only the smaller of the two classes.

⟨Add the appropriate class to `worklist` 34c⟩ ≡
```
    if (SparseSet_Member(worklist, class_num) ||
        classes[class_num].num_members > num_members)
        SparseSet_Insert(worklist, num_classes);
    else
        SparseSet_Insert(worklist, class_num);
    ◇
```
Macro referenced in scrap 34a.

2.5 Handling Commutative Operations

Commutative operations must be handled with an extension to the partitioning algorithm. The idea is to ignore the position when touching a use of a member of a congruence class. Now, instead of splitting classes based on which members were touched or not touched, we split classes based on which members were touched 0, 1, or 2 times. Consider the example code fragment in Figure 2.8, and assume that X $\not\cong$ Y. The initial partition is shown in Figure 2.8. Let the class containing X be the first removed from the worklist. Touching the uses of X will result in A and B being touched once and C being touched twice. Therefore, A and B will be placed in a class together, and C will be in a class by itself. Let the class containing Y be the next one removed from the worklist. Touching the uses of Y will result in A and B being touched once and D being touched twice. Therefore, no further splitting of classes is required. The final partition is shown in Figure 2.8.

$$\begin{aligned} A &\leftarrow X + Y \\ B &\leftarrow Y + X \\ C &\leftarrow X + X \\ D &\leftarrow Y + Y \end{aligned}$$

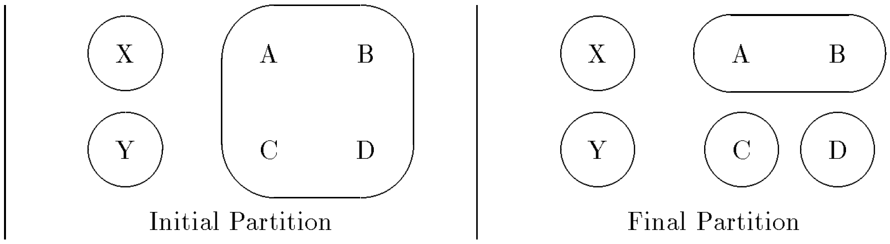

Figure 2.8: Commutativity Example

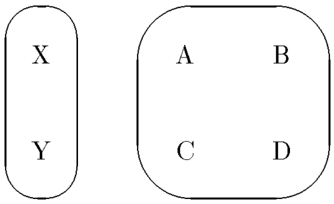

Figure 2.9: Partition for Second Commutativity Example

Now assume that X ≅ Y. The initial partition is shown in Figure 2.9. Let the class containing X and Y be the first removed from the worklist. Touching the uses of X and Y will result in A, B, C, and D all being touched twice. Therefore, no further splitting of classes is required, and the initial partition is also the final partition.

Recall that we left space for some extra fields in the `Lookup` structure when it was originally declared. For commutative operations, we need a field to count the number of times a defined item has been touched.

⟨Other `Lookup` fields 35a⟩ ≡

```
Unsigned_Int2 num_touches;
◇
```

Macro defined by scraps 35a, 43e.
Macro referenced in scrap 13a.

Initially, the `num_touches` field will be zero.

⟨Initialize other `Lookup` fields 35b⟩ ≡

```
lookup[item].num_touches = 0;
◇
```

Macro defined by scraps 35b, 44a.
Macro referenced in scraps 15a, 17a, 34b.

Recall that when we touched an item for non-commutative operations, we moved the item out of its current class and into the `intersections` array. The `touched_once` and `touched_twice` arrays are used like the `intersections` array. The first time an item is touched, it is moved from its original class into `touched_once`. The second time it is touched, it is moved from `touched_once` to `touched_twice`. We use two sets called `touched_once_classes` and `touched_twice_classes` to keep track of which classes have a non-empty entry in `touched_once` or `touched_twice`, respectively. These are analogous to the `classes_to_split` set used for non-commutative operations.

⟨Global Variables 36⟩ ≡

```
static Class *touched_once;
static Class *touched_twice;
static SparseSet touched_once_classes;
static SparseSet touched_twice_classes;
```
◇

Macro defined by scraps 5ac, 6d, 7b, 11c, 13b, 28a, 36, 57c, 58a.
Macro referenced in scrap 3.

Figure 2.10 depicts the partition if the variable A in Figure 2.8 is in class number 5. The first time A is touched, it will be removed from the `members` list of `classes[5]` and appended to the `members` list of `touched_once[5]`. The second time A is touched, it will be removed from the `touched_once[5]` and appended to `touched_twice[5]`. Each time the item is touched, the `num_touches` field of the `lookup` array will be incremented.

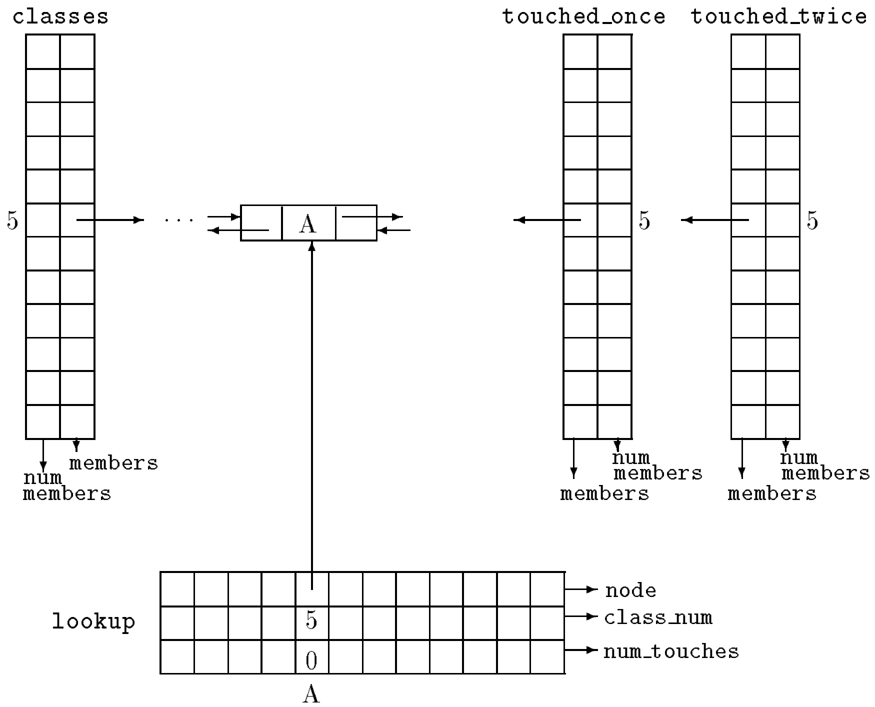

Figure 2.10: Data Structures for Handling Commutative Operations

Since these arrays and sets are indexed by class numbers, they are allocated with `defCount` entries. They are only allocated if the user has requested that we handle commutative operations.

⟨Initialize the data structures for commutative operations 37a⟩ ≡
```
if (do_commute)
{
    touched_once_classes = SparseSet_Create(partition_arena, defCount);
    touched_twice_classes = SparseSet_Create(partition_arena, defCount);
    touched_once = Arena_GetMem(partition_arena, defCount*sizeof(Class));
    touched_twice = Arena_GetMem(partition_arena, defCount*sizeof(Class));
}
◇
```
Macro referenced in scrap 28b.

Recall that the `classes_to_split` set is reset before touching the elements of a particular position. We must reset the `touched_once_classes` and `touched_twice_classes` sets before touching the items in any position.

⟨Reset the data structures for commutative operations 37b⟩ ≡
```
if (do_commute)
{
    SparseSet_Clear(touched_once_classes);
    SparseSet_Clear(touched_twice_classes);
}
◇
```
Macro referenced in scrap 29a.

When we touch a use in a commutative operation, we must move all the defined registers and tags [4] out of their current class and into the `touched_once` array and then into the `touched_twice` array.

⟨Touch the items defined by a commutative operation 37c⟩ ≡
```
{
    Unsigned_Int2 *item_ptr;
    Operation_ForAllDefs(item_ptr, oper)
    {
        Unsigned_Int item = *item_ptr;
        ⟨Move item out of its class and into touched_xxx 38a⟩
    }
    Operation_ForAllDefTags(item_ptr, oper)
    {
        Unsigned_Int item = *item_ptr;
        ⟨Move item out of its class and into touched_xxx 38a⟩
    }
}
◇
```
Macro referenced in scrap 31b.

[4] Currently, there are no commutative operations that define tags, but it is better to be safe than sorry

Update the number of times `item` has been touched during this pass of partitioning. Redundant-store elimination is another extension to the partitioning algorithm. It will be explained in Section 2.6. Notice that we don't check that all the items of the class are in the `touched_once` set. This is because they could be touched again and moved to `touched_twice`. We'll check the size just before we actually try to split the class.

⟨Move `item` out of its class and into `touched_xxx` 38a⟩ ≡

```
{
    Unsigned_Int class_num = lookup[item].class_num;
    Unsigned_Int num_touches = lookup[item].num_touches;
    Member_Node *node = lookup[item].node;

    if (!⟨This item is handled by redundant-store elimination 44b⟩)
    {
        if (!num_touches)
        {
            ⟨Make sure touched_once[class_num] is initialized 38b⟩
            ⟨Move node to touched_once[class_num] 39a⟩
        }
        else if (num_touches == 1)
        {
            ⟨Make sure touched_twice[class_num] is initialized 39b⟩
            ⟨Move node to touched_twice[class_num] 39c⟩
            ⟨See if touched_twice[class_num] contains the entire class 40a⟩
        }
        else
        {
            fprintf(stderr,
                    "Internal error: Can't touch an item more than twice: %d\n",
                    item);
            ABORT;
        }
    }
}
◇
```

Macro referenced in scrap 37c.

If this is the first time a member of `class_num` is touched, we must initialize `touched_once[class_num]`. This is done by creating an empty members list and adding `class_num` to the `touched_once_classes` set.

⟨Make sure `touched_once[class_num]` is initialized 38b⟩ ≡

```
if (!SparseSet_Member(touched_once_classes, class_num))
{
    Member_Node *members;

    ⟨Create a new members list 15b⟩
    touched_once[class_num].members = members;
    touched_once[class_num].num_members = 0;

    SparseSet_Insert(touched_once_classes, class_num);
}
◇
```

Macro referenced in scrap 38a.

Because we are using circular, doubly-linked lists, we can delete a node from its current list and append it to a new list in constant time. We also keep a count of the number of members of `touched_once[class_num]` and the number of times `item` has been touched.

⟨Move `node` to `touched_once[class_num]` 39a⟩ ≡

```
{
    Member_Node *members = touched_once[class_num].members;

    ⟨Delete node from its current list 23b⟩
    ⟨Append node to the members list 15c⟩
    touched_once[class_num].num_members++;
    lookup[item].num_touches++;
}
◇
```

Macro referenced in scrap 38a.

If this is the first time a member of `class_num` is moved into the `touched_twice` array, we must initialize `touched_twice[class_num]`. This is done by creating an empty members list and adding `class_num` to the `touched_twice_classes` set.

⟨Make sure `touched_twice[class_num]` is initialized 39b⟩ ≡

```
if (!SparseSet_Member(touched_twice_classes, class_num))
{
    Member_Node *members;

    ⟨Create a new members list 15b⟩
    touched_twice[class_num].members = members;
    touched_twice[class_num].num_members = 0;

    SparseSet_Insert(touched_twice_classes, class_num);
}
◇
```

Macro referenced in scrap 38a.

We move a node into the `touched_twice` array the same way we moved it into the `touched_once` array.

⟨Move `node` to `touched_twice[class_num]` 39c⟩ ≡

```
{
    Member_Node *members = touched_twice[class_num].members;

    ⟨Delete node from its current list 23b⟩
    touched_once[class_num].num_members--;
    ⟨Append node to the members list 15c⟩
    touched_twice[class_num].num_members++;
    lookup[item].num_touches++;
}
◇
```

Macro referenced in scrap 38a.

If the size of touched_twice[class_num] reaches the size of the original class, then this class is wholly contained, and will not be split. Notice that we simply copy the pointer to the start of the touched_twice list into the classes array. We know that all the members of the class have been moved into the touched_twice list, but there is still the dummy node in the classes list. This memory will not be recovered until the arena is destroyed. Perhaps if we were willing to sacrifice some execution time, we could avoid losing this memory.

⟨See if touched_twice[class_num] contains the entire class 40a⟩ ≡

```
if (touched_twice[class_num].num_members == classes[class_num].num_members)
{
    Member_Node *node;
    Member_Node *members = touched_twice[class_num].members;

    Class_ForAllMembers(node, members)
        lookup[node->item].num_touches = 0;

    SparseSet_Delete(touched_twice_classes, class_num);
    classes[class_num].members = members;
}
◇
```

Macro referenced in scrap 38a.

Once all use positions have been processed, we are ready to split the classes that represent commutative operations. The sets touched_once_classes and hit_twice_classes are precisely those classes that must be split.

⟨Split classes representing commutative operations 40b⟩ ≡

```
if (do_commute)
{
    ⟨Process the elements in touched_once_classes 41a⟩
    ⟨Process the elements in touched_twice_classes 41b⟩
}
◇
```

Macro referenced in scrap 29a.

Here is where we check for the **touched_once** set to be all of the class. Recall that we didn't check this earlier because some of the members could have later been moved to the **touched_twice** set. If the **touched_once** set is not the entire class, then its members are processed the same way the members of **classes_to_split** were.

⟨Process the elements in touched_once_classes 41a⟩ ≡

```
{
    Unsigned_Int class_num;
    SparseSet_ForAll(class_num, touched_once_classes)
    {
        if (touched_once[class_num].num_members == classes[class_num].num_members)
        {
            Member_Node *node;
            Member_Node *members = touched_once[class_num].members;

            Class_ForAllMembers(node, members)
                lookup[node->item].num_touches = 0;

            classes[class_num].members = members;
        }
        else
        {
            Unsigned_Int num_members = touched_once[class_num].num_members;
            Member_Node *members = touched_once[class_num].members;

            ⟨Remove members from classes[class_num] 34a⟩
        }
    }
}
◇
```

Macro referenced in scrap 40b.

The elements of **touched_twice_classes** are processed just like the members of **classes_to_split** were.

⟨Process the elements in touched_twice_classes 41b⟩ ≡

```
{
    Unsigned_Int class_num;
    SparseSet_ForAll(class_num, touched_twice_classes)
    {
        Unsigned_Int num_members = touched_twice[class_num].num_members;
        Member_Node *members = touched_twice[class_num].members;

        ⟨Remove members from classes[class_num] 34a⟩
    }
}
◇
```

Macro referenced in scrap 40b.

2.6 Eliminating Redundant Stores

Redundant-store operations write the same value to a memory location that was previously written there. Therefore, they do not alter the contents of memory, and they can be eliminated from the routine. Redundant stores should not be confused with dead stores which write a value to memory that is never subsequently read. Handling redundant scalar stores requires an extension to the partitioning algorithm. Consider the program fragment in Figure 2.11. The **FRAME** operation defines the initial values for the memory tags x and y. During the conversion to SSA form, all tags were given subscripts to give each one a unique definition point. The store operations have two tags associated with them[5]. The first tag is the *before* value of the memory location, and the second is the *after* value of the memory location. In our example, the store from r_0 is redundant while the store from r_1 is not. We would like to check for congruence between the before and after tag values to determine if a store is redundant. Unfortunately, the tag x_0 is defined by the FRAME operation, and the tag x_1 is defined by a store operation. Therefore, they will be in different classes in the initial partition. The initial partition is shown in Figure 2.12. Clearly, x_0 can never be congruent to x_1 using the unmodified partitioning algorithm.

`FRAME` $[x_0\ y_0]$
$\vdots$
$r_0 \leftarrow$ `LOAD` x_0
r1 $\leftarrow$ 1
$\vdots$
`STORE` r_0 $[x_0]\ [x_1]$
`STORE` r_1 $[y_0]\ [y_1]$

Figure 2.11: Example Program for Redundant-Store Elimination

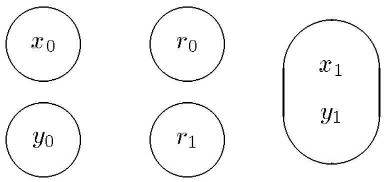

Figure 2.12: Initial Partition for Redundant-Store Elimination Example

We must treat scalar load and store operations as a copy from a register to memory or vice versa. Since copying a value does not change it, we must ensure that the source and destination of a copy will remain in the same congruence class throughout the partitioning process. To accomplish this, we keep a list of copies for each item in the partition. During the process of refining the partition, the copy list for an item moves from class to class with the original item. In our example routine, r_0 is a copy of x_0, and x_1 is a copy of r_0. Also, y_0 is a copy of r_1. Given this scheme, the initial partition is also the final partition (See Figure 2.13). Notice that $x_0 \cong x_1$, but $y_0 \not\cong y_1$. Thus, we can eliminate the first store but not the second.

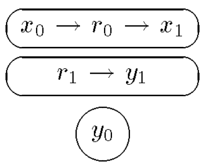

Figure 2.13: Partition to Enable Redundant-Store Elimination

[5]Actually, these are lists of tags. The first tag is the one that appears in the source, and the others are possible aliases. In this example, the lists have length one.

The copy list for an item will be kept in the item's Member_Node structure. Each element contains the SSA name of the copy and a pointer to the next element in the list.

⟨Type Declarations 43a⟩ ≡
```
typedef struct copy_node
{
    struct copy_node *next;
    Unsigned_Int2 item;
} Copy_Node;
```
◇

Macro defined by scraps 11ab, 13a, 22c, 29b, 43a, 57b, 63a.
Macro referenced in scrap 3.

Recall that we left space for some extra fields in the Member_Node structure when it was originally declared. For redundant-store elimination, we keep the copy list for an item there.

⟨Other Member_Node fields 43b⟩ ≡
```
struct copy_node *copies;
```
◇

Macro referenced in scrap 11a.

Initially, the copy list for an item is empty.

⟨Initialize other Member_Node fields 43c⟩ ≡
```
node->copies = NULL;
```
◇

Macro referenced in scraps 15b, 16a, 17a, 20a.

We'll need a macro to iterate through the copies of a node. It takes two arguments:

1. A pointer to a Copy_Node structure (the iterator variable), and
2. A pointer to a Member_Node structure.

⟨Macros 43d⟩ ≡
```
#define Node_ForAllCopies(copy, node)     \
    for (copy = (node)->copies;           \
         copy;                            \
         copy = copy->next)
```
◇

Macro defined by scraps 5b, 6b, 12, 29c, 43d.
Macro referenced in scrap 3.

While refining the partition, we do not touch any items that are in the copy list of some other item. Therefore, we will add a field in the Lookup structure to indicate if the item is a copy of some other item.

⟨Other Lookup fields 43e⟩ ≡
```
Boolean is_copy;
```
◇

Macro defined by scraps 35a, 43e.
Macro referenced in scrap 13a.

Figure 2.14 depicts the partition if x_0 from Figure 2.11 is class number five. The items r_0 and x_1 are in the copy list for x_0. The `node` field of the `lookup` entries for r_0 and x_1 point to the `Member_Node` structure for x_0.

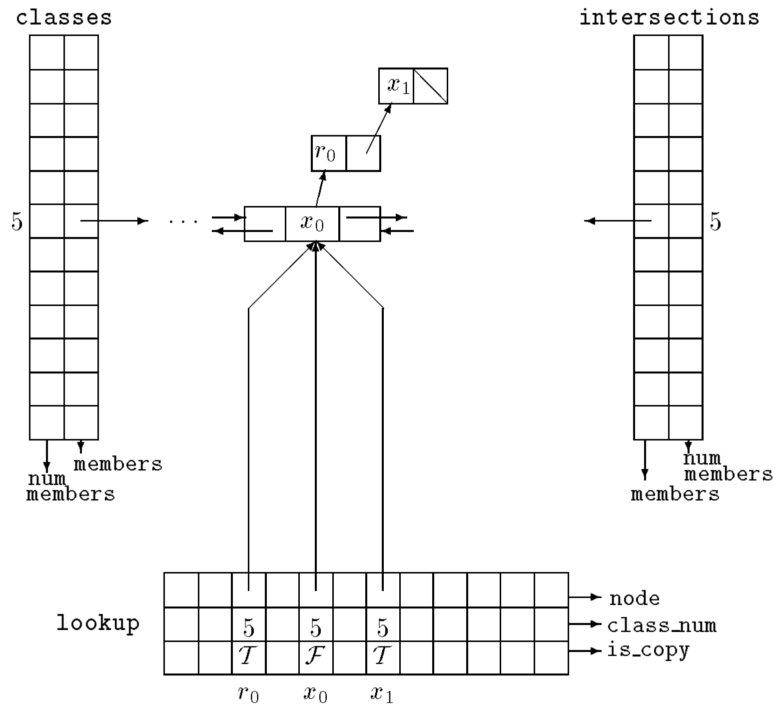

Figure 2.14: Data Structures for Redundant-Store Elimination

All items defined by operations other than scalar loads and stores are not copies.

⟨Initialize other `Lookup` fields 44a⟩ ≡

```
lookup[item].is_copy = FALSE;
```

◇

Macro defined by scraps 35b, 44a.
Macro referenced in scraps 15a, 17a, 34b.

An item defined by a scalar load or store operation will have its `is_copy` field set to `TRUE`.

⟨This `item` is handled by redundant-store elimination 44b⟩ ≡

```
lookup[item].is_copy
```

◇

Macro referenced in scraps 32a, 38a.

When we initialize the partition, scalar loads and stores require special handling. For a load operation, the defined register is a copy of the first referenced tag. Any other tags are possible aliases.

⟨Add a scalar load operation to the initial partition 45a⟩ ≡

```
{
    Unsigned_Int reg = oper->arguments[oper->defined - 1];
    Unsigned_Int tag = oper->arguments[oper->defined];

    ⟨Add reg to the copy list for tag 45b⟩
}
◇
```

Macro referenced in scrap 19a.

To add an item to the copy list, we allocate a `Copy_Node` structure to contain the item and insert it at the beginning of the copy list for the node. Notice that the `lookup` table entry points to the node for the original item, not the copy.

⟨Add `reg` to the copy list for `tag` 45b⟩ ≡

```
{
    Member_Node *node = lookup[tag].node;
    Copy_Node *copy_node = Arena_GetMem(partition_arena, sizeof(Copy_Node));

    copy_node->item = reg;
    copy_node->next = node->copies;
    node->copies = copy_node;
    lookup[reg].num_touches = 0;
    lookup[reg].is_copy = TRUE;
    lookup[reg].node = node;
}
◇
```

Macro referenced in scrap 45a.

For a store operation, the first defined tag is a copy of the last referenced register. Any other tags are possible aliases, and they are put into separate classes.

⟨Add a scalar store operation the the initial partition 45c⟩ ≡

```
{
    Unsigned_Int tag = *Operation_Second_List_Start(oper);
    Unsigned_Int reg = oper->arguments[oper->defined - 1];

    ⟨Add tag to the copy list for reg 46a⟩
    ⟨Put the aliases in separate classes 46b⟩
}
◇
```

Macro referenced in scrap 19a.

To add an item to the copy list, we allocate a `Copy_Node` structure to contain the item and insert it at the beginning of the copy list for the node. Notice that the `lookup` table entry points to the node for the original item, not the copy.

⟨Add `tag` to the copy list for `reg` 46a⟩ ≡

```
{
    Member_Node *node = lookup[reg].node;
    Copy_Node *copy_node = Arena_GetMem(partition_arena, sizeof(Copy_Node));

    copy_node->item = tag;
    copy_node->next = node->copies;
    node->copies = copy_node;
    lookup[tag].num_touches = 0;
    lookup[tag].is_copy = TRUE;
    lookup[tag].node = node;
}
◇
```

Macro referenced in scrap 45c.

The first tag in the defined tag list is the tag that is definitely being stored by this operation. All others are possible aliases of the first tag.

⟨Put the aliases in separate classes 46b⟩ ≡

```
{
    Boolean first = TRUE;
    Unsigned_Int2 *tag_ptr;
    Operation_ForAllDefTags(tag_ptr, oper)
        if (first)
            first = FALSE;
        else
        {
            Member_Node *node = Arena_GetMem(partition_arena, sizeof(Member_Node));

            node->item = *tag_ptr;
            node->copies = NULL;
            ⟨Create a new class for node 15a⟩
        }
}
◇
```

Macro referenced in scrap 45c.

When refining the partition, we bucket sort the uses of all members of a congruence class. The first step is to determine the maximum position of a use of any member. Since copies are also members of the class, we must also consider their uses.

⟨Check the uses of all the redundant-store information 46c⟩ ≡

```
{
    Copy_Node *copy_node;
    Node_ForAllCopies(copy_node, node)
    {
        Unsigned_Int item = copy_node->item;
        ⟨Check the uses of item 30c⟩
    }
}
◇
```

Macro referenced in scrap 30b.

During the actual bucket sorting, we must add the uses of copies to the appropriate bucket.

⟨Bucket sort the redundant-store information of `node` 47a⟩ ≡

```
{
    Copy_Node *copy_node;
    Node_ForAllCopies(copy_node, node)
    {
        Unsigned_Int item = copy_node->item;
        ⟨Bucket sort the uses of item 31a⟩
    }
}
◇
```

Macro referenced in scrap 30a.

Once the partition has stabilized, we must find all the items that have copies, and write down the correct `class_num` in the `lookup` table for all the copies. During the partitioning phase, we just let the copies go along for the ride without updating the `class_num` field of their `lookup` table entry.

⟨Prepare the partition for redundant-store elimination 47b⟩ ≡

```
{
    Unsigned_Int class_num;
    for (class_num = 1; class_num < num_classes; class_num++)
    {
        Member_Node *node;
        Copy_Node *copy_node;

        Class_ForAllMembers(node, classes[class_num].members)
            Node_ForAllCopies(copy_node, node)
            {
                lookup[copy_node->item].class_num = class_num;
                classes[class_num].num_members++;
            }
    }
}
◇
```

Macro referenced in scrap 27a.

During the renumbering process, we can determine that a store is redundant and eliminate it. Recall that each store is given a before and after tag value. These are stored in the first position of the referenced and defined tag lists respectively. If the before and after tag values are congruent, this store is redundant. We eliminate it and `continue` with the next operation in the instruction.

⟨If `oper` is a redundant store, eliminate it and continue 47c⟩ ≡

```
if (elim_stores &&
    opcode_specs[oper->opcode].details & STORE &&
    opcode_specs[oper->opcode].details & SCALAR)
{
    Unsigned_Int ref_tag = oper->arguments[oper->defined];
    Unsigned_Int def_tag = *Operation_Second_List_Start(oper);

    if (lookup[ref_tag].class_num == lookup[def_tag].class_num)
    {
        oper->critical = FALSE;
        num_removed++;
        continue;
    }
}
◇
```

Macro referenced in scrap 52b.

Chapter 3

Renumbering

Once the partition has stabilized, we are ready to move into the next phase of global value numbering. We renumber the ϕ-nodes and registers in the routine based on the congruence classes in the final partition. Since we have the class number of every SSA name stored in the `lookup` array, we can do this in a single pass over the routine. However, if the user wants to prepare the output for partial redundancy elimination, the register numbers must obey certain conventions. These will be discussed in the next section. Since the class numbers are more or less arbitrary, we will create a mapping from class numbers to register numbers that obey **partial**'s naming rules. We will create this mapping by overwriting the `class_num` field of each entry in the `lookup` array. This will allow us to perform renumbering as a separate step that is independent of this option.

⟨Renumber the ϕ-nodes and registers based on the congruence classes 48⟩ ≡

```
if (debug >= MAJOR_PHASES)
    fprintf(stderr, "Renumber the phi-nodes and registers\n");

if (prepare_for_partial)
    ⟨Create a mapping of register numbers to obey partial's register naming rules 49a⟩
⟨Renumber the φ-nodes and operations in the routine 51b⟩
◇
```

Macro referenced in scrap 4.

3.1 Preparing for Partial Redundancy Elimination

Our implementation of partial redundancy elimination requires the registers to be numbered so that the register number of an expression register will be higher than the register numbers of its operands [6, 5]. To accomplish this, we walk the CFG in reverse postorder to guarantee that the operands of an expression are visited before the expression. During this walk, we rename congruence classes representing registers in the order of appearance of some member. We'll use an array called **reg_map** to hold the mapping of class numbers to register names in the new numbering scheme. The **in_reg_map** set keeps track of which elements of **reg_map** have valid entries. Notice that we don't renumber the tags in the routine because their original names will be restored when we convert out of SSA form.

⟨Create a mapping of register numbers to obey **partial**'s register naming rules 49a⟩ ≡

```
{
    Arena partial_arena = Arena_Create();
    Unsigned_Int2 *reg_map = Arena_GetMem(partial_arena, num_classes*sizeof(Unsigned_Int2));
    VectorSet in_reg_map = VectorSet_Create(partial_arena, num_classes);
    Unsigned_Int num_registers;
    Block *block;

    if (debug >= MINOR_PHASES)
        fprintf(stderr, "    Create a mapping of register names for partial\n");

    ⟨Add class zero to reg_map 49b⟩
    ForAllBlocks_rPostorder(block)
    {
        ⟨Create a new register number for any φ-nodes that define registers 50a⟩
        ⟨Create a new register number for any operations that define registers 50b⟩
    }
    ⟨Overwrite the class_num field of the lookup array 51a⟩
    Arena_Destroy(partial_arena);
}
◇
```

Macro referenced in scrap 48.

When we created the partition, we allocated class zero to contain the SSA name zero. This name signifies an undefined value. We preserve this convention by forcing zero to always map to itself.

⟨Add class zero to **reg_map** 49b⟩ ≡

```
VectorSet_Insert(in_reg_map, 0);
reg_map[0] = 0;
num_registers = 1;
◇
```

Macro referenced in scrap 49a.

When renumbering the ϕ-nodes, we only consider the ones that define registers. Any ϕ-node whose congruence class has not been assigned a new register number will be given one.

⟨Create a new register number for any ϕ-nodes that define registers 50a⟩ ≡

```
{
    PhiNode *phi_node;
    Block_ForAllPhiNodes(phi_node, block)
        if (PhiNode_IsRegister(phi_node))
        {
            Unsigned_Int class_num = lookup[phi_node->newName].class_num;

            if (!VectorSet_Member(in_reg_map, class_num))
            {
                reg_map[class_num] = num_registers++;
                VectorSet_Insert(in_reg_map, class_num);
            }
        }
}
◇
```

Macro referenced in scrap 49a.

We iterate through all the operations in the block and assign a new register number to any defined register whose congruence class does not have an entry in `reg_map`.

⟨Create a new register number for any operations that define registers 50b⟩ ≡

```
{
    Inst *inst;
    Block_ForAllInsts(inst, block)
    {
        Operation **oper_ptr;
        Inst_ForAllOperations(oper_ptr, inst)
        {
            Operation *oper = *oper_ptr;
            Unsigned_Int2 *reg;
            Operation_ForAllDefs(reg, oper)
            {
                Unsigned_Int class_num = lookup[*reg].class_num;

                if (!VectorSet_Member(in_reg_map, class_num))
                {
                    reg_map[class_num] = num_registers++;
                    VectorSet_Insert(in_reg_map, class_num);
                }
            }
        }
    }
}
◇
```

Macro referenced in scrap 49a.

Once we have built `reg_map`, we must store these values in the `lookup` array. Overwriting the array allows us to renumber the routine without knowing that this mapping is being used.

⟨Overwrite the `class_num` field of the `lookup` array 51a⟩ ≡

```
{
    Unsigned_Int i;
    for (i = 0; i < defCount; i++)
        if (IsRegister(i))
            lookup[i].class_num = reg_map[lookup[i].class_num];
}
```

◇

Macro referenced in scrap 49a.

3.2 Renumbering the ϕ-nodes and Registers

Once the partitioning is completed and the register mapping (if requested) is set up, we are ready to renumber the ϕ-nodes and registers in the routine according to their congruence class. We accomplish this by overwriting every register name in the routine by the `class_num` value in the `lookup` array. Notice that we don't renumber the tags in the routine; their original names will be restored during the conversion out of SSA form.

⟨Renumber the ϕ-nodes and operations in the routine 51b⟩ ≡

```
{
    Unsigned_Int num_removed = 0;
    Block *block;

    if (debug >= MINOR_PHASES)
        fprintf(stderr, "    Renumber the phi-nodes and operations\n");

    ForAllBlocks(block)
    {
        ⟨Renumber the φ-nodes that define registers 52a⟩
        ⟨Renumber the operations that define registers 52b⟩
    }
    if (debug && elim_stores)
        fprintf(stderr, "       %d redundant store operations removed.\n",
                num_removed);
}
```

◇

Macro referenced in scrap 48.

We iterate through the ϕ-nodes and renumber the ones that define registers according to their congruence class. In Section 1.5, we declared the variable **max_register** because that value will be needed for the conversion out of SSA form. The correct value of this variable will be determined during the renumbering process. If any of the ϕ-nodes or operations define a register number larger than **max_register**, we update its value. Notice that we don't renumber the ϕ-nodes that define tags.

⟨Renumber the ϕ-nodes that define registers 52a⟩ ≡

```
{
    PhiNode *phi_node;
    Block_ForAllPhiNodes(phi_node, block)
        if (PhiNode_IsRegister(phi_node))
        {
            PhiNodeArg *arg;
            Unsigned_Int new_name = lookup[phi_node->newName].class_num;

            PhiNode_ForAllArgs(arg, phi_node)
                arg->item = lookup[arg->item].class_num;

            phi_node->newName = new_name;
            if (new_name > max_register) max_register = new_name;
        }
}
◇
```

Macro referenced in scrap 51b.

We iterate through the operations in the block. Before we renumber the registers, we try to eliminate the operation. The operation can be eliminated if it is a redundant scalar store and the user has requested this optimization (See Section 2.6).

⟨Renumber the operations that define registers 52b⟩ ≡

```
{
    Inst *inst;
    Block_ForAllInsts(inst, block)
    {
        Operation **oper_ptr;
        Inst_ForAllOperations(oper_ptr, inst)
        {
            Operation *oper = *oper_ptr;

            ⟨If oper is a redundant store, eliminate it and continue 47c⟩
            ⟨Renumber the registers according to their congruence class 53⟩
        }
    }
}
◇
```

Macro referenced in scrap 51b.

We renumber the used and defined registers of the operation by overwriting them. If any of the defined registers are given a number larger than `max_register`, we update its value. Notice that we don't renumber the tags.

⟨Renumber the registers according to their congruence class 53⟩ ≡

```
{
    Unsigned_Int2 *item;
    Operation_ForAllUses(item, oper)
        *item = lookup[*item].class_num;

    Operation_ForAllDefs(item, oper)
    {
        Unsigned_Int new_name = lookup[*item].class_num;

        *item = new_name;
        if (new_name > max_register) max_register = new_name;
    }
}
◇
```

Macro referenced in scrap 52b.

Chapter 4

Removing Operations

Chapters 2 and 3 explain how to partition the values in the routine into congruence classes and how to renumber the registers and ϕ-nodes so that congruent values are given the same number. However, these two steps alone will not improve the running time of the routine; we must also remove the redundant computations. There are three possibilities for achieving this goal:

Dominator-Based Removal The technique suggested by Alpern, Wegman, and Zadeck is to remove computations that are dominated by another member of the congruence class [3]. Figure 4.1 shows an example routine that we can improve with this method. Since the computation of z in block B_1 dominates the computation in block B_4, the second computation can be removed.

AVAIL-Based Removal The classical approach is to remove computations that are in the set of available expressions (AVAIL) at the point where they appear in the routine [2]. This approach uses data-flow analysis to determine the set of expressions available along all paths from the start of the routine. Notice that the calculation of z in Figure 4.1 will be removed because it is in the AVAIL set. In fact, any computation that would be removed by dominator-based removal would also be removed by AVAIL-based removal. However, there are improvements that can be made by the AVAIL-based technique that are not possible using dominators. Consider the routine in Figure 4.2. Since z is calculated in both B_2 and B_3, it is in the AVAIL set at B_4. Thus, the calculation of z in B_4 can be removed. However, since neither B_2 or B_3 dominate B_4, dominator-based removal could not improve this routine.

Partial Redundancy Elimination PRE is an optimization introduced by Morel and Renvoise [8]. Partially redundant computations are redundant along some, but not necessarily all, execution paths. Notice that the computation of z in Figure 4.2 is redundant along all paths to block B_4, so it will be removed by PRE. On the other hand, the routine in Figure 4.3 cannot be improved using AVAIL-based removal because z is not available along the path through block B_2. The calculation of z is computed twice along the path through B_3 but only once along the path through B_2. Therefore, it is considered partially redundant. PRE can move the computation of z from block B_4 to block B_2. This will shorten the path through B_3 and leave the length of the path through B_2 unchanged.

4.1 Dominator-Based Removal

To perform dominator-based removal, we consider each congruence class and look for pairs of members where one dominates the other. To make the algorithm efficient, we bucket sort the members of the class based on the preorder index in the dominator tree of the block where they are computed. A naive bucket sorting algorithm would keep a list of items defined for each block. Assume that items x and y are congruent and

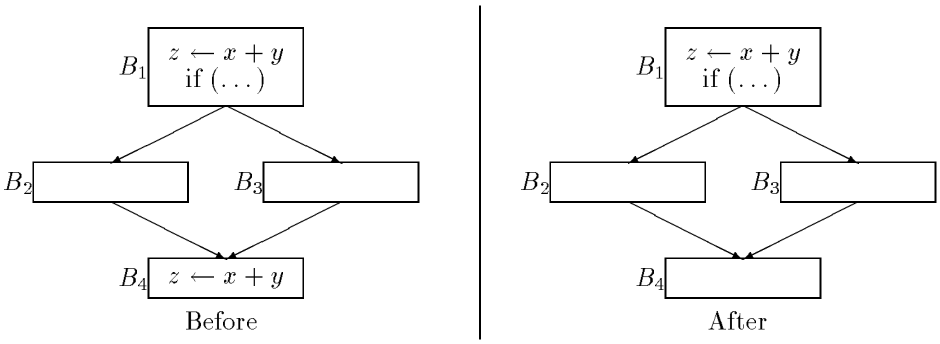

Figure 4.1: Program Improved by Dominator-Based Removal

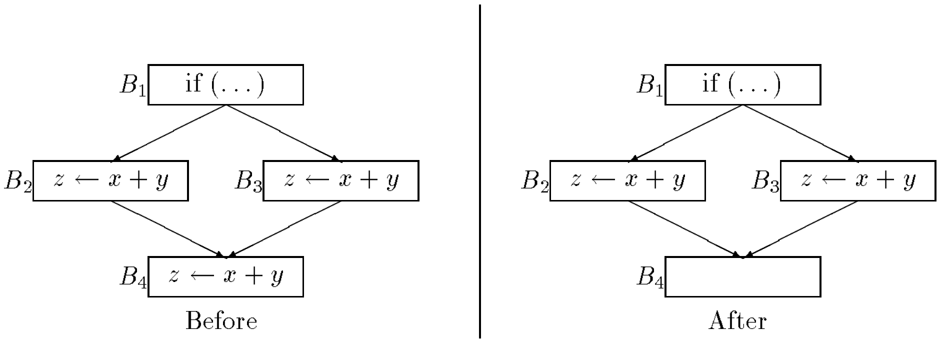

Figure 4.2: Program Improved by AVAIL-Based Removal

both are computed in block B. The bucket sorting algorithm would place them in a list indexed by the preorder index of B. See figure 4.4.

However, we can improve upon the naive algorithm. If more than one member is computed in the same block, only the one computed earliest in the block must be considered, because it dominates all the others. Thus, instead of an array of lists, we can keep an array of SSA names. When an item is inserted into an entry that already contains an item, we can select the one that is computed earlier in the block. This item will be written into the array entry and the operation computing the other will be removed immediately.

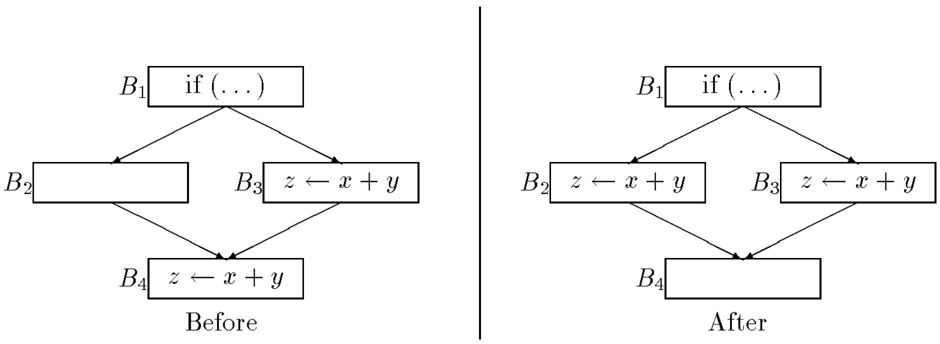

Figure 4.3: Program Improved by Partial Redundancy Elimination

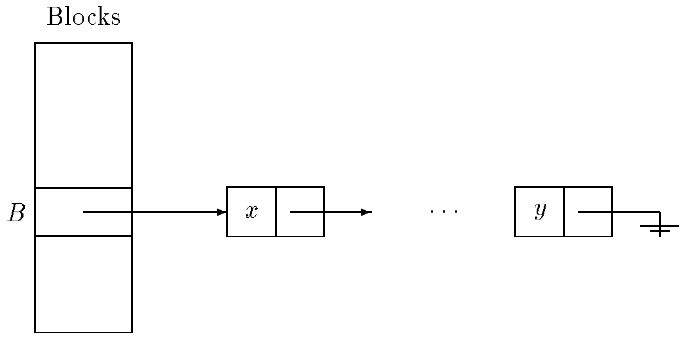

Figure 4.4: Naive Bucket Sorting Algorithm

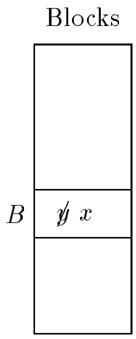

Figure 4.5: Better Bucket Sorting Algorithm

Assume that items x and y are congruent and both are computed in block B, and that x is computed earlier than y. The bucket sorting algorithm would replace y with x in the entry indexed by the preorder index of B. See Figure 4.5.

Once we have found the item computed earliest in each block, we can compare adjacent elements in the list and decide if one dominates the other. This decision is based on an ancestor test in the dominator tree. The entire process can be done in time proportional to the size of the class.

We only consider classes with more than one member, because a class with a single member can not cause any operations to be removed.

⟨Optionally perform dominator-based removal 57a⟩ ≡
```
if (do_dominators)
{
    Arena dom_arena = Arena_Create();
    Unsigned_Int num_removed = 0;
    Unsigned_Int i;

    if (debug >= MAJOR_PHASES)
        fprintf(stderr, "Perform dominator-based removal\n");

    ⟨Initialize the dominator data structures 58b, ... ⟩
    for (i = 1; i < num_classes; i++)
        if (classes[i].num_members > 1)
        {
            ⟨Bucket sort the members of classes[i] 59b⟩
            ⟨Compare items defined in different blocks 61a⟩
        }

    Arena_Destroy(dom_arena);

    if (debug)
        fprintf(stderr, "        %d phi-nodes or operations removed based on dominators.\n",
                num_removed);
}
◇
```
Macro referenced in scrap 4.

During dominator-based removal, we'll need to perform an ancestor test on the dominator tree. To do this in constant time, we must know the preorder index and the number of descendants of each block in the dominator tree. Since this information isn't recorded for us while the dominator tree is being built, we walk the dominator tree and record this information in an array called **dominator_info**. It is indexed by the **preorder_index** of each block in the DFS tree. Each entry contains the preorder index of the block in the dominator tree and the size of the subtree of the dominator tree rooted at this block.

⟨Type Declarations 57b⟩ ≡
```
typedef struct
{
    Unsigned_Int2 dominator_index;
    Unsigned_Int2 dominator_size;
} Dom_Info;
◇
```
Macro defined by scraps 11ab, 13a, 22c, 29b, 43a, 57b, 63a.
Macro referenced in scrap 3.

⟨Global Variables 57c⟩ ≡
```
static Dom_Info *dominator_info;
◇
```
Macro defined by scraps 5ac, 6d, 7b, 11c, 13b, 28a, 36, 57c, 58a.
Macro referenced in scrap 3.

We'll also need a preorder list of blocks in the dominator tree and an array of SSA names for bucket sorting the members of a class by block.

⟨Global Variables 58a⟩ ≡

```
static Unsigned_Int2 *dom_preorder_list;
static Unsigned_Int2 *buckets;
◇
```

Macro defined by scraps 5ac, 6d, 7b, 11c, 13b, 28a, 36, 57c, 58a.
Macro referenced in scrap 3.

Since these arrays are indexed by a block's preorder index, and preorder indices start with one, each array must have `block_count` + 1 elements.

⟨Initialize the dominator data structures 58b⟩ ≡

```
{
    Unsigned_Int array_size = block_count + 1;

    dominator_info = Arena_GetMem(dom_arena, sizeof(Dom_Info)*array_size);
    dom_preorder_list = Arena_GetMem(dom_arena, sizeof(Block *)*array_size);
    buckets = Arena_GetMemClear(dom_arena, sizeof(Unsigned_Int2)*array_size);
}
◇
```

Macro defined by scraps 58bd.
Macro referenced in scrap 57a.

The `walk_tree` function walks the dominator tree in preorder and initializes the `dominator_info` and `dom_preorder_list` arrays. It returns the size of the subtree of the dominator tree rooted at `block`.

⟨Functions 58c⟩ ≡

```
static Unsigned_Int walk_tree(Block *block, Unsigned_Int *index)
{
    Unsigned_Int size = 1;
    Unsigned_Int preorder_index = block->preorder_index;
    Unsigned_Int dom_index = (*index)++;
    Block_List_Node *child;

    dom_preorder_list[dom_index] = preorder_index;
    dominator_info[preorder_index].dominator_index = dom_index;
    Dominator_ForChildren(child, block->dom_node)
        size += walk_tree(child->block, index);

    return dominator_info[preorder_index].dominator_size = size;
}
◇
```

Macro defined by scraps 58c, 59a, 61b, 77b, 78a.
Macro referenced in scrap 3.

The initial call to `walk_tree` should be passed the `start_block` and a pointer to a variable that has been initialized to one.

⟨Initialize the dominator data structures 58d⟩ ≡

```
{
    Unsigned_Int index = 1;
    (Void) walk_tree(start_block, &index);
}
◇
```

Macro defined by scraps 58bd.
Macro referenced in scrap 57a.

Once we have set up our arrays, we can decide if block b_1 dominates block b_2 by performing an ancestor test in the dominator tree. Let p_1 and p_2 be the preorder indices of b_1 and b_2 respectively, and let ND be the number of descendants of b_1. Then b_1 dominates b_2 if and only if:

$$p_1 \leq p_2 < p_1 + ND$$

⟨Functions 59a⟩ ≡
```
static Boolean dominates(Unsigned_Int b1, Unsigned_Int b2)
{
    Unsigned_Int block1 = dom_preorder_list[b1];
    Unsigned_Int block2 = dom_preorder_list[b2];
    Unsigned_Int index1 = dominator_info[block1].dominator_index;
    Unsigned_Int size1 = dominator_info[block1].dominator_size;
    Unsigned_Int index2 = dominator_info[block2].dominator_index;

    return index1 <= index2 && index2 < index1 + size1;
}
◇
```
Macro defined by scraps 58c, 59a, 61b, 77b, 78a.
Macro referenced in scrap 3.

When bucket sorting, we must consider all members of the class and their copies. Refer to the discussion of redundant-store elimination (Section 2.6) for an explanation of the copy lists.

⟨Bucket sort the members of `classes[i]` 59b⟩ ≡
```
{
    Member_Node *node;
    Class_ForAllMembers(node, classes[i].members)
    {
        Unsigned_Int item = node->item;
        Copy_Node *copy_node;

        ⟨Add item to buckets 60a⟩
        Node_ForAllCopies(copy_node, node)
        {
            item = copy_node->item;
            ⟨Add item to buckets 60a⟩
        }
    }
}
◇
```
Macro referenced in scrap 57a.

We can only consider items that define registers, because we can only remove ϕ-nodes or operations that define registers. If buckets[index] is not zero, we have already found an item in this block, so we must decide if item dominates it. Since our algorithm for building SSA numbers the definitions in a block in increasing order of appearance, we can compare the SSA numbers of the two items. The earlier item will have the smaller SSA number. If the existing item dominates this one, we remove the defining operation for item. Otherwise, we remove the defining operation for buckets[index] and overwrite buckets[index] with item.

⟨Add item to buckets 60a⟩ ≡

```
if (IsRegister(item))
{
    Block *block = DefiningBlock(item);
    Unsigned_Int index = dominator_info[block->preorder_index].dominator_index;

    if (buckets[index])
    {
        if (buckets[index] < item)
        {
            Unsigned_Int del_item = item;
            ⟨Remove the definition of del_item 60b⟩
        }
        else
        {
            Unsigned_Int del_item = buckets[index];
            ⟨Remove the definition of del_item 60b⟩
            buckets[index] = item;
        }
        num_removed++;
    }
    else
        buckets[index] = item;
}
◇
```

Macro referenced in scrap 59b.

We remove a ϕ-node by setting all of its arguments to zero. Recall that a zero argument to a ϕ-node represents an undefined value. Therefore, the ConvertFromSSA routine will not insert copies for this ϕ-node. We remove an operation by setting its critical field to FALSE.

⟨Remove the definition of del_item 60b⟩ ≡

```
if (IsPhiNode(del_item))
{
    PhiNodeArg *arg;
    PhiNode_ForAllArgs(arg, DefiningPhiNode(del_item))
        arg->item = 0;
}
else
    DefiningOper(del_item)->critical = FALSE;
◇
```

Macro referenced in scrap 60a.

We use a "two-finger" algorithm for comparing items defined in different blocks. We consider adjacent pairs of non-zero entries in the `buckets` array. The `curr_block` variable points to the first block of the pair in consideration. The `next_block` variable points to the second block. These two variables move through the `buckets` array until each pair of adjacent blocks have been compared. After we have finished processing an entry in `buckets`, we reset its value to zero in preparation for processing the next class.

⟨Compare items defined in different blocks 61a⟩ ≡

```
{
    Unsigned_Int curr_block = move_pointer(buckets, 1);
    Unsigned_Int next_block = move_pointer(buckets, curr_block + 1);

    while (next_block)
    {
        if (dominates(curr_block, next_block))
        {
            DefiningOper(buckets[next_block])->critical = FALSE;
            num_removed++;
            buckets[next_block] = 0;
        }
        else
        {
            buckets[curr_block] = 0;
            curr_block = next_block;
        }
        next_block = move_pointer(buckets, next_block + 1);
    }
    buckets[curr_block] = 0;
}
◇
```

Macro referenced in scrap 57a.

The `move_pointer` function is used to find the next non-empty element of the `buckets` array. If all remaining elements are empty, the function will return zero.

⟨Functions 61b⟩ ≡

```
static Unsigned_Int move_pointer(Unsigned_Int2 *buckets, Unsigned_Int start)
{
    while (start <= block_count)
        if (buckets[start]) return start;
        else start++;

    return 0;
}
◇
```

Macro defined by scraps 58c, 59a, 61b, 77b, 78a.
Macro referenced in scrap 3.

4.2 AVAIL-Based Removal

To perform AVAIL-based removal, we must compute the set of available expressions at the beginning of each block. An expression is available if it is computed along all paths from `start_block`. If an operation computes a value already in the set of available expressions then it can be removed. First, we compute the AVAIL set for each block, then we remove any operations whose result is in the set.

Because of the properties of the partitioning algorithm, the data-flow problem solved is slightly different from the traditional AVAIL problem. We do not consider the killed set for a block because any values killed will be in different classes in the final partition. Consider the code fragment in Figure 4.6. Under the traditional data-flow framework, the assignment to X would kill the Z expression. However, since we know

$$Z \leftarrow X + Y$$
$$X \leftarrow \ldots$$
$$Z \leftarrow X + Y$$

Figure 4.6: AVAIL Example

$$\begin{aligned}
\text{AVAIL_in}_i &= \begin{cases} \emptyset, & \text{if } i \text{ is the entry block} \\ \displaystyle\bigcap_{j \in pred(i)} \text{AVAIL_out}_j, & \text{otherwise} \end{cases} \\
\text{AVAIL_out}_i &= \text{AVAIL_in}_i \cup \texttt{defined}_i
\end{aligned}$$

Figure 4.7: Data-Flow Equations for AVAIL

that we have partitioned the values into congruence classes, if the two assignments to Z are congruent, then the second one is redundant and can be removed. One way this can happen is if the assignment to X is congruent to the definition of X that reaches the first assignment to Z. Note that if the assignment to X caused the two assignments to Z to have different values, then they would not be congruent to each other, and they would be assigned different locations in the bit vectors. The data-flow equations we use are shown in Figure 4.7.

⟨Optionally perform AVAIL-based removal 62⟩ ≡

```
if (do_avail)
{
    Arena avail_arena = Arena_Create();

    if (debug >= MAJOR_PHASES)
        fprintf(stderr, "Perform AVAIL-based removal\n");

    ⟨Compute the AVAIL set for each block 63b⟩
    ⟨Remove operations whose result is in AVAIL 68a⟩

    Arena_Destroy(avail_arena);
}
◇
```

Macro referenced in scrap 4.

We will store some information for each block in the `block_extension` field contained in the `Block` structure. The `ConvertToSSA` routine stored the ϕ-nodes there, but since the routine has already been converted out of SSA form, we don't need them. However, we cannot redefine the `block_extension` structure so we will define a new structure and type cast the pointers to it.

⟨Type Declarations 63a⟩ ≡
```
typedef struct
{
    VectorSet defined;    /*  The set of values defined inside this block  */
    VectorSet AVAIL_in;   /*  The set of available expressions at the beginning
                              of this block  */
    VectorSet AVAIL_out;  /*  The set of available expressions at the end
                              of this block  */
    Boolean dirty;        /*  A flag indicating if AVAIL_in should be recomputed  */
} AVAIL_Extension;
```
◇

Macro defined by scraps 11ab, 13a, 22c, 29b, 43a, 57b, 63a.
Macro referenced in scrap 3.

We compute the AVAIL sets using iterative data-flow analysis. The `defined` set can be determined locally for each block. The analysis starts with an initial approximation to the solution and iteratively refines the solution until it stabilizes. We are only interested in available registers, so all bit vectors will have `num_registers` elements.

⟨Compute the AVAIL set for each block 63b⟩ ≡
```
{
    Unsigned_Int set_size = num_registers;

    if (debug >= MINOR_PHASES)
        fprintf(stderr, "    Compute the AVAIL set for each block\n");

    ⟨Find the defined set for each block 64a⟩
    ⟨Initialize the AVAIL data-flow problem 64c⟩
    ⟨Find the AVAIL set for all blocks 66a⟩
}
```
◇

Macro referenced in scrap 62.

The `defined` set will contain all the registers defined in a block.

⟨Find the `defined` set for each block 64a⟩ ≡

```
{
    Block *block;
    ForAllBlocks(block)
    {
        VectorSet defined;
        Inst *inst;

        ⟨Allocate the block_extension and the defined set 64b⟩
        Block_ForAllInsts(inst, block)
        {
            Operation **oper_ptr;
            Inst_ForAllOperations(oper_ptr, inst)
            {
                Operation *oper = *oper_ptr;
                Unsigned_Int2 *reg_ptr;

                Operation_ForAllDefs(reg_ptr, oper)
                    VectorSet_Insert(defined, *reg_ptr);
            }
        }
    }
}
```

◇

Macro referenced in scrap 63b.

⟨Allocate the `block_extension` and the `defined` set 64b⟩ ≡

```
block->block_extension = Arena_GetMem(avail_arena, sizeof(AVAIL_Extension));
defined = ((AVAIL_Extension *) block->block_extension)->defined =
    VectorSet_Create(avail_arena, set_size);
```

◇

Macro referenced in scrap 64a.

⟨Initialize the AVAIL data-flow problem 64c⟩ ≡

```
{
    Block *block;
    ForAllBlocks(block)
        if (block == start_block)
            ⟨Initialize AVAIL for the start_block 65a⟩
        else
            ⟨Initialize AVAIL for the other blocks 65b⟩
}
```

◇

Macro referenced in scrap 63b.

The initial value for the AVAIL set for the **start_block** is the empty set. In fact, we don't even need to allocate a set for it. We know that the AVAIL set for the **start_block** is definitely the empty set, so we set its **dirty** flag to **FALSE**.

⟨Initialize AVAIL for the `start_block` 65a⟩ ≡

```
{
    AVAIL_Extension *extension = (AVAIL_Extension *) start_block->block_extension;

    extension->AVAIL_out = VectorSet_Create(avail_arena, set_size);
    extension->AVAIL_in = NULL;
    extension->dirty = FALSE;
}
◇
```

Macro referenced in scrap 64c.

The initial value for the AVAIL set for blocks other than the **start_block** is the set of all registers. Since this is merely an initial guess for the AVAIL set, we set the **dirty** flag to **TRUE**.

⟨Initialize AVAIL for the other blocks 65b⟩ ≡

```
{
    AVAIL_Extension *extension = (AVAIL_Extension *) block->block_extension;
    VectorSet AVAIL_in = extension->AVAIL_in = VectorSet_Create(avail_arena, set_size);
    VectorSet AVAIL_out = extension->AVAIL_out = VectorSet_Create(avail_arena, set_size);

    VectorSet_Complement(AVAIL_in, AVAIL_in);
    VectorSet_Complement(AVAIL_out, AVAIL_out);
    extension->dirty = TRUE;
}
◇
```

Macro referenced in scrap 64c.

We perform iterative data-flow analysis to find AVAIL for all blocks. We step through the blocks in reverse postorder because it makes the analysis terminate faster. Notice that we only compute the information for a block if it is marked `dirty`. The algorithm terminates when an iteration completes with no changes to the AVAIL sets.

⟨Find the AVAIL set for all blocks 66a⟩ ≡

```
{
    Boolean changed;
    VectorSet temp = VectorSet_Create(avail_arena, set_size);

    do
    {
        Block *block;

        changed = FALSE;
        ForAllBlocks_rPostorder(block)
        {
            AVAIL_Extension *extension = (AVAIL_Extension *) block->block_extension;

            if (extension->dirty)
            {
                ⟨Start with AVAIL_out from the first predecessor 66b⟩
                ⟨Look at AVAIL_out for the other predecessors 67a⟩
                ⟨See if we have changed AVAIL_in, setting changed 67b⟩
            }
        }
    } while (changed);
}
◇
```

Macro referenced in scrap 63b.

The predecessors of a block are stored in a linked list pointed to by the `pred` field of the `Block` structure. We'll use the `AVAIL_out` set from the first predecessor as our initial value for the `AVAIL_in` set. The set will be computed in `temp` so we can see if we have changed anything.

⟨Start with `AVAIL_out` from the first predecessor 66b⟩ ≡

```
{
    Edge *edge = block->pred;
    extension->dirty = FALSE;
    VectorSet_Copy(temp, ((AVAIL_Extension *) edge->pred->block_extension)->AVAIL_out);
}
◇
```

Macro referenced in scrap 66a.

After we have looked at the **AVAIL_out** set from the first predecessor, we iterate through the other predecessors by starting with **block->pred->next_pred** (the second predecessor) and moving through the **next_pred** pointers until we reach a **NULL** pointer. For each of these predecessors, we update the intersection of the **AVAIL_out** sets. This intersection will be the **AVAIL_in** set for the block.

⟨Look at **AVAIL_out** for the other predecessors 67a⟩ ≡

```
{
    Edge *edge;
    for (edge = block->pred->next_pred;
         edge;
         edge = edge->next_pred)
    {
        AVAIL_Extension *extension = (AVAIL_Extension *) edge->pred->block_extension;
        VectorSet_Intersect(temp, temp, extension->AVAIL_out);
    }
}
◇
```

Macro referenced in scrap 66a.

If the AVAIL set for this block has changed, we must recompute the **AVAIL_out** set and mark all the successor blocks **dirty**.

⟨See if we have changed **AVAIL_in**, setting changed 67b⟩ ≡

```
if (!VectorSet_Equal(extension->AVAIL_in, temp))
{
    changed = TRUE;
    VectorSet_Copy(extension->AVAIL_in, temp);
    VectorSet_Union(extension->AVAIL_out, extension->AVAIL_in, extension->defined);
    ⟨Mark all the successors dirty 67c⟩
}
◇
```

Macro referenced in scrap 66a.

⟨Mark all the successors dirty 67c⟩ ≡

```
{
    Edge *edge;
    Block_ForAllSuccs(edge, block)
        ((AVAIL_Extension *) edge->succ->block_extension)->dirty = TRUE;
}
◇
```

Macro referenced in scrap 67b.

Once we have computed AVAIL for each block, we are ready to start removing operations from the routine. We step through the blocks and use the **AVAIL_in** set as a guide for removing operations.

⟨Remove operations whose result is in AVAIL 68a⟩ ≡

```
{
    Unsigned_Int num_removed = 0;
    Block *block;

    if (debug >= MINOR_PHASES)
        fprintf(stderr, "    Remove operations whose result is in AVAIL\n");

    ForAllBlocks(block)
    {
        VectorSet AVAIL_in = ((AVAIL_Extension *) block->block_extension)->AVAIL_in;
        Inst *inst;
        Block_ForAllInsts(inst, block)
        {
            Operation **oper_ptr;
            Inst_ForAllOperations(oper_ptr, inst)
            {
                Operation *oper = *oper_ptr;
                ⟨If oper is in AVAIL, remove it 68b⟩
            }
        }
    }

    if (debug)
        fprintf(stderr, "        %d operations removed based on AVAIL.\n",
                num_removed);
}
◇
```

Macro referenced in scrap 62.

We can remove any **LOAD** or **EXPR** operation whose result register is in the **AVAIL_in** set. If the operation is not removed, we add its defined registers to the **AVAIL_in** set because they are available to operations later in the block. Notice that we can safely add these before we finish processing all operations in the instruction[1]. This is because if two operations in the same instruction compute the same value, we can eliminate either one of them. We choose to eliminate the second.

⟨If `oper` is in AVAIL, remove it 68b⟩ ≡

```
{
    Unsigned_Int details = opcode_specs[oper->opcode].details;
    Unsigned_Int reg = oper->arguments[oper->referenced];

    if ((details & (LOAD | EXPR)) && VectorSet_Member(AVAIL_in, reg))
    {
        oper->critical = FALSE;
        num_removed++;
    }
    else
        ⟨Add all the defined registers to AVAIL_in 69⟩
}
◇
```

Macro referenced in scrap 68a.

[1] All operations in an `ILOC` instruction are considered to execute in parallel.

⟨Add all the defined registers to `AVAIL_in` 69⟩ ≡

```
{
    Unsigned_Int2 *reg_ptr;
    Operation_ForAllDefs(reg_ptr, oper)
        VectorSet_Insert(AVAIL_in, *reg_ptr);
}
```

◇

Macro referenced in scrap 68b.

Appendix A

Memory Management

Figure A.1 shows the lifetimes of the major data structures used in **gval**. The major phases of execution are shown at the top of the diagram. The column marked *Data* contains variable names and descriptions of the data structures. The bars in the center of the diagram represent the lifetimes of the data. The column marked *Arena* contains the name of the arena where the data is allocated. Notice that all data allocated in `temp_arena` is short-lived. These objects will be freed using `Arena_Mark` and `Arena_Release`.

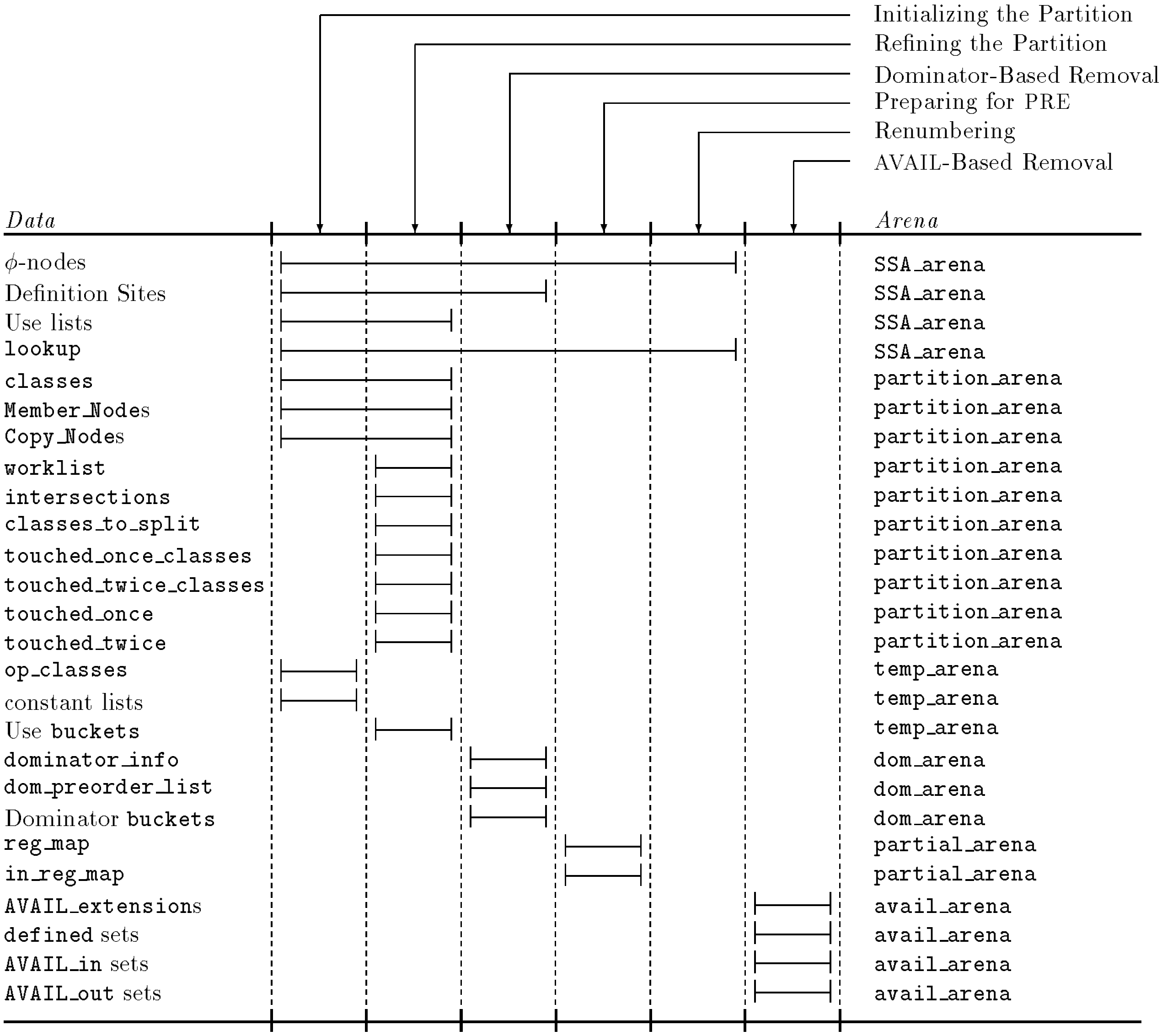

Figure A.1: Memory Use Diagram for **gval**

Appendix B

Debugging Output

B.1 Printing a Histogram of the Congruence Classes

If the user specified the `-h` option, we will print a histogram of the sizes of the congruence classes to `stderr`. This is useful for assessing the effectiveness of the optimization. The histogram is printed in three columns:

1. The size (number of members) of the class
2. The number of classes with this size
3. A number of "*"s to represent the number of classes

The last line of the histogram shows the total number of classes and members. Figure B.1 shows an example histogram.

```
Class     Number of
Size      Classes
1       | 39     ***************************************************************
2       | 8      **************

6       | 1      *

19      | 1      *

Total: 49 classes with 80 members
```

Figure B.1: Example Histogram

⟨Optionally print a histogram 72⟩ ≡

```
if (hist)
{
    Unsigned_Int max_size = 0;
    Unsigned_Int2 *counters;
    Unsigned_Int max_counter = 0;

    ⟨Find the maximum class size 73a⟩
    ⟨Count how many partitions of each size there are 73b⟩
    ⟨Print the histogram 73c⟩
    ⟨Print the total number of classes and members 74b⟩
}
◇
```

Macro referenced in scrap 13c.

⟨Find the maximum class size 73a⟩ ≡

```
{
    Unsigned_Int i;
    for (i = 1; i < num_classes; i++)
    {
        Unsigned_Int size = classes[i].num_members;
        if (size > max_size) max_size = size;
    }
}
```

◇

Macro referenced in scrap 72.

⟨Count how many partitions of each size there are 73b⟩ ≡

```
{
    Unsigned_Int i;
    counters = Arena_GetMemClear(partition_arena, (max_size+1)*sizeof(Unsigned_Int2));

    for (i = 1; i < num_classes; i++)
    {
        Unsigned_Int size = classes[i].num_members;
        Unsigned_Int counter = ++counters[size];
        if (counter > max_counter) max_counter = counter;
    }
}
```

◇

Macro referenced in scrap 72.

If there are no classes of a given size, a blank line will be printed, but there will be at most one blank line at a time.

⟨Print the histogram 73c⟩ ≡

```
{
    Unsigned_Int i;
    Boolean blank_line = FALSE;

    fprintf(stderr, "Class\t  Number of\n");
    fprintf(stderr, "Size\t  Classes\n");
    for (i = 1; i <= max_size; i++)
    {
        Unsigned_Int counter = counters[i];
        Unsigned_Int num_stars = counter*60/max_counter;

        if (counter)
        {
            fprintf(stderr, "%d\t| %d\t", i, counter);
            ⟨Print num_stars stars 74a⟩
            fprintf(stderr, "\n");
            blank_line = FALSE;
        }
        else if (!blank_line)
        {
            fprintf(stderr, "\n");
            blank_line = TRUE;
        }
    }
}
```

◇

Macro referenced in scrap 72.

Notice that we always print at least one star.

⟨Print num_stars stars 74a⟩ ≡
```
{
    Unsigned_Int j;

    if (!num_stars) num_stars = 1;
    for (j = 0; j < num_stars; j++)
        fprintf(stderr, "*");
}
```
◇

Macro referenced in scrap 73c.

Once we have completed printing the histogram, we print a summary showing the total number of classes and members.

⟨Print the total number of classes and members 74b⟩ ≡
```
fprintf(stderr, "\nTotal: %d classes with %d members\n",
        num_classes-1, defCount-1);
```
◇

Macro referenced in scrap 72.

B.2 Printing the Partiton

It is useful to be able to print either the initial or the final partition. We will do so if the `debug` level is higher than `PARTITION`. Figure B.2 shows an example of a final partition. Each line displays the SSA names of the members of one class in curly braces ({}). The number after the pound sign (#) is the number of members of the class. If a star (*) appears before the class number, this indicates that the class represents ϕ-nodes.

⟨Optionally print the initial partition 74c⟩ ≡
```
if (debug >= PARTITION)
{
    fprintf(stderr, "Initial partition:\n");
    ⟨Print the partition 75⟩
    ⟨Check the lookup table 77a⟩
}
```
◇

Macro referenced in scrap 13c.

⟨Optionally print the final partition 74d⟩ ≡
```
if (debug >= PARTITION)
{
    fprintf(stderr, "\nFinal partition:\n");
    ⟨Print the partition 75⟩
    ⟨Check the lookup table 77a⟩
}
```
◇

Macro referenced in scrap 13c.

```
Final partition:
Equiv. class 1 = #1{ 1 }
Equiv. class 2 = #1{ 2 }
Equiv. class *3 = #1{ 17 }
Equiv. class *4 = #1{ 59 }
Equiv. class *5 = #1{ 76 }
Equiv. class *6 = #1{ 25 }
Equiv. class *7 = #1{ 15 }
Equiv. class 8 = #1{ 3 }
Equiv. class 9 = #1{ 4 }
Equiv. class 10 = #1{ 5 }
Equiv. class 11 = #1{ 6 }
Equiv. class 12 = #2{ 7 18 }
Equiv. class 13 = #19{ 8 10 12 19 22 60 62 67 69 77 26 28 33 35 40 42 47 49 54 }
Equiv. class 14 = #6{ 63 70 29 36 43 50 }
Equiv. class 15 = #2{ 9 21 }
Equiv. class 16 = #1{ 39 }
Equiv. class 17 = #1{ 53 }
Equiv. class 18 = #1{ 78 }
Equiv. class 19 = #1{ 68 }
Equiv. class 20 = #2{ 51 37 }
⋮
Equiv. class 45 = #1{ 11 }
Equiv. class 46 = #2{ 34 48 }
Equiv. class 47 = #1{ 71 }
Equiv. class 48 = #1{ 72 }
Equiv. class 49 = #1{ 73 }
lookup is OK.
```

Figure B.2: Example Final Partition

We print the classes in order, displaying the class number, the number of members, and the `members` list. If a class represents ϕ-nodes, we print a "*" just in front of its class number (See class 2 in Figure B.2).

⟨Print the partition 75⟩ ≡

```
{
    Unsigned_Int i;
    for (i = 1; i < num_classes; i++)
    {
        Member_Node *members = classes[i].members;

        if (IsPhiNode(members->next->item))
            fprintf(stderr, "Equiv. class *%d = #%d", i, classes[i].num_members);
        else
            fprintf(stderr, "Equiv. class %d = #%d", i, classes[i].num_members);

        ⟨Print the members list 76⟩
    }
}
◇
```

Macro referenced in scraps 74cd.

We print the members of a class by iterating through the list. If any member has a non-empty copy list, we print the copies in parentheses after the member. Refer to the discussion of redundant-store elimination (Section 2.6) for an explanation of the copy lists.

⟨Print the `members` list 76⟩ ≡

```
fprintf(stderr, "{");
{
    Member_Node *node;
    Class_ForAllMembers(node, members)
    {
        fprintf(stderr, " %d", node->item);
        if (node->copies)
        {
            Copy_Node *copy_node;

            fprintf(stderr, " (");
            Node_ForAllCopies(copy_node, node)
                fprintf(stderr, " %d", copy_node->item);
            fprintf(stderr, " )");
        }
    }
    fprintf(stderr, " }\n");
}
◇
```

Macro referenced in scrap 75.

Since the lookup array must be manipulated consistently with the classes array, it is useful to check that we have done so. For each class, we iterate through its members list and check the lookup entry for each item. If the lookup entry is correct, the class_num and node fields must have the proper values. Otherwise, we print an error message.

⟨Check the lookup table 77a⟩ ≡

```
{
    Unsigned_Int i;
    Boolean error = FALSE;

    for (i = 1; i < num_classes; i++)
    {
        Member_Node *node;
        Class_ForAllMembers(node, classes[i].members)
        {
            Unsigned_Int item = node->item;
            if (lookup[item].class_num != i)
            {
                fprintf(stderr, "Error: lookup[%d].class_num = %d",
                        item, lookup[item].class_num);
                fprintf(stderr, " (should be %d)\n", i);
                error = TRUE;
            }
            if (lookup[item].node != node)
            {
                fprintf(stderr, "Error: lookup[%d].node is incorrect\n", item);
                error = TRUE;
            }
        }
    }
    if (!error) fprintf(stderr, "lookup is OK.\n");
}
◇
```

Macro referenced in scraps 74cd.

B.3 Printing Dominator Information

The tree_printer function can be passed to Block_Dump_All to display the dominator tree information about a block in the CFG.

⟨Functions 77b⟩ ≡

```
static Void tree_printer(Block *block)
{
    Unsigned_Int index = block->preorder_index;

    fprintf(stderr, "dominator index = %d,",
            dominator_info[index].dominator_index);
    fprintf(stderr, " dominator size = %d.\n",
            dominator_info[index].dominator_size);
}
◇
```

Macro defined by scraps 58c, 59a, 61b, 77b, 78a.
Macro referenced in scrap 3.

⟨Print dominator tree info 77c⟩ ≡

```
Block_Dump_All(tree_printer, 0);
◇
```

Macro never referenced.

B.4 Printing AVAIL Information

The `AVAIL_printer` function can be passed to `Block_Dump_All` to display the information about available expressions for a block in the CFG.

⟨Functions 78a⟩ ≡

```
static Void AVAIL_printer(Block *block)
{
    AVAIL_Extension *extension = (AVAIL_Extension *) block->block_extension;

    if (extension)
    {
        if (extension->defined)
        {
            fprintf(stderr, "defined = ");
            VectorSet_Dump(extension->defined);
        }
        else
            fprintf(stderr, "defined = NULL.\n");

        if (extension->AVAIL_in)
        {
            fprintf(stderr, "AVAIL in = ");
            VectorSet_Dump(extension->AVAIL_in);
        }
        else
            fprintf(stderr, "AVAIL in = NULL\n");

        if (extension->AVAIL_out)
        {
            fprintf(stderr, "AVAIL out = ");
            VectorSet_Dump(extension->AVAIL_out);
        }
        else
            fprintf(stderr, "AVAIL out = NULL\n");
    }
}
```

◇

Macro defined by scraps 58c, 59a, 61b, 77b, 78a.
Macro referenced in scrap 3.

⟨Print AVAIL info 78b⟩ ≡

```
Block_Dump_All(AVAIL_printer, 0);
```

◇

Macro never referenced.

Appendix C

Indices

C.1 Index of File Names

C.2 Index of Macro Names

C.3 Index of Identifiers

C.4 Function Prototypes

The **walk_tree** function walks the dominator tree in preorder and initializes the **dominator_info** and **dom_preorder_list** arrays. It returns the size of the subtree of the dominator tree rooted at **block**. The initial call to **walk_tree** should be passed the **start_block** and a pointer to a variable that has been initialized to one.

⟨Prototypes 85a⟩ ≡

```
static Unsigned_Int walk_tree(Block *block, Unsigned_Int *index);
```

◇

Macro defined by scraps 85abc, 86.
Macro referenced in scrap 3.

Once we have used **walk_tree** to initizliae the **dominator_info** and **dom_preorder_list** arrays, we can use the function **dominates** to determine if one block dominates another. The arguments to **dominates** should be the preorder indices in the dominator tree of the two blocks.

⟨Prototypes 85b⟩ ≡

```
static Boolean dominates(Unsigned_Int b1, Unsigned_Int b2);
```

◇

Macro defined by scraps 85abc, 86.
Macro referenced in scrap 3.

The **move_pointer** function is used to find the next non-empty element of the **buckets** array. If all remaining elements are empty, the function will return zero.

⟨Prototypes 85c⟩ ≡

```
static Unsigned_Int move_pointer(Unsigned_Int2 *buckets, Unsigned_Int start);
```

◇

Macro defined by scraps 85abc, 86.
Macro referenced in scrap 3.

The `tree_printer` and `AVAIL_printer` functions can be passed to `Block_Dump_All` to display the dominator tree information or the AVAIL information, respectively, about a block in the CFG.

⟨Prototypes 86⟩ ≡
```
static Void tree_printer(Block *block);
static Void AVAIL_printer(Block *block);
```
◇

Macro defined by scraps 85abc, 86.
Macro referenced in scrap 3.

Bibliography

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, Massachusetts, 1974.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[3] Bowen Alpern, Mark N. Wegman, and F. Kenneth Zadeck. Detecting equality of variables in programs. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 1–11, San Diego, California, January 1988.

[4] Preston Briggs. The massively scalar compiler project. Technical report, Rice University, July 1994. Preliminary version available via anonymous ftp.

[5] Preston Briggs and Keith D. Cooper. Effective partial redundancy elimination. *SIGPLAN Notices*, 29(6):159–170, June 1994. *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation.*

[6] Preston Briggs and Rob Shillner. Elimination of partial redundancies. Technical report, Rice University, March 1993. Available via anonymous ftp.

[7] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

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