

THESIS

GARBAGE ELIMINATION IN SA-C HOST CODE

Submitted by
Steve Segreto
Department of Computer Science

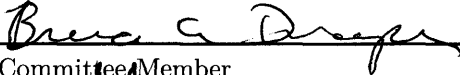
In partial fulfillment of the requirements
for the Degree of Master of Science
Colorado State University
Fort Collins, Colorado
Fall 2001

COLORADO STATE UNIVERSITY

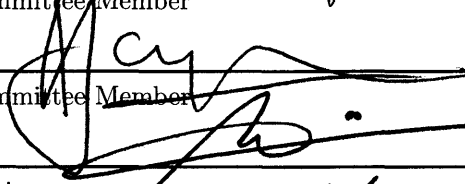
September 4, 2001

WE HEREBY RECOMMEND THAT THE THESIS PREPARED UNDER OUR SUPERVISION BY STEVE SEGRETO ENTITLED GARBAGE ELIMINATION IN SA-C HOST CODE BE ACCEPTED AS FULFILLING IN PART REQUIREMENTS FOR THE DEGREE OF MASTER OF SCIENCE.

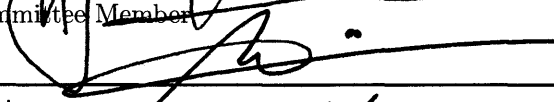
Committee on Graduate Work




Committee Member



Committee Member



Adviser



Department Head

ABSTRACT OF THESIS

GARBAGE ELIMINATION IN SA-C HOST CODE

Single-assignment C (SA-C) is a functional programming language with a rich instruction set designed to create and manipulate arrays using array slices and window generators. It is well-suited for the fields of graphics, AI and image processing within reconfigurable computing environments. Garbage is defined as any SA-C array data which is unused or unreferenced in the host code program heap at any time. Garbage must not be created and it must be freed as soon as possible. In this paper it will be shown that the single-assignment properties of the language create garbage when single-assignment occurs in loops. This behavior is studied and a static solution is presented called pointer reuse. The non-circular aliases resulting from strict single-assignment alias creation coupled with the side-effect free nature of statement blocks lead to a dynamic reference counting technique which can provide immediate elimination of garbage. Aliases and special loop-carried variable dependencies complicate matters further and are examined in this paper.

Steve Segreto
Department of Computer Science
Colorado State University
Fort Collins, Colorado 80523
Fall 2001

ACKNOWLEDGEMENTS

Grateful acknowledgements to my advisor, Dr. Wim Böhm, who kept my flame of enthusiasm lit even in the rain.

DEDICATION

Dedicated to my mom and my dad, and my brothers, Matt and Dave. Thanks for all of your love and support.

TABLE OF CONTENTS

1	Introduction	1
1.1	Problem Statement	1
1.2	Sources of Garbage	2
1.3	Approach	3
2	Background	4
2.1	The SA-C Language	4
2.1.1	Statements	4
2.1.2	Statement Blocks	5
2.1.3	While Loops	6
2.1.4	For Loops	6
2.1.5	User-Defined Functions	7
2.2	SA-C Arrays	7
2.2.1	Array Operations	8
2.2.2	Array-Component Generators	9
2.2.3	Structure-building Operations	9
2.3	DDCF Graphs	10
2.3.1	Nodes that create arrays	10
2.3.2	Nodes that consume arrays	12
2.4	Host Code Generation	13
2.4.1	Aliases	14
2.4.1.1	Conditional Expressions	15
2.4.1.2	Dope Vectors	16
2.4.1.3	Colon Notation	16

2.4.1.4	Array Reference Example	18
2.4.2	Host Code Arrays	19
2.4.2.1	Dynamic Array Properties	19
3	Statically Determinable Garbage	21
3.1	Overview	21
3.2	Array of Arrays	22
3.3	Basic Single Assignment	23
3.4	Creator Reference Scheme	25
3.5	Nextified Single Assignment	26
3.6	Basic Nextified Single Assignment	28
3.7	Full Nextified Alias	30
3.8	Partial Nextified Alias	30
3.9	Nextified Single Assignment Aliasing	32
3.10	Static Dependency Analysis	33
3.10.1	Arrays Returned by Functions	34
4	Reference Counting	35
4.1	Overview	37
4.2	Loop Levels	38
4.3	Basic Approach	39
4.4	Initializing the Reference Count	40
4.5	Incrementing the Reference Count	40
4.5.1	Edge Splits	40
4.5.2	Aliases	42
4.6	Decrementing the Reference Count	42
4.6.1	Compound DDCF Nodes	43
4.6.2	Simple DDCF Nodes	44

4.7	Unfinalized Nextified Variables	44
5	Performance Evaluation	46
5.1	Evaluation of Reference Counting	47
5.2	Evaluation of Pointer Reuse	48
5.3	Performance Results	48
5.4	Speed Performance	50
5.5	Complexity Analysis	50
6	Conclusions	53
	References	55

LIST OF TABLES

2.1	Structure-Building Operations	9
2.2	Array-Creation Nodes	11
2.3	Array Consumer Nodes	12
5.1	Test Suite A	49
5.2	Test Suite B	50

LIST OF FIGURES

1.1	SA-C Compilation Paths	2
2.1	Example of DDCF Sink Node	12
2.2	Example of Logical and Physical SA-C array.	13
2.3	Logical References forming an Alias Chain	14
2.4	Host Code Aliases	15
2.5	Dope Vector	18
3.1	Array-of-arrays	22
3.2	Loop with Array Definition	23
3.3	Basic Single Assignment Data Flow	24
3.4	Nextified Variable Inside of Loop	26
3.5	Nextified Variable Outside of Loop	26
3.6	Basic Nextified Single Assignment	28
3.7	Full Nextified Alias	30
3.8	Partial Nextified Alias	31
3.9	Nextified Single Assignment Aliasing	32
4.1	Example of Circular Aliasing	35
4.2	Example of Variable Shadowing	36
4.3	Host Code Example of Variable Shadowing	37
4.4	Host Code Example of Circular Aliasing	37
4.5	Example of Loop Levels	38
4.6	Example of DDCF Input Edge Split	41

4.7	Example of DDCF Generator Node	44
4.8	Example of Unfinalized Nextified Variable	45
5.1	Example of Nested DDCF Graphs	51
5.2	Example of Wide DDCF Graphs	52
5.3	Example of Long DDCF Graphs	52

Chapter 1

Introduction

The Cameron research group (www.cs.colostate.edu/cameron) uses the SA-C language for applications that do computationally intensive work with large image matrices. SA-C is a single-assignment functional programming language. Programs written in SA-C are partly compiled to host code and partly to equivalent programs in a hardware description language called VHDL. VHDL is used to represent control and data flow for a piece of hardware called an FPGA (field programmable gate array), which is widely used in reconfigurable computing. There are two compilation paths for a SA-C program. The input SA-C program is parsed into a data-dependence control-flow graph (DDCF graph), which can either be translated into *host code* plus a data-flow graph (DFG) or into *host code* only. The DFG can be used to create a VHDL specification for the SA-C code, so it can be run on a reconfigurable computing system. A RCS is a board with an FPGA and local memories. Host code is defined as a C program, created by a back-end program called the code generator. The host code program can be compiled using any traditional C compiler into a von Neumann machine executable which produces output equivalent to that of the FPGA. The two paths are illustrated in Figure 1.1. This paper will focus on the second compilation path, in which SA-C code is compiled to equivalent host code. Throughout the paper there will be a distinction between the code generator (a back-end program which is part of the SA-C compiler) and its output (host code).

1.1 Problem Statement

Garbage can be defined as any unused or unreferenced memory which exists during a program's

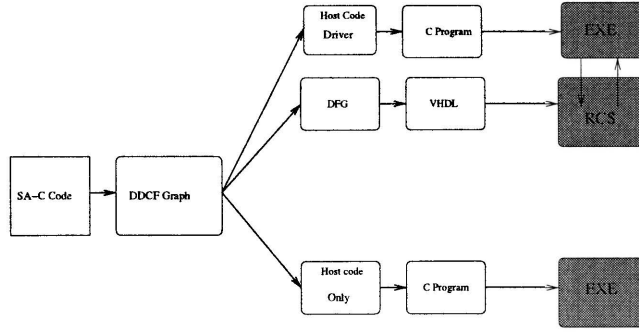


Figure 1.1: SA-C Compilation Paths

execution. The purpose of this thesis is to eliminate garbage from the host code. Garbage must not be created inside the host code and any data which becomes garbage must be freed as soon as possible. A *heap leak* is a special case of garbage in which a pointer p references some block of memory A, but is then re-assigned to reference a different block of memory B. If p is the last reference to the memory block A, and A has not been freed from the program heap, A is a heap leak. Heap leaks can be statically resolved, but any piece of data can be dynamically subjected to "live data analysis" through a process called *reference counting*. Reference counting entails keeping a counter showing how many variables reference a particular piece of data at any given time. When a piece of data is first created, the single copy of the pointer to it causes the reference count to be initialized to one. This count is incremented when copies of pointers to the structure are created, which keeps the data alive longer. The count is decremented when copies of pointers to the structure are destroyed. When the count reaches zero, this indicates that the data is **garbage**. This garbage is immediately freed and can no longer be accessed.

1.2 Sources of Garbage

For any particular piece of host code, the garbage this paper is interested in comes in the form of array data. Array data is allocated and stored in the host code program's heap. There are two ways array data can become garbage. The first is by singly-assigning to the data's pointer, overwriting the address to the first data block with the address to a different data block. The second way occurs when a reference to the data exists but will never be used again. A static solution (pointer reuse)

and a dynamic solution (reference counting) are presented herein to remedy both these situations. After these techniques are applied, **garbage-free** host code is expected to be produced for any given SA-C code.

1.3 Approach

The single-assignment properties of the language create garbage when array single-assignment occurs inside a loop and the resulting array definition is not an alias (as defined in section 1.4.1). This behavior is studied and a static solution called **pointer reuse** is presented. The non-circular aliases that result from strict single-assignment alias creation coupled with the external side-effect free nature of statement blocks enable a dynamic **reference counting** technique to work for the SA-C language. The benefits in terms of space performance that result from pointer reuse and reference counting are given in the performance evaluation section of the paper. The test suites used to ensure the correctness of garbage-free host code are also presented. The paper concludes with some remarks about garbage elimination in single-assignment languages and implementation details in general.

Chapter 2

Background

2.1 The SA-C Language

The SA-C language is a single-assignment, expression-oriented language with a rich instruction set designed to create and manipulate arrays using array slices and window generators. It capitalizes on the parallelism inherent in many matrix operations. SA-C has scalar base types, complex types, and arrays composed of scalars or complex numbers. The SA-C Language - Version 1.0 is an excellent introductory paper by (Hammes and Böhm, 99) featuring full example programs and commentary. SA-C has rectangular multidimensional arrays with components limited to scalars and complex numbers [3]. The language allows the programmer to easily generate array slices and sub-arrays through special loop constructs. Loops can also create arrays or reduce arrays down to scalar values. More details of SA-C arrays are presented in the next section.

2.1.1 Statements

SA-C has only three statements: assignment, **print** and **assert** [3]. There are two flavors of assignment, *single-assignment* (the default flavor) and *nextified single assignment* (available by using a special **next** keyword). The latter will be discussed when loops with loop-carried dependencies are introduced. The first type of assignment statements form the bulk of statement blocks and differ from C in three ways. First, all assignments declare and define a variable simultaneously, including arrays. Second, multiple values can be assigned concurrently [3]. Third, the assignment is a statement not an expression, so it does not return a value [3]. Apart from producing a value a function (or other operator), must not change the *state* of the computation. To do so implies the function has

a **side-effect**. For example using read/write to change a file's state or updating a global variable are both examples of operations which have side-effects. An individual SA-C assignment statement has no external side-effects at the language definition level. At the implementation level, statements in loops *may* allocate new memory from the program heap, losing the reference to the memory from the last loop iteration. Thus some SA-C statements have the property of producing garbage. This is not exactly a side-effect, and is called the **single-assignment property**.

2.1.2 Statement Blocks

A block of statements can occur as an expression [3]. SA-C blocks differ from C blocks in that they always produce return values via the keyword **return**, which is followed by a parenthesized, comma-separated list of expressions. This property, coupled with the single-assignment property of statements described above, changes the scoping rules for SA-C variables slightly. Similarly to C, a variable which is declared and defined on the left-hand side of an assignment statement comes into scope beginning with the semicolon that terminates that statement and ending with the closing parentheses that ends the statement block containing the statement [3]. Unlike C, a newly declared variable may *shadow* a previously declared variable of the same name which is in the same lexical range. In other words, variable shadowing allows a programmer to recycle variable names without concern. Note that a statement block has absolutely no external side-effects. The values of variables after exiting a statement block are the same as when entering that block. This includes the bodies of functions and loops, though additional scoping issues arise in the context of **for** loops (described later) [3]. Not only does every statement block return one or more values, conditional expressions, loops and functions also return one or more values. This means every **if** must have a matching **else** [3]. The **switch** expression takes an ordinal switch key to determine which case branch to execute. Each branch can have an optional block of statements, and must return the same number and types of values. Every **if** expression can be represented by an equivalent **switch** expression. While it is true that statement blocks have no *external* side-effects, it will be shown in the next section that some statement blocks have some implementation effects, such as garbage creation (loops) or alias creation (conditional expressions).

2.1.3 While Loops

There are two kinds of loops in SA-C, denoted by the keywords **while** and **for** [3]. The **while** loop is more general and should be used when the number of loop iterations is not known. Because of single-assignment, a loop control variable declared and defined inside a loop body cannot be properly updated without additional syntax. The following SA-C code does not increment the loop control variable *n* as expected, instead it loops forever:

```
uint8 n = 0;
while (n < 10)
{
    uint8 n = n + 1;    // n = 1 always
} return (n);
```

It should be rewritten to properly increment the loop control variable:

```
uint8 n = 0;
while (n < 10)
{
    next n = n + 1;    // The old n can be used to compute the new n
} return (final (n));
```

Here the new value for an iteration is assigned by using the keyword **next** in place of the type declaration that would otherwise occur [3]. SA-C allows a **next** assignment to each variable at most once *within the body of a loop*, in addition to an assignment that takes place prior to the loop [3]. In other words, nextified variable shadowing is **not** allowed. The keyword **next** is used by the SA-C language to denote that a loop-carried dependency exists for a particular variable. Any variable with a **next** assignment *must* have been defined before the loop is entered, so the use of the variable in iteration one will have a data type and value [3]. The variable on the left-hand side of the **next** assignment is termed a **nextified variable**. A loop can return a **final** value for any nextified variable, but because of the side-effect free nature of statement blocks, it will leave the outer value of the nextified variable unchanged [3].

2.1.4 For Loops

The **for** loop is used when the number of loop iterations is known before the loop starts [3]. There are two flavors of **for** loop, those with loop carried dependencies, expressed using **next** assignments

and "forall" loops without loop carried dependencies [3]. Both kinds have the same three parts: a *generator*, a *body*, and a *return expression*.

SA-C **for** loops use generators both for loop predicates and to produce data values for each iteration. SA-C **for** loop generators can produce scalar values, array-components and windows within arrays. The array component generator can generate elements, vectors, planes or slices from an array, and send these into each iteration of the loop. The window generator can generate an arbitrary sub-array by shifting a window over the source array. These generators can be combined into compound generators using the **dot** and **cross** keywords. Each generator variable is given a value using the single-assignment principle. The scope rules for a variable created by a loop generator are similar to the rules for other variables [3]. There is a restriction that a loop generator target variable may not be referenced elsewhere in that loop's generator. Loops can return an expression or they can be used to reduce an array or structure an array. A loop can return the **final** value of a nextified variable [3].

2.1.5 User-Defined Functions

A SA-C program consists of one or more function definitions. The function named "main" is considered the top-level function. All function names are in scope throughout the entire program. SA-C functions cannot be recursive, as a SA-C program is to be laid out on an FPGA [3]. SA-C functions pass their arguments by value and nextified assignments cannot occur across function parameters.

2.2 SA-C Arrays

The strength of SA-C is the ease with which it allows a programmer to create, manipulate and reduce arrays. SA-C has multidimensional arrays with components limited to scalars and complex numbers, i.e. arrays of arrays are not allowed in SA-C [3]. All arrays are rectangular and are not ragged [3]. They are homogeneous and the elements are stored at equal distances in memory, so that array element addresses can be calculated using simple linear expressions [3]. The following terms, defined in the SA-C Language Version 1.0, are used to describe arrays.

1. **Rank:** The number of dimensions in an array

2. **Extents:** There is one extent for each rank, it represents the length in that dimension
3. **Size:** Multiplying all the extents together yields the number of elements or size
4. **Shape:** An array's rank and extents taken together are its shape
5. **Component Type:** The SA-C scalar data type of all array elements
6. **Type:** Array type matching is done based on component type and rank, *not* extents or size.

During program execution, an array carries its extents with it and these extents are accessed either explicitly through the **extents** operator or implicitly through a loop generator. Arrays of different sizes but equal rank and component type are considered the same type, so passing arrays via functions is simplified. SA-C arrays have a **row-major** storage sequence order which varies the right-most indices the fastest [3]. Note that this implies a data access order for loop generators. This will be explored in more detail in the next chapter. When a SA-C program declares and defines an array, it will provide a type specifier for the component type and a sequence of static or dynamic (:) extents. Arrays input to the **main** function typically have dynamically defined extents, and these are passed to any copies or slices taken during the program. A drawback of using dynamically defined array extents is that the compiler cannot statically know if an array is empty. An *empty array* is defined as a size 0 array, where by the definition just given at least one extent is 0.

2.2.1 Array Operations

Arrays can either be created with an explicit definition, through the return value of a loop or by using a structure-building array operation such as **array_concat()** or **array_conperim()** [3]. **array_concat()** will concatenate two arrays of equal type and of equal extents in all but the right-most direction together [3]. **array_conperim()** will surround an existing array with a perimeter of a certain width and value [3]. Arrays can also be reduced to single scalars or small arrays of scalars using an array reduction operator. The reduction may be performed monolithically using an **array_** operator or piece-by-piece by combining a loop reduction with a specific loop generator. The array reduction/loop reduction operators allow the SA-C programmer to perform the reduction

on user-defined *regions* of the array or to reduce using a *boolean mask*. User-defined regions are defined using a label array whose shape is identical to the shape of the array being reduced [3]. The boolean mask is the same shape as the source array and determines which elements will be used in the reduction. Typical reductions including summing all the values into one value, returning the median value, multiplying all the values together, etc.

2.2.2 Array-Component Generators

An array component generator can generate column or row slices from an array. It can stride through an array (e.g. odd or even rows/columns). It can generate individual elements from an array. A `~` is used in the access pattern to denote that individual elements from this rank must be taken and a `:` indicates that a whole slice in this rank must be taken. The default access pattern takes all individual elements from the array (i.e. **for a in A**). To access all rows in A use **for a (~,:) in A**, which creates a loop in which *a* has the value of each row from A, one row per iteration. To access its individual elements **for v in a** can be used.

2.2.3 Structure-building Operations

There are three kinds of structure-building operators that can be used to create arrays for loop return values, and two kinds of structure-building operators for creating arrays monolithically. They are summarized in Table 2.1 [3]. The first three operators must be paired with a loop generator to generate multiple instances of their component parameter *a*.

Operator	Description
array(a)	Build an array out of the specified components, which may themselves be arrays, creating a result array whose rank is the sum of the ranks of the generator and the component
concat(a)	Allows the component arrays to be concatenated to have different sizes in the last rank (similar to array_concat())
tile(a)	Tiles the component arrays together in each dimension of the loop. All component arrays must be the same shape
array_concat(A,B)	Concatenates arrays A and B together in the last rank
array_conperim(A,w,v)	Surrounds an existing array A with a perimeter of a certain width and value

Table 2.1: Structure-Building Operations

2.3 DDCF Graphs

Data-Dependence-Control-Flow (DDCF) graphs are used as an intermediate representation in the SA-C compiler [2]. The graphs are acyclic and hierarchical (i.e. some nodes contain subgraphs within them) [2]. The entire SA-C language can be represented by DDCF graphs [2]. There are two kinds of DDCF nodes, simple and compound nodes. Compound nodes contain subgraphs and simple nodes do not. Nodes have input and output ports that interface the node to the rest of the graph by means of edges [2]. The document 'The SA-C Compiler Data-Dependence-Control-Flow (DDCF)' by Hammes and Böhm (1999) is an excellent further reference [2]. All functions in a SA-C program have associated DDCF graphs. There are three top-level loop nodes, `ND_FORALL`, `ND_FORNXT` and `ND_WHILE`. Loops use loop-generator graphs to send data through iterations and loop-return nodes to return data. Loops can either be structure-building loops, that is they build up an array from smaller pieces, or loop reductions where the input is reduced to a simpler, probably scalar form. Structure-building loops use the `ND_CONSTRUCT_ARRAY`, `ND_CONSTRUCT_CONCAT` and `ND_CONSTRUCT_TILE` nodes for loop-return and their top-level generator graphs will generally be window or slice generators (`ND_WINDOW_GEN` or `ND_SLICE_GEN`). Loop reductions also often use `ND_WINDOW_GEN` or `ND_SLICE_GEN` in their generator graphs. SA-C allows loops to exist with loop-carried dependencies, such loops are denoted by the `ND_FORNXT` and `ND_WHILE` nodes, and use two simple nodes to represent each "nextified" variable. `ND_NEXT` and `ND_G_INPUT_NEXT`. These loops will be examined in more detail later.

2.3.1 Nodes that create arrays

Table 2.2 shows the DDCF nodes that could potentially create an array as their output [2]. It should be noted that a node which does not consume its array input (according to the definition in the next sub-section) would therefore produce it as output. Since this does not entail creating a new array, no additional memory is required. Therefore this table is not a complete reference of all DDCF nodes which could produce an array as output, but rather all DDCF nodes that *may* require new memory space for that array output.

Node Name	Description
ND_ARRAYREF	This node can extract either scalars or slices. Its input is a pattern of extraction, for example $(-, :, :, -)$.
ND_ARR_DEF	This node creates a constant array from a collection of scalars.
ND_ARR_CONPERIM	This node creates an array with a constant-value perimeter of a certain width.
ND_ARR_CONCAT	This node concatenates two arrays.
ND_CONSTRUCT_ARRAY	This node accumulates pieces into an array return.
ND_CONSTRUCT_CONCAT	This node concatenates pieces into a concat return.
ND_CONSTRUCT_TILE	This node tiles pieces into a tile return.
ND_WINDOW_GEN	This node is an array window generator node used in a loop generator graph. The primary output is the array window that is generated from the first input which is the source array.
ND_SLICE_GEN	This node is an array component generator node used in a loop generator graph. The first input is the source array and the first output is the generated slice.
ND_REDUCE_MODE	This node is a loop reduction node which outputs a rank-one integer array of the most frequently occurring values in the source array (the first input).
ND_REDUCE_VALS_AT_MINS	This node is a loop reduction node which outputs a rank-two array of the minimal array elements and their indices.
ND_REDUCE_VALS_AT_MAXS	This node is a loop reduction node which outputs a rank-two array of the maximal array elements and their indices.
ND_REDUCE_HIST	This node is a loop reduction node which outputs a rank-one integer array which is the histogram of the source array (first input).
ND_ACCUM_HIST	This node is the same as a ND_REDUCE_HIST node except it also accepts a label array and accum range as input.

Table 2.2: Array-Creation Nodes

2.3.2 Nodes that consume arrays

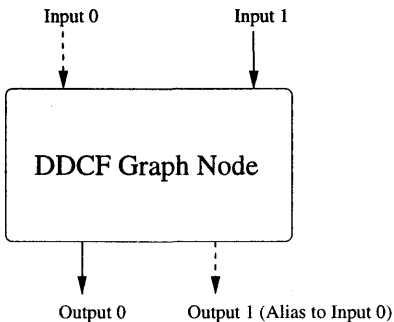


Figure 2.1: Example of DDCF Sink Node

Consider an arbitrary DDCF graph node G , having x array inputs and y array outputs. Any input which is *not* aliased by an output is **consumed** by the node G . This is the strictest definition of consuming data, since the node G now acts as a **sink node** for any non-aliased inputs. See Figure 2.1 for an example of a DDCF node which sinks input 1 but not input 0. For SA-C it is not feasible to treat every DDCF graph node as a potential sink node. Instead a subset of graph nodes has been selected such that all top-level program statement and control flow nodes will act as consumers. Not every statement block will act as a consumer, since the statement blocks associated with conditional expressions will not (e.g. ND_CASE). Conditional expressions as a whole will consume however (e.g. ND_SWITCH). Table 2.3 shows the DDCF nodes that have been selected to act as array consumers. It is a special case when a node's input is aliased by an output and will be examined in the section on Reference Counting.

Node Group	Description
Array References	ND_ARRAYREF
Conditional Expressions	ND_SWITCH
Loop Nodes	ND_WHILE, ND_FORNXT, ND_FORALL
Function Calls	ND_FCALL
Array Component Generators	ND_SLICE_GEN, ND_WINDOW_GEN, ND_ELE_GEN
Array Reductions	ND_REDUCE_ type nodes and label array reduction nodes
Structure-Building Operations	See Table 2.1
Single-Assignment Array Creation	ND_ARR_DEF
Other SA-C Statements	ND_PRINT, ND_ASSERT
Miscellaneous	ND_CAST, ND_OFFSETS, ND_LOOP_INDICES, ND_FEED_NEXT, ND_EXTENTS

Table 2.3: Array Consumer Nodes

2.4 Host Code Generation

Host code is generated in a straightforward manner from the DDCF graph. Array payloads are allocated on the program heap and references to them are kept in host code array structures. An example structure is the `ArrayUint8` struct, shown below.

```
typedef struct {
    int rank;
    int size;
    int extents[8];
    int mults[8];
    Uint8 *base;
    Uint8 *mem;
} ArrayUint8;
```

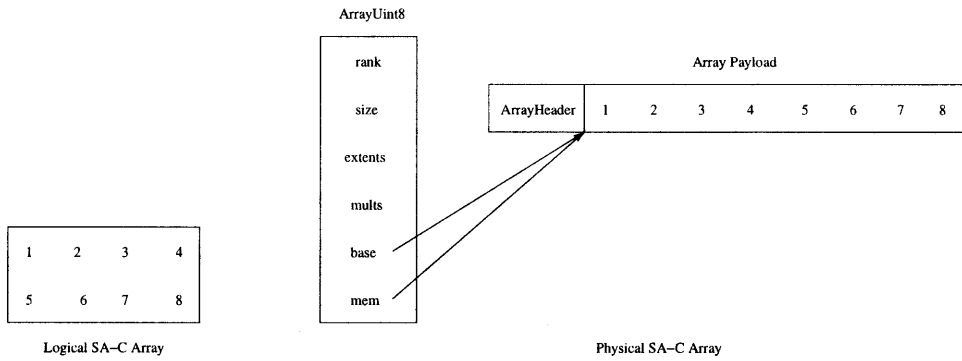


Figure 2.2: Example of Logical and Physical SA-C array.

In the example above, **Uint8** indicates the data type of each payload element. Additional fields include rank, size, extents, multipliers, and two pointer fields, **mem** and **base**. **mem** points to the area in heap where the payload has been allocated and **base** contains the address of the first array element. All the extents and all the multipliers taken along with the base address comprise a **dope vector** (described in more detail below). A logical SA-C array is the array as viewed by the SA-C programmer and the physical SA-C array consists of one or more **references** (the host code array structure previously described) and the **payload** (the array data plus header). See Figure 2.2 for an example. The array payload includes an **ArrayHeader** structure. The **ArrayHeader** is allocated before the payload and can be accessed by subtracting the `sizeof (ArrayHeader)` from the **mem** pointer. It contains a reference count and an `is_nextified` flag. Since the header is associated with the payload and not the `ArrayUint8` struct, the reference count counts how many `ArrayUint8` structs

will be pointing their **mem** pointers to this payload (i.e. how many consumers there will be for this array data line). The closest equivalent to the array payload is the data flow line in a SA-C DDCF graph which corresponds to unique array data.

2.4.1 Aliases

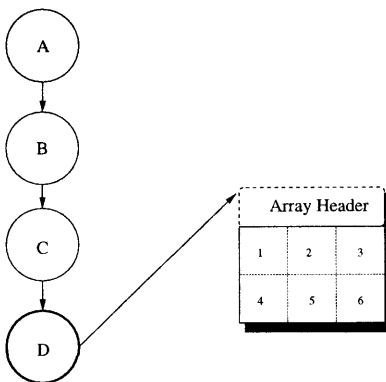


Figure 2.3: Logical References forming an Alias Chain

A **reference** is the host code array structure which references the array payload. An **alias** is a reference which references a payload which already exists. It occurs when a trivial assignment of the form *source* = *dest* is generated such that *dest* has been previously singly-assigned earlier in the SA-C program. **Logical references** are illustrated in Figure 2.3 and form an **alias chain** in which the first reference is termed the **creator reference** and all other references in the chain are aliases. In the figure, the reference 'D' is the creator reference and references A, B and C form an alias chain to D. The chapter on reference counting will show that these alias chains are non-circular due to strict single assignment. Logical references are useful for examining various properties of single assignment as they apply to the SA-C language. A **physical reference** is the host code array structure itself and results from the way in which the host code implements reference semantics using pointer semantics. Figure 2.4 illustrates a set of physical references. Note that one physical reference has more than one actual pointer. All the dependence figures presented in the next chapter can be equivalently redrawn using physical references. Physical references are implementation specific.

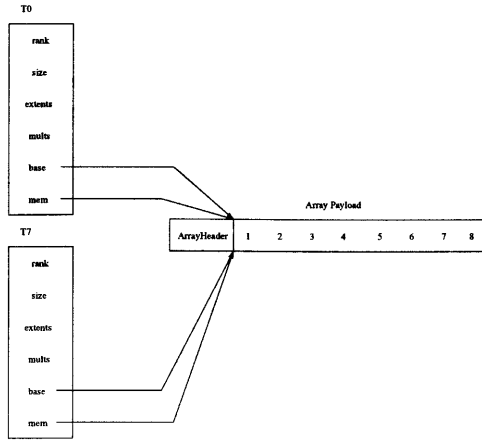


Figure 2.4: Host Code Aliases

2.4.1.1 Conditional Expressions

When graph outputs are generated after a conditional expression has terminated, it is done through aliases. Consider Figure 2.4, which illustrates the situation when the default case is taken for the following SA-C code.

```
uint8[:,:] main (uint8 A[:,:], uint8 n)
{
    uint8 Res[:,:] =
    switch (n) {
        case 1: { uint8 R[:,:] = for e in A {
                    uint8 E[2,2] = { {e, e}, {e, e} };
                } return (tile (E));
                } return (R)
        default: return (A)
    };
} return (Res);
```

The array A is passed through unchanged, and so the code generator transfers A through the ND_SWITCH output as an alias. The following host code snippet shows this, where T1 is the switch key n, T0 is the input array A and T7 is the output from the ND_SWITCH:

```
{ /* switch block */
switch (T1) {
case 1 :
{ /* case block */
{
    /* Code to create R by tiling 2x2 elements from A omitted.
    T5 builds up each piece, T6 is equivalent to R and contains all
    the T5's tiled together. */
}}
T7 = T6; /* T7 is an alias to T6 */
} /* case block */
```

```

break;
default :
{ /* case block */
T7 = T0; /* T7 is an alias to T0 */
} /* case block */
break;
}
} /* switch block */

```

2.4.1.2 Dope Vectors

Traditionally a **dope vector** is a structure containing information about arrays that a program allocates dynamically during program execution. Dope vectors are required when the bounds of an array are not known at compile time, which is the case with SA-C. Additionally, it is easy to create substructures such as array slices and windows by taking one dope vector out of another. SA-C dope vectors are implicitly contained in the host code array struct. A SA-C dope vector constitutes a single **base address** and a **multiplier** and **extent** pair for each rank of the array. The following fields of the host code array struct correspond to a SA-C dope vector:

```

int rank;
int extents[8];
int mults[8];
Uint8 *base;

```

The base addresses' data type of `Uint8` can be any SA-C data type. The arrays `extents[]` and `mults[]` are *parallel arrays* and both use the same value (`rank - 1`) for their index upperbound. The following symbols are used to notate the components of a dope vector D :

$n = \text{rank} - 1$; extents = $e_0..e_n$; multipliers = $m_0..m_n$ and base address = α .

2.4.1.3 Colon Notation

Dope vectors are used to efficiently implement **colon notation**. Array slices can be taken using colon notation, similar to that of Fortran 90 [3]. A lone colon in a given dimension signifies taking the entire index range of the dimension ($0..e_n - 1$, where $n = \text{dimension}$) [3]. A subrange can be specified with integer expressions on either or both sides of the colon, and an optional step as a third parameter [3]. In this paper, subranges will be specified as `lwb:upb:step` where each is an integer expression in the dynamic range determined by the bounds checker.

The address of array element $A[i_0, \dots, i_n]$ is given by Equation 2.1:

$$\&A[i_0, \dots, i_n] = \alpha + \sum_{q=0}^n (pm_q * i_q) \quad (2.1)$$

where α = parent base address, $n = rank - 1$ and $pm = m_0, \dots, m_n$ which comprise the parent's multipliers.

The largest structure which is not a substructure of any other structure is equivalent to the creator reference for this array and has a parent's dope vector which is the same as its own. It will be called the **first dope vector**. Since arrays are stored in row-major order, which means varying the right-most indices the fastest, the multipliers for the first dope vector are taken by leaving the right-most multiplier as 1 and computing the remaining multipliers according to the recurrence relation described in Equation 2.2:

$$\sum_{q=n-1}^0 m_q = (m_{q+1} * e_{q+1}) \quad (2.2)$$

Note that $m_n = 1$ and $n = rank - 1$.

Since SA-C does all bounds checking dynamically using the array's extents, this means i_0 must be in the range $0..e_0 - 1$ and in general each i_n must be in the range $0..e_n - 1$. Since only substructures and structures can be taken using colon notation, n is no greater than the rank of the first dope vector, but may be lesser, resulting in a lower ranked slice being taken.

2.4.1.4 Array Reference Example

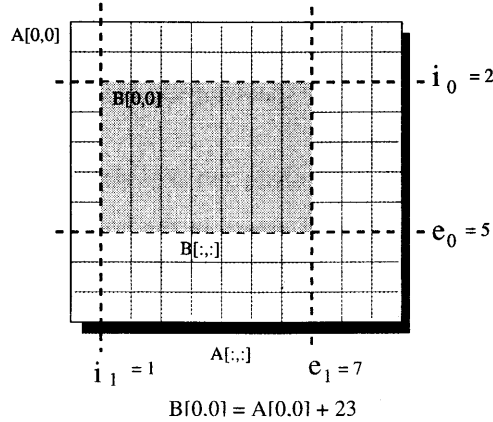


Figure 2.5: Dope Vector

Assume A is a 10x11 rank 2 matrix of Uint8s. The following SA-C code returns B as a substructure of A:

```
uint8[:,:] main (uint8 A[:,:]) {
    uint8 B[:,:] = A[2:6, 1:7];
} return (B);
```

The dope vector D for matrix A is given as follows:

$$D = \{ \{ \text{rank} = 2 \}, \{ \text{extents} = \{ 10, 11 \} \}, \{ \text{mults} = \{ 11, 1 \} \}, \{ \text{base} = 0xa \} \}$$

The shaded part of Figure 2.5 illustrates the new dope vector D' taken from D as matrix B.

$$D' = \{ \{ \text{rank} = 2 \}, \{ \text{extents} = \{ 5, 7 \} \}, \{ \text{mults} = \{ 11, 1 \} \}, \{ \text{base} = 0xa + 23 \} \}$$

First the new base address is computed using $B[0,0] = A[2,1]$:

$$\&B[0,0] = \&A[2,1] = 0xa + 11 * 2 + 1 * 1 = 0xa + 23$$

For each rank r there is a new extent e_r whose value is given by using the values specified in the subrange notation for this slice. Thus $e_r = 1 + ((upb_r) - (lwb_r)) / (step_r)$ or 0 whichever is greater.

$$e_0 = 1 + ((6) - (2)) / (1) = 5; e_1 = 1 + ((7) - (1)) / (1) = 7$$

The new multipliers are taken as the parent's multipliers in each rank: $m_r = pm_r$. The new size is the product of all the extents. To get the address of $B[1,0]$, take the address of $B[0,0] + 11$ (as per Equation 2.1 above). Substructures can also be taken from D' or from D and new substructures

will *never* need more than a new dope vector. No heap memory is required for any dope vector except the first. For this reason, subsequent dope vectors are considered to form an alias chain as described previously.

2.4.2 Host Code Arrays

There will be the following kinds of arrays in the host code.

1. main and function input arrays
2. local arrays
3. intermediate structures

For each main input there will be one distinct payload. Inputs to non-main functions will refer to previously allocated payloads. Local arrays consist of the following types of arrays:

1. Arrays assigned via single assignment including function return values and loop generator target array variables (i.e. used to take windows or slices)
2. Arrays assigned via nextified single assignment denoting a variable with loop-carried dependencies

There is one type of intermediate structure used by the host code. The "array-of-arrays" is used to accumulate sub-arrays, which are later arranged into a result array as the return value of a structure-building loop. Additionally, the "growable" property can be applied to any array data. The data starts in an array of a fixed capacity and grows by doubling its capacity when required. Growable arrays (g_arrays) are useful for storing truly variable amounts of information, like the mode of an array.

2.4.2.1 Dynamic Array Properties

SA-C host code arrays have dynamic characteristics which can be associated with either their array payload or with the reference to the payload (i.e. the host code array structure). If a characteristic is per payload, then multiple references share that characteristic. If the characteristic is per reference

then multiple references have distinct values for that characteristic. The following characteristics are determined dynamically.

1. `is_nextified` - This characteristic is per payload
2. `reference count` - This characteristic is per payload
3. `is_dopevector` - This characteristic is per reference
4. `is_empty` - This characteristic is by necessity per reference
5. `is_param` - This characteristic is per reference

It is unfortunate from an implementation standpoint that emptiness is determined dynamically, because neither pointer reuse or reference counting should be applied to empty references. This complicates host code somewhat by requiring explicit size checks before dereferencing arbitrary array references.

Chapter 3

Statically Determinable Garbage

Single-assignment creates references at the host code level. These references can either be aliases (either resulting from conditional expressions or dope vectors) or creator references (§1.4.1). Loop generators produce data for each loop iteration. A creator reference allocates and points to **heap data**. The process of creating alias references inside a loop is called **single-assignment aliasing** and generates no **heap data flow**. The process of creating creator references inside a loop is called **single-assignment creation** and results in heap data flow.

3.1 Overview

Every intermediate structure-building operation uses the array-of-arrays technique described below to accumulate its respective components (whether aliases or data) into an array and then restructure the array according to the desired structure-building operation after the loop has terminated. Specifically, the ND_CONSTRUCT_ARRAY, ND_CONSTRUCT_CONCAT and ND_CONSTRUCT_TILE DDCF nodes classify as structure-building. The loop generator graphs for these operations usually declare and define component pieces used to build up the structure. These pieces are usually alias references to the source array, since they are typically slices or windows. Specifically, the ND_WINDOW_GEN, ND_SLICE_GEN, and ND_ARRAYREF DDCF nodes typically generate alias references instead of creator references.

Some structure-building operations use one of the proper array creation nodes listed in the previous chapter, like ND_ARRAY_DEF. These nodes generate a malloc() statement and thereby generate **creator references**. If a creator reference occurs inside a loop which iterates N times, the

single-assignment property results in $N - 1$ pieces of garbage being created. **Pointer reuse** will be presented to eliminate this garbage. Nextified single assignments inside loop statement blocks can also create garbage and have considerably more complicated inter-dependencies possible.

3.2 Array of Arrays

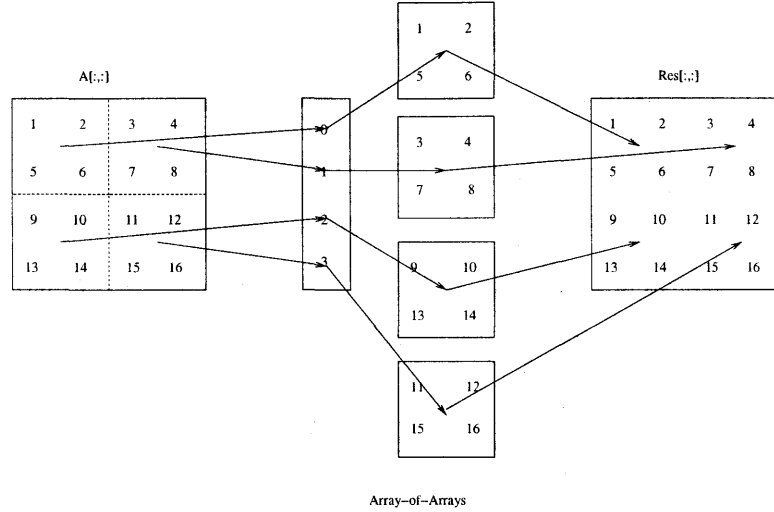


Figure 3.1: Array-of-arrays

Consider SA-C code which tears an $N \times N$ matrix into smaller windows of size $w \times w$, and then tiles them back into the same $N \times N$ structure (Figure 3.1). This code will use an `ND_CONSTRUCT_TILE` node coupled with an `ND_WINDOW_GEN` node to show two things. The first is how the intermediate array-of-arrays structure works. The second is a loop which creates aliases instead of heap data (single-assignment aliasing).

```
uint8[:, :] main (uint8 A[:, :]) {
    uint8 Res[:, :] = for window W[2,2] in A step (2, 2) return (tile (W));
} return (Res);
```

In each loop iteration the `ND_WINDOW_GEN` node generates a dope vector alias (a substructure) of the source array. It does this in a single-assignment fashion, such that the previous dope vector alias is lost and overwritten with the new one. This does not create garbage because the left-hand side of the assignment will either never have existed or existed only through single-assignment. As each window is generated by each loop iteration, a *copy* of it is accumulated or collected into

an array-of-arrays. In general, any data which must be accumulated in order to create a larger piece of new data will have a *copy* of it placed in an array-of-arrays. Every element of the array-of-arrays is heap data and will not be an alias to anything. When the structure building operation of **tile(W)** is performed, the payload block of the result array **Res** is allocated and the pieces in the array-of-arrays are copied into **Res** in the appropriate locations. Now there is no sharing and after **Res** is constructed, the array-of-arrays can be de-allocated (piece-by-piece, then the whole structure). This entire process requires $3*n$ space, where n is the total number of elements in the source array **A**. Knowing that this operation always works with copies of the respective pieces allows static de-allocation of the array-of-arrays structure immediately after it is consumed, which can lead to a linear increase in space performance or a polynomial increase if there is a loop around the tiling process.

3.3 Basic Single Assignment

Consider code which contains a structure-building loop that creates a new array for each iteration and uses it to tile together its result. This example will illustrate a loop which creates data and hence garbage (single-assignment creation).

```
uint8[:,:] main (uint8 A[:,:]) {
    uint8 Res[:,:] = for e in A {
        uint8 E[2,2] = { {e, e}, {e, e} };
    } return (tile (E));
} return (Res);
```

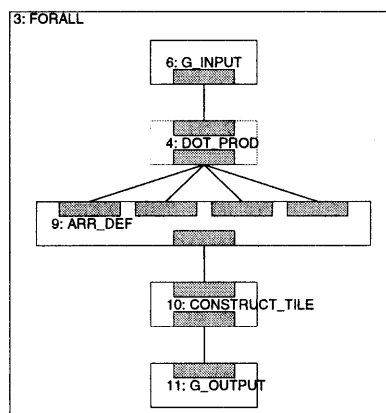


Figure 3.2: Loop with Array Definition

The DDCF graph for this SA-C code contains an ND_ARR_DEF node inside an ND_FORALL loop node using an ND_ELE_GEN to generate scalars (Figure 3.2). The following kind of host code is generated.

```
// Create and initialize an empty array-of-arrays
// Loop through all elements in A
for ( all rows in A )
for ( all cols in A ) {
    // Take e out of A as A[row][col]
    // Malloc space for array E in pointer T0.base
    SC_malloc (T0.base)
    // Fill E with e
    // Put a copy of E into the array-of-arrays
}
// Restructure the array-of-arrays into Res
// Free each piece (E) in the array-of-arrays
// Free the array-of-arrays
```

T0 is the array E[2,2]. The code generator has generated a malloc inside a loop for the creator reference T0. This means after the first iteration, the pointer T0.base will start leaking heap (See Figure 3.3). There is a "better" way to re-use the pointer T0.base than just overwriting its old address with a new one. This loop produces and wastes a new 2 x 2 array for every single element of the N x N source matrix, creating $O(N^2)$ garbage. Assuming the size of the garbage to be constant, such loops produce garbage on an order of magnitude directly proportional to their loop generator's order of magnitude. Referring to Figure 3.3 it is clear that the array data produced in iterations 0, 1 and 2 is garbage after the loop statement block finishes.

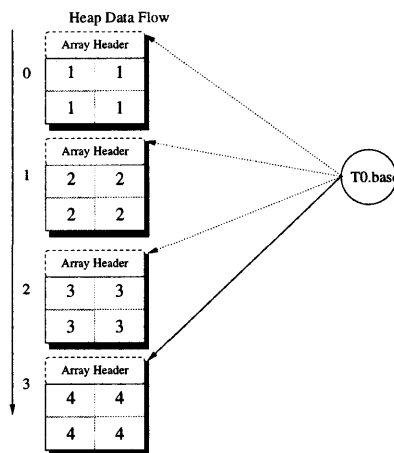


Figure 3.3: Basic Single Assignment Data Flow

Pointer reuse entails freeing the pointer p at the statement before the `malloc`. In other words, the creator reference has the property that it eliminates what it created in the previous loop iteration. For this reason, pointer reuse only works for basic single assignment and must be disabled for nextified single assignment. By virtue of its implementation (associating it only with statements that `malloc`), pointer reuse only occurs for the last reference in an alias chain (it will be shown that alias chains are non-circular).

3.4 Creator Reference Scheme

Consider the situation when a new reference r is to be attached to an alias chain as a result of single-assignment. The **creator reference scheme** states that any reference r attached to an arbitrary alias chain will eventually connect to a creator reference R which references actual data rather than another reference. Additionally, all aliases in the chain below r are logically older than r and all aliases above r are guaranteed to be both newer and non-creator references. The chain is such that as long as all members were attached using Basic Single Assignment, no reference except the creator reference has the chance of creating garbage. It will be shown in the chapter on reference counting that circular aliases are not allowed.

3.5 Nextified Single Assignment

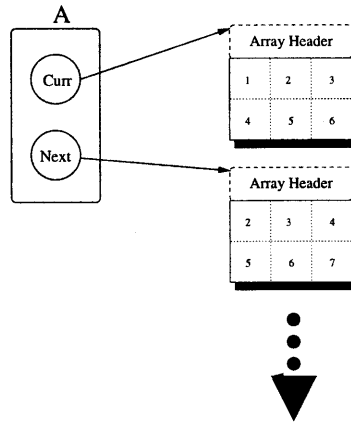


Figure 3.4: Nextified Variable Inside of Loop

A SA-C variable can be composed of more than one reference and hence more than one value. This occurs when there is an implicit back-edge in the DDCF graph (i.e. loop-carried dependencies) and is produced when the SA-C programmer flags a variable as *nextified*. Figure 3.4 illustrates a logical nextified variable **A** during a loop and Figure 3.5 illustrates the same variable outside its loop. It has two references, a current reference and a next reference, both of which may be generated as either an alias reference or a creator reference. Pointer reuse only worked with creator references and was outside the scope of alias references, but with nextified assignments the *type* of reference generated becomes very important. To logically distinguish the references in **A**, they will be written as **A.curr** or **A.next**. They are created as two distinct host code array structures in the generated host code and have their own unique identifiers. Figure 3.5 shows that once outside its dependent loop, the variable again has one logical reference to data.

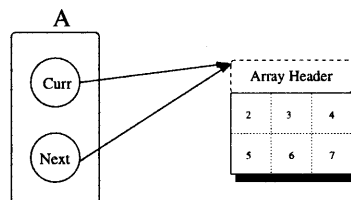


Figure 3.5: Nextified Variable Outside of Loop

Nextified assignments *must* adhere to these restrictions:

1. Nextified assignments always occur in loops (ND_WHILE, ND_FORNXT)
2. Nextified assignments represent a back-edge in the DDCF graph
3. Nextified assignments can occur at most *once* to a particular variable within the body of a loop. Shadowing of nextified variables is forbidden.
4. There is always an initial assignment to the nextified variable prior to entering the loop

Futhermore, SA-C language semantics dictate that:

1. Statement blocks are side-effect free in nature, therefore a copy of a nextified variable's initial value is used during the loop and is generated through a ND_FEED_NEXT DDCF node
2. Some of the data generated for each loop iteration is generated by a loop generator, which accesses a source array in row-major order. The curr reference follows behind the data flow of the next reference and these semantics imply curr and next will have a row-major order in relation to each other if they access any data generated by a row-major loop generator.

A SA-C programmer is only able to indicate a preferred value for one half of a nextified variable, the next reference. The current reference **follows** the next and cannot be single assigned to.

There are four kinds of nextified single assignment which result from the ability to assign only to the next reference of a nextified variable. They are summarized below and examined in detail in the sections that follow.

1. Basic Nextified Single Assignment
2. Full Nextified Alias
3. Partial Nextified Alias
4. Nextified Single-Assignment Aliasing

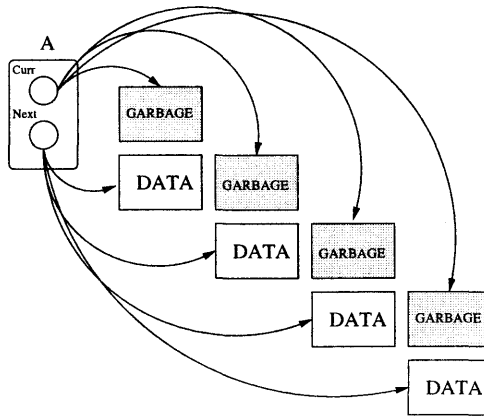


Figure 3.6: Basic Nextified Single Assignment

3.6 Basic Nextified Single Assignment

Usually "nextified variables" indicate that the variable requires its value from the last loop iteration in order to compute its "next" value (e.g. `next A = f(A);`). This simple dependence is illustrated in Figure 3.6 and results from basic nextified single assignment. A loop with a single loop-carried dependency described by nextified array A is generated as follows:

1. Initialization section creates a copy of A and the A.curr and A.next references reference this copy
2. Loop predicate either enters the loop body or skips the whole loop
3. Loop body contains the single nextified assignment to A.next and many potential uses of A.curr and A.next
4. Transfer occurs at the end of each iteration and moves A.curr to A.next
5. Goto to Loop Predicate

All transfer and initialization of $A.curr$ and $A.next$ occur through trivial assignment statements of the form $source = dest$. The last thing the loop does before jumping to the predicate is transfer the current reference portion of any nextified variable to its next reference (step 4 above). It does this in two steps, first it holds the next references in temporaries, then it moves the temps on top

of the current references. This technique avoids analysis of the possible dependencies between two nextified SA-C variables. For example, assume T1 is A.curr and T2 is A.next and A is a nextified array of Uint8s.

```
{
ArrayUint8 Tmp1;
Tmp1 = T2; // Hold A.next in Tmp1
// A.curr is now garbage!
T1 = Tmp1; // Leak A.curr and make A.curr reference A.next
}
```

When A.curr is transferred to A.next at the end of the loop's iteration, A.curr becomes garbage in the form of a heap leak. The host code should free A.curr immediately before assigning it to A.next:

```
{
ArrayUint8 Tmp1;
Tmp1 = T2; // Hold A.next in Tmp1
SC_free (T1) // A.curr is now garbage, so eliminate it
T1 = Tmp1; // Make A.curr reference A.next
}
```

The chain of aliases the current reference will follow to reach a creator reference is exactly the same as the chain followed by the next regardless of how long that chain is. The garbage indicated in Figure 3.6 can be statically determined and eliminated.

3.7 Full Nextified Alias

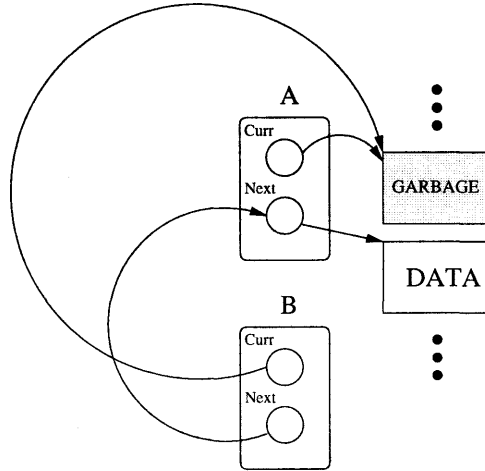


Figure 3.7: Full Nextified Alias

A full nextified alias results when the SA-C programmer assigns one nextified variable B to another nextified variable A through trivial assignment. Consider the following SA-C code in which B is a full nextified alias to A.

```
uint8[:,:] main (uint8 n, uint8 A[3,2], uint8 B[3,2]) {
    uint8 R[:,:] =
        for uint8 i in [n] {
            next A =
                for a in A dot b in B
                    return (array ((a+b)/2));
            next B = A;
        } return (final (A));
} return (R);
```

Figure 3.7 shows that garbage can still be statically determined and the only wrinkle is that nextified variable B shares the same garbage as A. This sharing can be statically determined and *only* A will attempt to clear up its garbage.

3.8 Partial Nextified Alias

The **next** keyword is position sensitive. If a nextified variable B is declared as an alias to nextified variable A, it will either be a full or partial nextified alias depending on whether the assignment of next B occurs after the assignment of next A (full nextified alias) or before the assignment of next A (partial nextified alias). Therefore a partial nextified alias results when the SA-C programmer

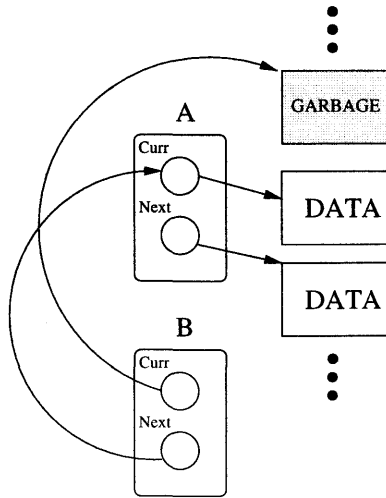


Figure 3.8: Partial Nextified Alias

assigns one nextified variable B to the *current* value of another nextified alias A (A.curr). This causes variable B to lag one iteration behind A (which already lags one iteration behind). Both of the SA-C codes below result in B being defined as a partial nextified alias to A.

```

uint8[:,:] main (uint8 n, uint8 A[3,2], uint8 B[3,2]) {
    uint8 R[:,:] =
        for uint8 i in [n] {
            uint8 BB[:,:] = A;
            next A =
                for a in A dot b in B
                    return (array ((a+b)/2));
            next B = BB;
        } return (final (A));
} return (R);

uint8[:,:] main (uint8 n, uint8 A[3,2], uint8 B[3,2]) {
    uint8 R[:,:] =
        for uint8 i in [n] {
            next B = A;
            next A =
                for a in A dot b in B
                    return (array ((a+b)/2));
        } return (final (A));
} return (R);

```

Figure 3.8 shows that garbage can still be statically determined. Note that nextified variable A produces *no* garbage because its garbage is consumed as B's next.

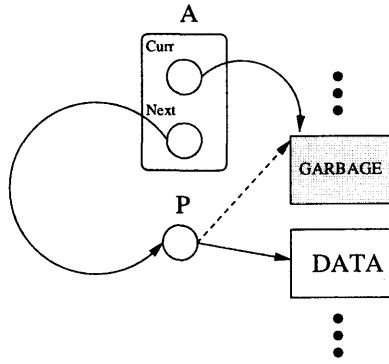


Figure 3.9: Nextified Single Assignment Aliasing

3.9 Nextified Single Assignment Aliasing

Nextified single assignment aliasing occurs when the SA-C program defines a nextified variable B as an alias to a source array (either by means of a dope vector or a conditional expression). A SA-C program can create a nextified variable which will take its next value from an alias each loop iteration. The alias could reference heap data or other aliases (eventually referencing heap data). Consider the following SA-C code in which the nextified variable A will select one way of producing its next value in iteration 0 and a different way of producing its next value in iteration 1.

```
int32[:] main (int32 A[:], int32 sentinel)
{
    uint8 n = extents (A);
    assert (n%2==0, "n should be even");

    bits1 odd_phase = 0b1;

    int32 res[:] = for _ in [n] {
        next A = ((odd_phase == 0b1) ?
            for window W[2] in A step (2) {
                int32 E[2] = {array_min (W),array_max (W)};
            } return (tile (E)) :
            /* Even-phase */
            for window W[2] in array_conperim (A,1,sentinel) step(2) {
                int32 E[2] = {array_min(W),array_max(W)};
                int32 m[1] = {array_max(W)};
            } return ( tile ( ((array_min(W)==sentinel) ? m : E)) ));
        next odd_phase = odd_phase^0b1;
        print (true, A);
    } return (final(A));
} return (res);
```

$A.next$ is assigned as the result of the conditional expression, which is an alias to the unnamed result of one of the two structure-building operations shown above. Assume the result of either

structure-building operation is referenced by **P**. Figure 3.9 illustrates the resulting dependency. The primary drawback to this scheme is that it contends with pointer reuse, because the nextified variable attaches to an alias chain which terminates in a creator reference. The creator reference, having no knowledge that a nextified variable just attached to its alias chain will attempt to reuse its reference, which interferes with the desire of the nextified variable to eliminate its garbage. Figure 3.9 indicates this problem with a dashed line from the creator reference *P* to its garbage. A solution is to make trivial assignments resulting from conditional expressions smart enough to tell that there is a nextified variable on the left-hand side and thereby prevent pointer reuse through the right-hand side.

3.10 Static Dependency Analysis

The code generator can statically determine what is garbage for each of the four cases given above by examining the DDCF function graph. The id of the current ND_NEXT node's output node is determined by following the backedge of the current ND_NEXT node and using that (node, port) pair to determine the unique id of the output node. We call this unique id *dest_id* throughout. The *src_id* is determined by taking the current ND_NEXT node's *In_next_id* field and using it to index a node in the node list, then taking that node's unique id on output port 0.

```
dest_id = get_source_id (fg, it->val, 0);
tgt_in_node = fg->nodes[it->val].In_next_id;
src_id = fg->nodes[tgt_in_node].outputs[0].unique_id;
```

The code generator will generate a sequence of trivial assignments between nextified arrays of the form *src_id = dest_id*, each trivial assignment will leak through its *src_id* reference, so the desire is to free *src_id* if possible. To determine if an array is a full nextified alias of another nextified array, search through all other ND_NEXT nodes and see that their *dest_id* is not the same as this ones. If two ND_NEXT nodes share the same *dest_id*, the code generator can't free the second *src_id*. To determine if an array is a partial nextified alias of another nextified array, search backwards in the graph, looking for other ND_NEXT nodes which have the same *dest_id* as this particular array's *src_id*. The following algorithm has been incorporated into the code generator while transferring nextified variables:

```

/* Clean up garbage copies of nextified arrays. For each ND_NEXT node, free its src_id if
  a) Its src_id is not a previous ND_NEXT node's dest_id
  b) Its dest_id is not a previous ND_NEXT node's src_id
  c) Its dest_id is not another ND_NEXT node's dest_id
*/
for (i=0, it=fg->nodes[id].My_nodes; it!=NULL; it=it->link)
{
  if ( (fg->nodes[it->val].nodetype == ND_NEXT) &&
        (fg->nodes[it->val].inputs[0].ty.kind == Array) )
  {
    OK_to_free = 1; /* By default assume this ND_NEXT has a garbage array */
    tgt_in_node = fg->nodes[it->val].In_next_id;
    src_id = fg->nodes[tgt_in_node].outputs[0].unique_id;
    dest_id = get_source_id (fg, it->val, 0);
    /* Look through all other ND_NEXT nodes and perform the tests above: O(n^2) */
    for (j = 0, jt=fg->nodes[id].My_nodes; jt!=NULL; jt=jt->link)
    {
      if ( (fg->nodes[jt->val].nodetype == ND_NEXT) &&
            (fg->nodes[jt->val].inputs[0].ty.kind == Array) )
      {
        int other_tgt_in_node, other_dest_id;
        other_tgt_in_node = fg->nodes[jt->val].In_next_id;
        other_src_id = fg->nodes[other_tgt_in_node].outputs[0].unique_id;
        other_dest_id = get_source_id (fg, jt->val, 0);
        /* Case A */
        if ( (other_dest_id == src_id) && (j < i) )
        {
          OK_to_free = 0;
          break;
        }
        /* Case B */
        if ( (other_src_id == dest_id) && (j < i) )
        {
          OK_to_free = 0;
          break;
        }
        /* Case C */
        if ( (other_dest_id == dest_id) && (j != i) )
        {
          OK_to_free = 0;
          break;
        }
      }
      j++;
    }
    if (OK_to_free) fprintf (fp, "  SC_free (T%d.mem)\n", src_id);
  }
  i++;
}

```

3.10.1 Arrays Returned by Functions

An array which is returned by a user-defined function in SA-C is assigned to a left-hand side variable using the same single-assignment property previously examined. Therefore, pointer reuse is also usefully applied to these returned arrays.

Chapter 4

Reference Counting

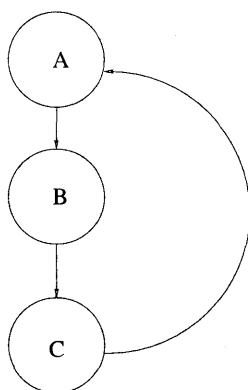


Figure 4.1: Example of Circular Aliasing

Reference counting the SA-C language is possible because aliases can only be created via **strict** single assignment. The **strict** qualifier requires that a variable declaration be accompanied by a complete and immediate definition of its value. The variable cannot be partially defined now and partially defined later. Strict single-assignment entails the following:

1. A variable gets a value *once*
2. A variable gets a value *immediately* upon declaration

These properties ensures that aliases are **non-circular** in nature, since at the time of assignment the creator reference of every alias chain is known. Figure 4.1 illustrates **circular aliasing**. Consider the following SA-C code in which a circular alias similar to the figure is attempted.

```
uint8 [:] main (uint8 C[:])  
{
```

```

uint8 B[:] = C;
uint8 A[:] = B;
C = A; // This statement is illegal
} return (Res);

```

The arrow in Figure 4.1 indicates that A is a reference to B and thus points from A to B. The last SA-C statement is illegal because it is a re-assignment to variable C rather than a single-assignment.

Here is the new corrected code.

```

uint8[:] main (uint8 C[:])
{
    uint8 B[:] = C;
    uint8 A[:] = B;
    uint8 C[:] = A; // This C shadows the input C with an alias
} return (Res);

```

For this new code, the last single-assignment statement implements variable shadowing by simply creating a new fourth variable (the variable consequently holds an alias to A). The input to main is **T3**, the array B is **T2**, the array A is **T1** and the last C is **T0**. The result is **variable shadowing** and is illustrated in Figure 4.2. With variable shadowing, the resulting reference is either a creator reference or an alias reference, but never a circular alias.

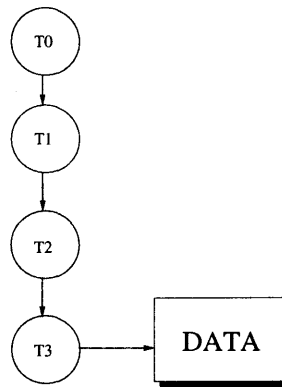


Figure 4.2: Example of Variable Shadowing

Reference counting single-assignment code is fairly straightforward. The trivial assignments above will create a situation in which four host code structures reference the same array payload. The array payload should wind up with a reference count of 4.

Figure 4.3 illustrates simple reference counting. If circular aliasing were possible, this reference counting technique would invariably fail. Whenever an existing alias is assigned to alias a variable

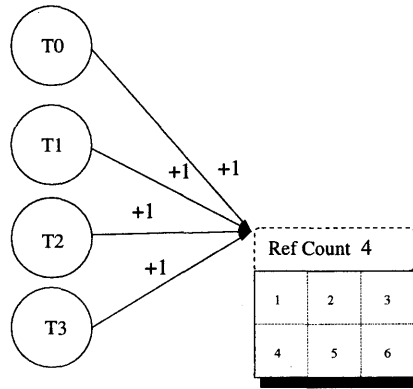


Figure 4.3: Host Code Example of Variable Shadowing

which in turn is an alias to the first alias, circular aliasing occurs. The circular assignment would create a "phantom" increase in the reference count as illustrated in Figure 4.4. Fortunately SA-C does not allow circular assignment to occur by virtue of its strict single-assignment nature, and thus simple reference counting can be employed.

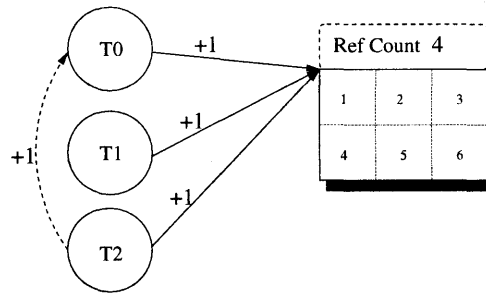


Figure 4.4: Host Code Example of Circular Aliasing

4.1 Overview

DDCF nodes fall into one of *five* categories:

1. Simple nodes, containing no subgraph
2. Compound nodes containing a list of nodes contained in their subgraph, including input nodes, but not output nodes.
3. Input nodes corresponding to input **xxx** of this graph node

4. Output nodes corresponding to output `yyy` of this graph node
5. Nodes with a back-edge (`ND_NEXT`)

A compound nodes subgraph takes the form of a set of input nodes, followed by a set of interior nodes, terminated by a set of output nodes. Edges connect the data between these nodes appropriately. The interior nodes of a `ND_FORALL` or `ND_FORNEXT` compound node are of the form: loop generator subgraph; body subgraph; and loop-return subgraph. All of the various loop-return nodes are simple and have direct associations with the language's return nodes [4]. The loop generator nodes take arrays as inputs and emit array components as outputs, whereas the construction return nodes take components and emit complete arrays as outputs [4].

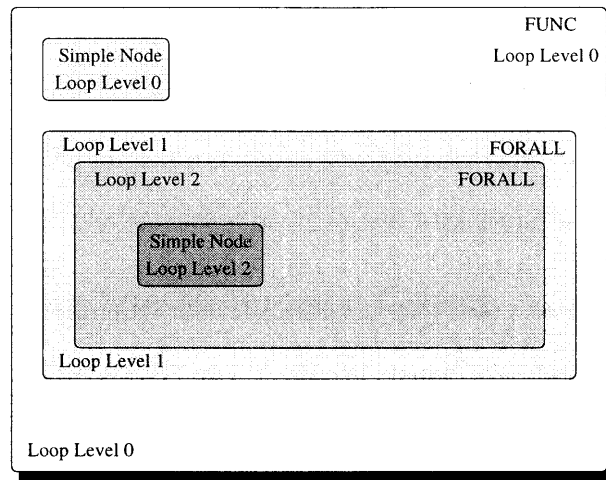


Figure 4.5: Example of Loop Levels

4.2 Loop Levels

In the DDCF graph for a specific SA-C code, it is possible to compute a static **loop level**. The loop level for a DDCF graph node G is defined as 0 if the graph node has no loop control flow associated with it. This is irrespective of any nodes contained in G 's subgraph. A `ND_FORALL` graph node has a loop level of 1 if it is contained inside a `ND_FUNC` graph node. A `ND_FORALL` graph contained inside another `ND_FORALL` graph has a loop level of 2, and so on. As the code generator sequentially traverses the DDCF graph, it will increment the loop level when entering a

loop node and decrement the loop level once the final curly brace (`}`) of the loop node has been generated. Everything contained in a `ND_FUNC` node starts at loop level 0, whereas everything contained inside a loop node is guaranteed to start with a loop level greater than 0 (Figure 4.5). This is useful because the code generator knows if a statement will be generated inside a host code loop or not.

4.3 Basic Approach

This paper’s basic approach to reference counting is given below. The sections which follow will examine each of the items listed below in more detail:

1. Initialize all array reference counts to 1.
2. Increment an array reference count for edge splits occurring at loop level 0.
3. Increment an array reference count exactly **once** for aliases resulting from dope vectors and conditional expressions.
4. Array consumers (as listed in Table 2.3) indicate a desire to decrement the reference counts for each of their array inputs by exactly **one**. This is achieved by adding the unique id for the graph node’s input to a list called the *consumed array* list. Items are added to the list only if not already present, such that a unique id appears in the list only once. This occurs at *any* loop level.
5. Array consumers at loop level 0 process the consumed array list and generate a decrement reference count instruction for each element of the list. The list is then emptied.

Reference counting is **not** performed for inputs to non-main functions. Nextified variables are a special case for the reference counter and will not be added to the consumed array list. The copy generated by `ND_FEED_NEXT` and any references to the nextified variable contained in the dependent loop graph are not reference counted. Once the dependent loop graph has been generated, the **final** value emitted from the loop graph will again be subject to reference counting and if no

final value is emitted, the variable is considered **unfinalized** and its reference count is decremented by one, resulting in it being freed. The section on unfinalized nextified variables details this.

4.4 Initializing the Reference Count

Whenever an array payload is allocated by the host code it will be created such that its associated reference count is set to 1. Creating an array header in front of an array payload is incorporated into the act of allocating space for the array. Initializing the reference count to 1 is incorporated into the act of creating the array header. Whenever an array is passed as an argument to a non-main function, it will be flagged as a **parameter**. Parameters are dynamically not subjected to reference counting by checking the value of the `is_param` flag. This has two beneficial qualities. The first is that aliases to non-main input parameters can be freely created so long as they "inherit" the `is_param` flag value. The second allows mixed language routines to locally create arrays and pass them to SA-C functions without having to also create array headers for those arrays. For example, a C program could statically declare/define an array and pass it to a SA-C function as input.

4.5 Incrementing the Reference Count

There are two situations when the reference count for a host code array needs to be incremented. The first situation occurs only at loop level 0 and the other situation can occur anywhere. In both cases, incrementing the reference count for a particular host code array entails locking a global mutex, going to the arrayheader and incrementing the associated reference count and then unlocking the mutex once finished. The act of incrementing the reference count is therefore performed *atomically* and corresponds to a critical section.

4.5.1 Edge Splits

Using the DDCF graph representation of a SA-C code, data flows through DDCF nodes via edges which connect input and output nodes. It is possible for any compound DDCF node to *split* a given output data to multiple inputs. Each DDCF node which outputs data has an array of $1..n$ output ports. These outputs correspond to distinct data which can be produced by the DDCF node. For

each output data, there is a list of (node, port) pairs to which that data flows. This corresponds to an edge connecting this DDCF node's output to the designated input. Each of the unique values returned by a SA-C statement will have a separate edge in the DDCF graph. If the value of a particular piece of data **A** is required to create the value for each of these edges, then an **edge split** will occur. The edge split entails sending the value of **A** into each of the subgraphs responsible for creating the output values. Consider the following SA-C code in which the function **main** returns multiple values, each of which requires the value of the input variable **A**:

```
uint8, float main (uint8 A[:,:])
    return (array_min (A), array_mean ((float[:,:])A));
```

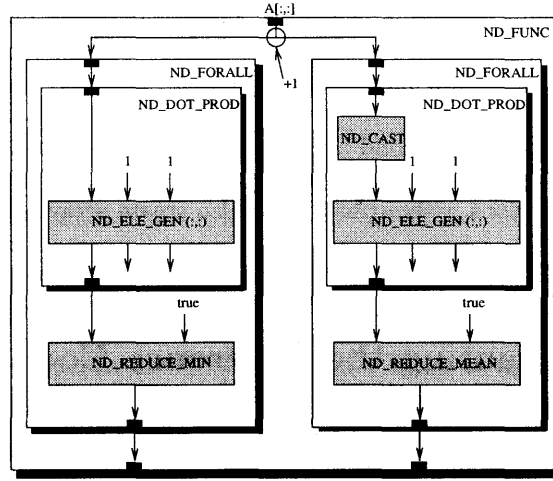


Figure 4.6: Example of DDCF Input Edge Split

Figure 4.6 shows that **A** is sent into both **ND_FORALL** subgraphs in order to produce the two output scalars.

After the code generator generates the code for a given DDCF node, it checks for edge splits by counting the number of targets ((node, port) pairs) for a given output port. If this value is more than 1 and the code generator is generating code at loop level 0, then the reference count for the data corresponding to this output port is incremented by the number of edge splits minus one (provided the source array is not a parameter). In Figure 4.6 the reference count for **A** would be incremented by one after the **ND_G_INPUT** node has been generated (i.e. *before* the two **ND_FORALL** subgraphs).

4.5.2 Aliases

Any time an alias is created whether through a trivial assignment or a dope vector alias, the source array's reference count should be incremented by one (provided the source array is not a parameter). Unfortunately, aliasing can occur inside various kinds of control flow, including loops, which would result in multiple increments to the reference count. The technique of **single-increment** must be employed to emulate the single-assignment behavior of statements which by virtue of being part of a loop body are actually multiple (destructive) assignments. Single-increment consists of incrementing the reference count only *once* instead of every loop iteration. This is achieved in host code using a special flag associated with the reference (the host code array structure). This flag is initially unset and will later be set exactly *once* the first time the reference is singly-assigned to. The assignment which yielded an alias will increment only if the flag was unset.

4.6 Decrementing the Reference Count

The act of decrementing the reference count for a particular host code array is also performed atomically using the *same* global mutex which is used to achieve an atomic increment. The decrement critical section also includes a check to see if the reference count was decremented to 0, and if so, a **free()** statement is issued to release the associated host code array's heap memory. The **mem** and **base** pointers are then set to **NULL**. Note that the increment reference count will get a lock on the global mutex *before* pointing to the host code array's array header, and if it finds a null pointer instead, it will silently not increment anything.

After the code generator finishes generating the code typical for each of the array consumers listed in Table 2.3, it also adds all of that node's array inputs to a **consumed array list**. There is one input for each input port and the **unique ids** associated with them are added to the consumed array list according to the following conditions:

1. The node is an array consumer node
2. The input **unique id** is an array

3. The input `unique id` is not already in the consumed array list
4. The input `unique id` is not a parameter
5. The input `unique id` does not belong to a nextified variable
6. The input `unique id` does not belong to a copy created by a `ND_FEED_NEXT` node

Additionally, if after adding each input's `unique id` to the list, the code generator is also at loop level 0, then the consumed array list is processed as follows. Each `unique id` is removed from the list and a decrement reference count by one statement is generated for it. The consumed array list is then emptied. The sub-sections below detail the behavior of the algorithm for various DDCF graph nodes.

4.6.1 Compound DDCF Nodes

The overall effect of this algorithm is to descend into a compound node's hierarchical graph structure and collect distinct graph inputs at any depth into a list. It then decrements everything in the list when it returns to loop level 0. This typically happens when the code generator is *between* subgraphs in a `ND_FUNC` node.

Some compound nodes form well-known subgraphs entirely composed of array consumers, such that it is pointless to collect distinct graph inputs from the well-known subgraph because they are the same as the inputs for the array consumer nodes. For example, loop generator subgraphs such as `ND_DOT_PROD` and `ND_CROSS_PROD` contain only generator nodes such as `ND_ELE_GEN` and `ND_WINDOW_GEN`, whose inputs are the same as the generator subgraph inputs.

This makes it redundant to attempt to collect the inputs for the generator subgraph into the list. Therefore, `ND_DOT_PROD` and `ND_CROSS_PROD` are not considered array consumers, because they are guaranteed to feed their inputs only to nodes from Table 2.3. Figure 4.7 illustrates a loop generator subgraph for the SA-C loop for `VA(~,:) in A cross VB(:,~) in B`.

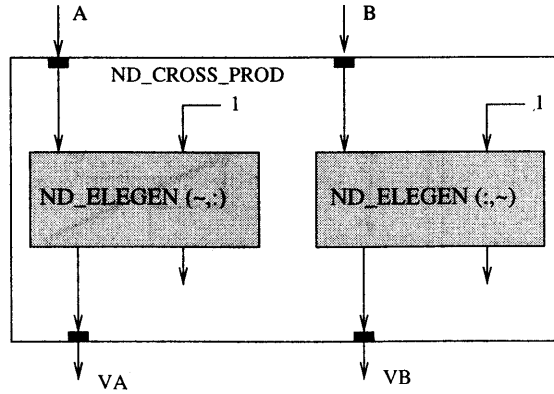


Figure 4.7: Example of DDCF Generator Node

4.6.2 Simple DDCF Nodes

When applied to DDCF graph nodes which are not compound nodes *and* which create dope vectors at loop level 0, the algorithm has the effect of incrementing the reference count to the array when the alias is created, and then immediately decrementing the reference count to the same array after the node is generated. It is possible but somewhat more complicated to statically decide that a given input for a given node should *not* be added to the consumed array list, because that input is aliased inside the node. The algorithm described in this paper does not go to that extent and at loop level 0 it blindly decrements the reference count for everything in the consumed array list.

4.7 Unfinalized Nextified Variables

Consider the following SA-C code, in which the **final** value of nextified variable B is not returned from the dependent loop:

```
uint8[:,:] main (uint8 n, uint8 A[3,2], uint8 B[3,2]) {
    uint8 R[:,:] =
        for uint8 i in [n] {
            uint8 BB[:,:] = A;
            next A =
                for a in A dot b in B
                    return (array ((a+b)/2));
            next B = BB;
        } return (final (A));
} return (R);
```

Within the dependent loop, B is consumed by A's loop generator graph. The code generator will refrain from inserting B in the consumed array list because it is a nextified variable. To determine

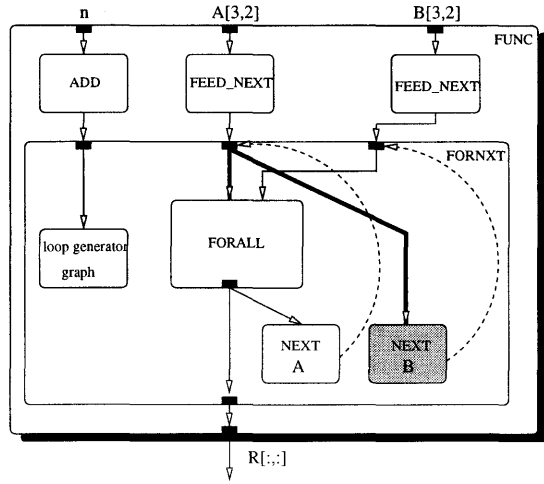


Figure 4.8: Example of Unfinalized Nextified Variable

that B is unfinalized, the code generator looks for an edge that splits to a ND_NEXT node but never to a ND_G_OUTPUT node. Finalized nextified variables always output through a ND_G_OUTPUT node. B is sent only to a ND_FORALL node and a ND_NEXT node but not a ND_G_OUTPUT node (Illustrated in Figure 4.8). Since this is statically determinable, the code generator will add the correct unique id for B to a list of unfinalized nextified variables and generate a decrement reference count for all elements in this list when it next processes the consumed array list. Note that the correct unique id for B is taken by following the back-edge of B's ND_NEXT node. In this case, since B is a partial nextified alias, the back-edge references a different ND_G_INPUT_NEXT node than the one which created the edge split (see Figure 4.8).

Chapter 5

Performance Evaluation

This chapter presents space performance results for each of the garbage elimination techniques. All of the SA-C codes referred to in this chapter were taken from either the **examples** directory of the SA-C test suite or the **sim.test** directory. The **examples** directory consists of 154 SA-C codes and the **sim.test** directory has 169. More information regarding these codes is available from the Cameron research group.

It is possible to produce a **heaptrace** for any SA-C host code program's execution, using the **-heap** command line switch. This heaptrace records all allocations and deallocations of memory blocks (specified as addresses) which occurred while the host code was executing. The difference between the number of allocations and the number of matching deallocations is defined as the amount of garbage in the host code. All amounts of garbage are given in heap leak units rather than bytes. For example, if the amount of garbage is given as 4, there were 4 heap leaks, rather than 4 heap leaks, each of 200 bytes for 800 bytes total.

To determine the space performance benefit derived from each of the garbage elimination techniques of pointer reuse and reference counting it is best to apply only one or the other at a time. The first two sections evaluate each of the techniques by itself. The last section presents several specific performance results, indicating garbage-free host code is consistently produced when both techniques are applied in unison.

5.1 Evaluation of Reference Counting

The amount of garbage (in heap leak units) which reference counting can eliminate is defined as reference counting's **space performance upperbound**. Assume host code is generated which reference counts its array data flow through its DDCF graph using input edges/aliases as producers and function level DDCF nodes as consumers (as described in the last chapter). Reference counting will only occur for each of the declared host code array structs. There will be at least one host code array struct for each array declared/defined in the SA-C code, including loop generator variables, which are not explicitly defined. Intermediate structures and garbage resulting from statically determinable heap leaks will not be considered. It is possible for each of these array structs to have its **mem** pointer pointing to a distinct memory block. The code generator will create unique ids for all variables in the DDCF graph. The code generator can determine the total number of array structs it will create locally for each function and the number of arrays which will be output from the `main()` function. The difference of the two values is reference counting's space performance upperbound.

The following values are relevant:

- Let α = the total number of host code array structs
- Let β = the total number of arrays output from `main`
- Let A = the total number of memory allocations during program execution
- Let F = the total number of matching frees
- reference counting upperbound = $\alpha - \beta$
- amount of garbage = $A - F$

This upperbound is statically computed and is only correct for SA-C codes which do not contain user-defined functions which declare arrays as local variables. The local arrays for any function (including `main()`) are eliminated using reference counting. The latest they could be eliminated is the ending curly brace of the function's statement block. The upperbound computed above adds the array structs for non-main locals exactly *once*. In reality, if a non-main function is invoked more than once, but no reference counting has been performed, then multiple leaked copies of each non-main function local array will exist as garbage, causing the amount of garbage to exceed the statically computed upperbound. Therefore this upperbound should be used only as a *rule of thumb*, rather than a hard and fast value. Here is an example using the SA-C code `histograms.sc` taken

from the SA-C **examples** test suite. This code does not include any user-defined functions. It is executed *without* reference counting but *with* all statically determinable garbage eliminated through pointer reuse.

- $\alpha = 24, \beta = 4, A = 67, F = 55$
- reference counting upperbound = $\alpha - \beta = 20$
- amount of garbage = $A - F = 12$

Since the amount of garbage remaining (12) is less than or equal to the reference counting upperbound (20), it follows that reference counting should eliminate the remaining 12 pieces of garbage. Executing `histograms.sc` again but this time *with* reference counting and pointer reuse enabled yields the following results:

- $\alpha = 24, \beta = 4, A = 67, F = 67$
- reference counting upperbound = $\alpha - \beta = 20$
- amount of garbage = $A - F = 0$

The reference counting technique eliminated the remaining garbage, producing garbage-free host code.

5.2 Evaluation of Pointer Reuse

To show the validity of pointer reuse, reference counting is disabled while pointer reuse is enabled. If for every SA-C code in the test suite that does not contain function calls, the difference ($A - F$) is less than its respective reference counting upperbound, then pointer reuse is functioning correctly and reference counting should eliminate the remaining heap leaks.

5.3 Performance Results

The following 10 SA-C codes comprise *Test Suite A* (Table 5.1). The codes were selected to illustrate certain situations as indicated in the list below:

1. `array_reductions.sc` - various array reductions
2. `concat_test.sc` - concat and tile
3. `histograms.sc` - array histograms

4. mergesort.sc - function call to user-defined function with local array
5. probel.sc - function call to user-defined function with local array that takes a dope vector of its input array
6. slices.sc - substructures taken using array reference nodes
7. tst121.sc - partial nextified aliasing and unfinalized nextified variable (from `sim_test` directory)
8. tst470.sc - tile in a function call
9. while1.sc - loops with loop-carried dependencies
10. zero_trips.sc - zero extent arrays

SA-C Code	Allocations	Garbage (-nrc)	Garbage (ptr reuse <i>only</i>)
array_reductions.sc (26)	26	16	9
concat_test.sc (7)	109	27	7
histograms.sc (20)	67	67	12
mergesort.sc (9)	58	39	19
probel.sc (6)	173	173	4
slices.sc (1)	11	11	1
tst121.sc (6)	37	37	3
tst470.sc (5)	122	57	12
while1.sc (2)	106	32	2
zero_trips.sc (2)	4	3	2

Table 5.1: Test Suite A

The first column of Table 5.1 gives the name of the SA-C code as well as the statically computed reference counting upperbound in parentheses. Each SA-C code was executed using `-heap`. The second column lists the total number of allocations found in the resulting heaptrace file. The third column gives the amount of garbage produced when the respective SA-C code was executed using `-nrc`. The final column gives the amount of garbage produced for SA-C code in which reference counting was disabled, but pointer reuse was enabled. Since `mergesort.sc` and `tst470.sc` contain function calls to user-defined functions with local arrays the amount of garbage produced when only pointer reuse was employed *does* exceed the statically computed upperbound for reference counting.

The other SA-C codes are below the limit however, including `probe1.sc`, which only created sub-structures (dope vectors) of its input array inside its user-defined function.

When both reference counting and pointer reuse are employed, the total amount of garbage for all 10 test codes is 0, indicating garbage-free host code. The sum total amount of garbage for all 154 test codes in the `examples` test suite plus the 169 test codes in the `sim.test` test suite is 0.

5.4 Speed Performance

SA-C Code	Execution Time	Execution Time (-nrc)
<code>array_reductions.sc</code> (26)	0.000u	0.000u
<code>concat_test.sc</code> (7)	0.020u	0.010u
<code>histograms.sc</code> (20)	0.000u	0.000u
<code>mergesort.sc</code> (9)	0.000u	0.000u
<code>probe1.sc</code> (6)	0.000u	0.000u
<code>slices.sc</code> (1)	0.020u	0.000u
<code>tst121.sc</code> (6)	0.010u	0.010u
<code>tst470.sc</code> (5)	0.000u	0.000u
<code>while1.sc</code> (2)	0.010u	0.000u
<code>zero_trips.sc</code> (2)	0.000u	0.000u

Table 5.2: Test Suite B

Table 5.2 gives the name of the SA-C code as well as the execution time with and without reference counting (using the `-nrc` switch). All times are given in microseconds. `slices.sc` shows that reference counting aliases (resulting from dope vectors or conditional expressions) results in a speed performance decrease because the reference count is incremented and then immediately decremented (fruitlessly used). Overall for this test suite, reference counting seems to have little impact on execution times.

5.5 Complexity Analysis

The following figures illustrate typical DDCF graph node arrangements found in most SA-C programs. For each of the arrangements, space requirements will be evaluated in addition to performance gains resulting from either pointer reuse or reference counting techniques.

1. Figure 5.1: $0, \dots, m$ Nested DDCF graphs, exhibiting fine-grained parallelism by moving around in a common source array (the array from the outermost loop)

2. Figure 5.2: $0, \dots, m$ DDCF graphs exhibiting temporal parallelism by sharing a common source array
3. Figure 5.3: $0, \dots, m$ DDCF graphs prohibiting parallelism by imposing a sequential execution order on the graphs

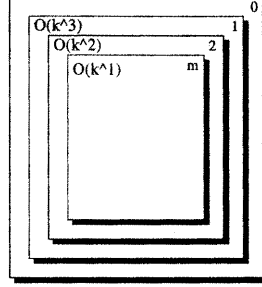


Figure 5.1: Example of Nested DDCF Graphs

Assume an array A has size n . The space complexity for Figure 5.1 is $O(n)$ since all the inner loops work from the outermost source array A . The reference counting scheme employed in this paper will increment the reference count only at level 0 and decrement again only at level 0, so reference counting this type of graph provides **no** performance gain. Consider the scenario in which the innermost loop creates a new array B also of size n . If the innermost loop iterates k times, pointer reuse ensures that at any time, only 1 copy of B exists. Without it, the innermost loop generates $O(k)$ versions of B and the innermost loop is executed a polynomial number of times by its outer loops. If each outer loop also iterates k times, then these graphs require $O(k^m)$ space. Thus these graphs have polynomial space requirements if they create an array in their innermost loop. Pointer reuse yields a polynomial performance gain by keeping the space requirement to $O(1)$ at any give time.

The space complexity for Figure 5.2 is also $O(n)$ because all the graphs share the same source array A . Reference counting provides no direct performance gain, because you will never be able to free A any sooner than the last of the m graphs finishes using it. Reference counting allows these m graphs to be executed in any non-deterministic order without worrying about losing A . Pointer reuse also yields no performance gain on this type of graph.

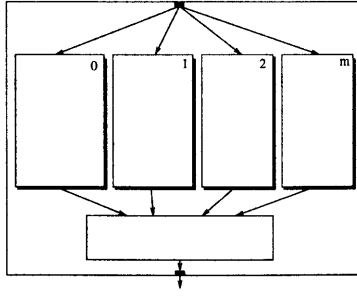


Figure 5.2: Example of Wide DDCF Graphs

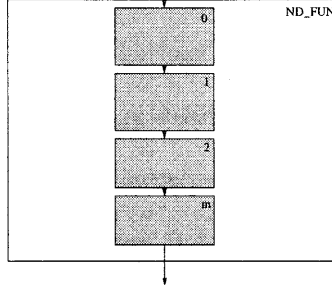


Figure 5.3: Example of Long DDCF Graphs

The space complexity for Figure 5.3 is $O(m * n)$ assuming m to somehow be non-constant. These SA-C programs take an array A and produce the array A' which is then used to produce A^2 and so on m times. Each array requires n elements. Reference counting yields a linear performance gain by freeing A after A' is created, keeping the space complexity to $O(n)$ throughout. Pointer reuse does not yield any performance gain for this type of graph either. Since m is unlikely to be non-constant, reference counting really yields only a constant performance gain.

Chapter 6

Conclusions

At the language definition level, SA-C is a strict single-assignment language with side-effect free statement blocks. To implement these semantics in C host code, several design choices were made. The single-assignment semantics of SA-C were implemented using C pointer semantics, resulting in heap leaks. These occur when a pointer points at one thing and then it points at another thing or nothing at all, possibly losing the address to the first thing it pointed to. Dynamic array extents prevent the code generator from statically determining if an array reference is to an empty array or not. This requires a costly host code size check before any arbitrary access through a reference (dereference) can be made.

Any DDCF graph node will contain a topologically sorted list of subgraph nodes. Using the DDCF graph, host code is sequentially generated and at this time, the topologically sorted order is used to generate all the single-assignment statements of the language. As previously shown, the **next** keyword is position sensitive when defining one nextified variable in terms of an earlier nextified variable. It will create either a full or a partial nextified alias depending only on the order of the nextified assignments. In a strict single-assignment language it should be possible to execute a group of statements in any order desired and always receive the same execution result. This is not possible in SA-C loops which contain loop-carried dependencies, because sequential ordering has been applied for nextified assignment.

Since **pointer reuse** ensures that only one reference to each array payload comes out of a loop, reference counting can be performed outside of all loops, at the function level. The reference count is

taken as the length of the alias chain including the terminating creator reference. Reference counting at the function level is beneficial for two reasons. First it avoids incrementing and decrementing reference counts inside of loops. Second, determining when to eliminate garbage created from an inner loop is dependent on if it is referenced by the outermost loop, making it difficult to determine when to safely eliminate garbage. Reference counting at the function level has none of these difficulties because it simply happens between loop level 0 DDCF nodes.

It has been shown that by combining some static dependency analysis (pointer reuse) with a simple reference counting scheme that all garbage can be eliminated from generated SA-C host code. Furthermore, the majority of the garbage is eliminated simply by applying the static technique of pointer reuse. Since reference counting is beneficial only for a certain type of SA-C program it is an area of future research to enable the SA-C compiler to statically determine whether it should turn on or off reference counting.

REFERENCES

- [1] J. P. Hammes and A. P. W. Böhm. *The SA-C Compiler - Version 1.0.11*. Colorado State University, August 1999.
- [2] J. P. Hammes and A. P. W. Böhm. *The SA-C Compiler Data-Dependence-Control-Flow (DDCF)*. Colorado State University, January 1999.
- [3] J. P. Hammes and A. P. W. Böhm. *The SA-C Language - Version 1.0*. Colorado State University, December 1999.
- [4] J. P. Hammes, R. E. Rinker, D. M. McClure, A. P. W. Böhm, and W. A. Najjar. *The SA-C Compiler Dataflow Description*. Colorado State University, February 2000.
- [5] B. J. MacLennan. *Principles of Programming Languages*. CBS College Publishing, second edition, 1987.