The SA-C Compiler Data-Dependence-Control-Flow (DDCF)

J. P. Hammes and A. P. W. Böhm
Colorado State University

June 21, 2001

Data-Dependence-Control-Flow (DDCF) graphs are used as an intermediate representation in the SA-C compiler, suitable for performing a variety of optimizations. The graphs are acyclic and hierarchical, i.e. some nodes contain subgraphs within them. The entire SA-C language can be represented by DDCF graphs.

1 DDCF Nodes

There are two kinds of DDCF nodes: *simple* nodes are bottom-level, whereas *compound* nodes contain subgraphs. All nodes have input and output *ports* that interface the node to the rest of the graph by means of *edges*. Each input port may have either an incoming edge or a literal value. Each output port has zero or more outgoing edges.

Every port has a SA-C type tag that specifies the data type of the values that travel through the port. When an input port and an output port are connected by an edge, their type tags must match. Some nodes have node-specific information associated with them. For example, a ND_FCALL node will contain the name of the function being called. See section 10 for a complete description of the node-specific imformation that applies to each node type.

Compound nodes contain ND_G_INPUT, ND_G_INPUT_NEXT and ND_G_OUTPUT nodes (called I/O nodes) that serve as the interface between the node's internals and its exterior. Each of these I/O node types has node-specific information that tells which external port the node is associated with, as well as which compound node it lives in. A compound node "knows" its input and output nodes, via arrays of node identifying numbers, as well as its other internal nodes via a linked list of node identifiers.

Figure 1 shows the conventions used when drawing DDCF nodes. For all nodes, input ports occur along the top of the node, and output ports occur along the bottom. There is an implicit left-to-right ordering of the ports. A *simple* node's interior is shown with its node type and any node-specific information that may apply. A *compound* node contains a subgraph, with I/O nodes represented as small black rectangles along the top and bottom of the compound node, to reduce clutter. A ND_G_INPUT_NEXT is distinguished from a ND_G_INPUT by the fact that there is a dashed implicit edge from a ND_NEXT node back to its associated ND_G_INPUT_NEXT node. An input port can be targeted by an edge or by a constant value. All values pass through the boundaries of a compound node via I/O nodes.

Every port has a SA-C *type* tag (see the `TypeInfo` structure in appendix D. The tag tells the kind of data, whether it is scalar or array, the rank, and optionally a size for each dimension if it is known. A '-1' value indicates that a dimension's size is not known; this is represented in a SA-C program by

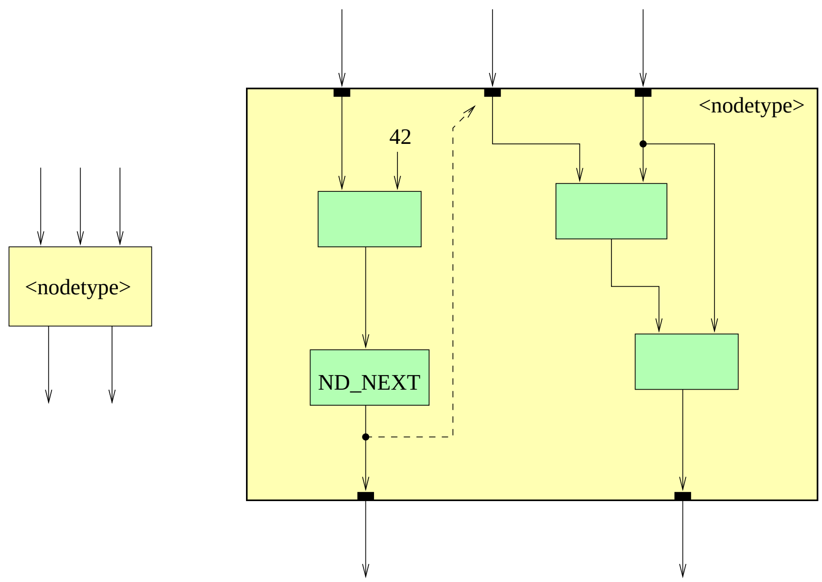


Figure 1: An example of a simple node (left) and compound node (right.) Both have three input ports and two output ports. The compound node contains nine internal nodes (including its I/O nodes), and its middle input is "nextified".

a ':' within the declaration's square brackets. When an edge connects an output port to an input port, the type tags of the ports must match, except for the dimension sizes. (This mismatch can occur only when edges from ND_CASE nodes target the same output port. There may be different sizes on the different output ports, and the input port they target will show a '-1' since the size can vary at run time.) Since an edge connects ports with matching types, it is reasonable to show the edge with that type.

2 DDCF Graphs of SA-C Functions

All functions in a SA-C program, both those that are defined and those that are only prototyped, have an entry in the top-level DDCF data structure represented as a linked list of `FuncGraph`s. See appendix D for detailed information.

The `nodes` array is expandable. The `nodes_allocated` field indicates the size of the array, and the `nodes_used` field indicates the number of nodes that are currently used. Node allocations take place through a call to `alloc_ddcf_node`, which receives the function's `FuncGraph` pointer as its parameter. If all the nodes are used, the routine allocates a larger array, copies the old information into the new array, and deallocates the old array. If the `nodes_used` field equals zero, the function is only a prototype. Otherwise, `nodes[0]` is a ND_FUNC node representing the compound node for the function definition.

3 Numeric and Bit Operator Nodes

Figure 2 shows the names and descriptions of the DDCF nodes that perform numeric and bit-manipulation operations. The nodes are simple, with straightforward meanings.

name	inputs	outputs	cmpnd	description
ND_ADD	2	1	N	add
ND_SUB	2	1	N	subtract
ND_MUL	2	1	N	multiply
ND_DIV	2	1	N	divide
ND_MOD	2	1	N	mod
ND_LT	2	1	N	less than
ND_GT	2	1	N	greater than
ND_LE	2	1	N	less than or equal
ND_GE	2	1	N	greater than or equal
ND_EQ	2	1	N	equal
ND_NEQ	2	1	N	not equal
ND_AND	2	1	N	boolean and
ND_OR	2	1	N	boolean or
ND_BIT_AND	2	1	N	bit and
ND_BIT_OR	2	1	N	bit or
ND_BIT_EOR	2	1	N	bit exclusive or
ND_LEFT_SHIFT	2	1	N	bit left shift
ND_RIGHT_SHIFT	2	1	N	bit right shift
ND_COMPLEX	2	1	N	create complex value from two scalars
ND_REAL	1	1	N	take real value from complex
ND_IMAG	1	1	N	take imaginary value from complex
ND_NOT	1	1	N	boolean not
ND_NEG	1	1	N	numeric negate
ND_MUL_MACH	2	1	N	A multiply node whose output width is the sum of its input widths.
ND_LEFT_SHIFT_MACH	2	1	N	A left shift node whose output width is wider than its left input width.
ND_SQRT_MACH	2	1	N	A square root node whose output width is half of its input width.

Figure 2: Numeric and bit-manipulation nodes. The last three are variations that don't correspond to the traditional SA-C rules for determining the width of the output.

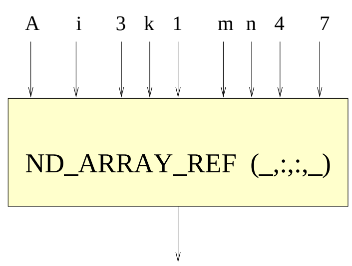
4 Array Nodes

Figure 3 shows the various node types that create and use array values.

The ND_ARRAYREF node can extract both array elements (scalars) or slices. It has node-specific information (see section 10) that indicates the pattern of extraction, in this example `(_,:,:,_)`. The meaning of the inputs is determined by the extraction pattern. For example, the SA-C expression `A[i,3:k,m:n:4,7]` will produce the following node:

name	ins	outs	cmpnd	description
ND_ARRAYREF	var	1	N	This node takes an element or a slice from an array. Node-specific information indicates which dimensions are sliced (via a ':' spec) and which have a specified index. The first input is the source array. In addition, each slice dimension creates three inputs (low, high and step), whereas each specified index creates one input. See text for an example.
ND_ARR_DEF	var	1	N	This node creates a constant array from a collection of scalar values. The number of inputs equals the size of the array. Node-specific information holds the array's shape information.
ND_ARR_CONPERIM	3	1	N	This node creates an array with a constant-value perimeter. The first input is the source array. The second is the perimeter width. The third is the perimeter value.
ND_ARR_CONCAT	2	1	N	This node concatenates two arrays.
ND_ARR_VERT_CONCAT	2	1	N	This node concatenates two 2D arrays in the vertical dimension.
ND_EXTENTS	1	var	N	This returns the extents of the input array. The number of outputs equals the rank of the array.

Figure 3: Array-related nodes



5 Miscellaneous Nodes

Figure 4 shows various other node types. The ND_FUNC node represents an entire function definition. Its input ports represent the function's parameters, and its output ports represent the function's return values. The ND_FUNC node is *always* the first node in that function's array of nodes.

The ND_PRINT and ND_ASSERT nodes may have string constants as inputs. Since the type tags for input ports represent SA-C types, and SA-C has no strings, type information for these inputs does not exist.

The ND_VOIDED node occurs where a previous node in the array has been deleted, and it should simply be skipped over whenever nodes are being processed.

name	ins	outs	cmpnd	description
ND_FUNC	var	var	Y	This is the top-level function node. It has an input for each parameter, and an output for each return value.
ND_FCALL	var	var	N	This node calls a user-defined function. It has an input for each argument, and an output for each return value. Node-specific information indicates which function is called.
ND_INTRINCALL	var	var	N	This node calls an intrinsic function. Node-specific information indicates which function is called.
ND_PRINT	var	0	N	This node prints values and strings. The first input is a boolean, determining whether or not the node will print. The rest are SA-C values and strings.
ND_ASSERT	var	0	N	This node tests a SA-C assertion (the first input.) The other inputs are SA-C values and strings.
ND_CAST	1	1	N	This node casts a value from one type to another.
ND_G_INPUT	0	1	N	This is an input node for a compound parent node. Node-specific information indicates the parent node id, and the port number it associates with.
ND_G_OUTPUT	1	0	N	This is an output node for a compound parent node. Node-specific information indicates the parent node id, and the port number it associates with.
ND_G_INPUT_NEXT	0	1	N	This is an input node for a loop parent node. Node-specific information indicates the parent node id, and the port number it associates with. The input is a "nextified" variable of the loop.
ND_VHDL_CALL	var	var	N	This is a call to an external VHDL routine.
ND_VOIDED	0	0	N	This is an empty spot in the nodes array, and should be ignored.

Figure 4: Miscellaneous nodes

6 Switches

The ND_SWITCH node is a compound node corresponding to switch and conditional expressions in SA-C. Figure 5 describes the nodes associated with switches.

name	ins	outs	cmpnd	description
ND_SWITCH	var	var	Y	This is the top-level switch node. It has an input for each value used in the switch, and an output for each return value.
ND_SWITCH_KEY	1	0	N	This node is a sink for the switch select expression.
ND_SELECTORS	var	0	N	This node is a sink node for the keys of a ND_CASE node.
ND_CASE	var	var	Y	This node corresponds to a case or default of a switch. The number of inputs is the number of values that are needed in the case. The number of outputs is the number of values returned by the switch expression.

Figure 5: Nodes associated with switch expressions

Every ND_SWITCH node contains exactly one ND_SWITCH_KEY node that sinks an expression whose value is used to select among the various ND_CASE graphs. One ND_CASE graph may lack a ND_SELECTORS node, and corresponds to a default in the switch expression. Every other ND_CASE graph has exactly one ND_SELECTORS node that sinks the case values, which are always constants. SA-C conditional expressions (if-else) are represented by ND_SWITCH graphs since the conditional is a special case of the more general switch. The ND_G_OUTPUT nodes of a ND_SWITCH graph are the only DDCF nodes in which an input port can be targeted by more than one edge.

As an example, figure 6 shows the DDCF graph produced by the following SA-C expression:

```
switch (n+2) {
    case 3, 4 : return (m*4)
    case 5    : return (42)
    default   : { uint8 p = n*2+m; } return (p) }
```

7 Loops

Figure 7 shows the top-level loop nodes. There are three loop graph nodes: ND_FORALL, corresponding to a SA-C `for` loop without nextified variables; ND_FORNXT, from a `for` loop with nextified variables; and ND_WHILE, corresponding to a `while` loop. The two `for` loop forms use the loop generator graphs ND_CROSS_PROD and ND_DOT_PROD to produce internal loop values, whereas the ND_WHILE graph uses a ND_WHILE_PRED to enclose the predicate expression that controls the loop. All three loop forms use the same set of loop-return nodes, which are discussed first.

7.1 Loop-return nodes

The various loop-return nodes are all simple nodes. The optional mask that is available in many of the SA-C reductions is mandatory in the DDCF graphs; if the mask was not explicitly specified,

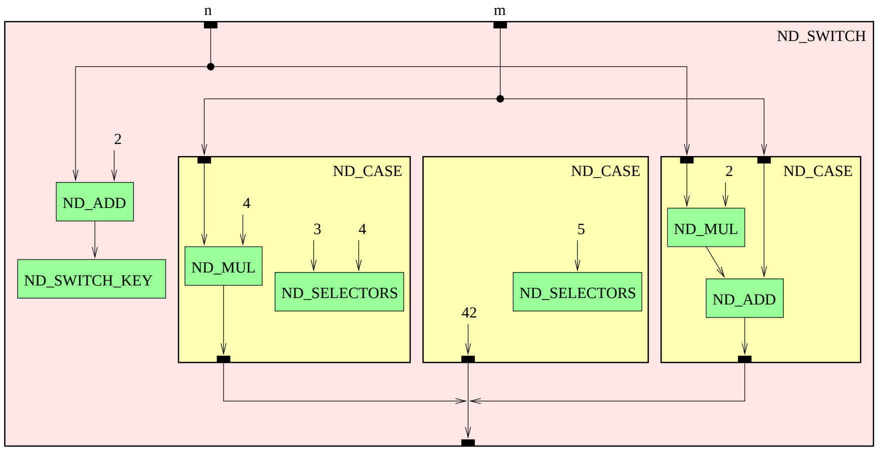


Figure 6: Example of DDCF graph of a switch expression.

name	ins	outs	cmpnd	description
ND_FORALL	var	var	Y	Parallel `for` loop.
ND_FORNXT	var	var	Y	A `for` loop with loop-carried dependencies.
ND_WHILE	var	var	Y	A `while` loop.
ND_WHILE_PRED	var	0	Y	Graph containing the `while` loop predicate expression.

Figure 7: Loop nodes

a `true` input is placed on the mask input of the reduction node. Figure 8 defines the various loop-return nodes.

7.2 Parallel For Loops

Each SA-C `for` loop has exactly one generator graph, either a ND_CROSS_PROD or a ND_DOT_PROD compound node. (When a loop has only one simple generator, i.e. no explicit dot or cross product, it will be enclosed in either a ND_CROSS_PROD or a ND_DOT_PROD graph.) There are three simple generator nodes that can occur within the compound node: ND_ELE_GEN, ND_SCALAR_GEN, and ND_WINDOW_GEN. In the case of a ND_CROSS_PROD graph, the order of the simple generators is determined by the order in which they occur in the graph's list of internal nodes. A ND_LOOP_INDICES node will occur in a ND_CROSS_PROD or ND_DOT_PROD graph if the SA-C loop contains a `loop_indices` call. Figure 9 describes the loop generator nodes.

Figure 10 shows an example of each of the three simple generator nodes, as well as cross- and dot-product examples. Figure 11 shows an example of a parallel `for` loop graph.

7.3 For loops with loop-carried dependencies

A `for` loop with at least one "nextified" variable is designated by a ND_FORNXT node, but is otherwise like a parallel `for` loop. Three simple nodes relate to nextified variables: ND_NEXT, ND_FEED_NEXT and ND_G_INPUT_NEXT. The incoming value for a nextified variable flows to the graph through an external ND_FEED_NEXT node, and into the graph through a ND_G_INPUT_NEXT

name	ins	outs	cmpnd	description
ND_CONSTRUCT_ARRAY	1	1	N	Array return.
ND_CONSTRUCT_CONCAT	1	1	N	Concat return.
ND_CONSTRUCT_TILE	1	1	N	Tile return.
ND_REDUCE_SUM	2	1	N	Sum reduction; the first input is the reduced value, and the second input is the mask.
ND_REDUCE_PRODUCT	2	1	N	Product reduction; the first input is the reduced value, and the second input is the mask.
ND_REDUCE_MIN	2	1	N	Min reduction; the first input is the reduced value, and the second input is the mask.
ND_REDUCE_MAX	2	1	N	Max reduction; the first input is the reduced value, and the second input is the mask.
ND_REDUCE_AND	2	1	N	And reduction; the first input is the reduced value, and the second input is the mask.
ND_REDUCE_OR	2	1	N	Or reduction; the first input is the reduced value, and the second input is the mask.
ND_REDUCE_MEAN	2	1	N	Mean reduction; the first input is the reduced value, and the second input is the mask.
ND_REDUCE_ST_DEV	2	1	N	Standard deviation reduction; the first input is the reduced value, and the second input is the mask.
ND_REDUCE_MODE	2	1	N	Mode reduction; the first input is the reduced value, and the second input is the mask.
ND_REDUCE_MEDIAN	2	1	N	Median reduction; the first input is the reduced value, and the second input is the mask.
ND_REDUCE_VALS_AT_xxxS	2	1	N	Values-at-mins and Values-at-maxs reductions; the first input is the reduced value, and the second input is the mask.
ND_REDUCE_VAL_AT_FIRST_xxxS	2	1	N	Values-at-first-max and Values-at-first-min reduction; the first input is the reduced value, and the second input is the mask.
ND_REDUCE_VAL_AT_LAST_xxxS	2	1	N	Values-at-last-max and Values-at-last-min reduction; the first input is the reduced value, and the second input is the mask.
ND_REDUCE_HIST	3	1	N	Histogram reduction; the first input is the reduced value, the second input is the mask, and the third input is the range.
ND_ACCUM_SUM	4	1	N	Accumulated sum reduction; the first input is the reduced value, the second is the label, the third is the range, and the fourth is the mask.
ND_ACCUM_PRODUCT	4	1	N	Accumulated product reduction; the first input is the reduced value, the second is the label, the third is the range, and the fourth is the mask.
ND_ACCUM_MIN	4	1	N	Accumulated min reduction; the first input is the reduced value, the second is the label, the third is the range, and the fourth is the mask.
ND_ACCUM_MAX	4	1	N	Accumulated max reduction; the first input is the reduced value, the second is the label, the third is the range, and the fourth is the mask.
ND_ACCUM_AND	4	1	N	Accumulated and reduction; the first input is the reduced value, the second is the label, the third is the range, and the fourth is the mask.
ND_ACCUM_OR	4	1	N	Accumulated or reduction; the first input is the reduced value, the second is the label, the third is the range, and the fourth is the mask.
ND_ACCUM_MEAN	4	1	N	Accumulated mean reduction; the first input is the reduced value, the second is the label, the third is the range, and the fourth is the mask.
ND_ACCUM_ST_DEV	4	1	N	Accumulated standard deviation reduction; the first input is the reduced value, the second is the label, the third is the range, and the fourth is the mask.
ND_ACCUM_MEDIAN	4	1	N	Accumulated median reduction; the first input is the reduced value, the second is the label, the third is the range, and the fourth is the mask.
ND_ACCUM_HIST	5	1	N	Accumulated histogram reduction; the first input is the reduced value, the second is the label, the third is the accum range, the fourth is the hist range, and the fifth is the mask.

Figure 8: Loop-return nodes

name	ins	outs	cmpnd	description
ND_CROSS_PROD	var	var	Y	Array cross product graph.
ND_DOT_PROD	var	var	Y	Array dot product graph.
ND_ELE_GEN	var	var	N	Array component generator node. The first input is the source array. One additional input exists for each iterative dimension, representing the step size. It is mandatory and set to '1' if the SA-C source did not specify a step size. The first output is the array component that is generated. One additional output exists for each iterative dimension, representing the array index being produced.
ND_WINDOW_GEN	var	var	N	Array window generator node. The first input is the source array. Two additional inputs exist for each dimension of the source array, one for the window size and one for the step. The first output is the array window that is generated. One additional output exists for each dimension, representing the array index being produced.
ND_SCALAR_GEN	var	var	N	Array scalar generator node. There are three inputs for each value being generated, representing start value, end value and step. There is one output for each value being produced.
ND_LOOP_INDICES	0	var	N	Loop indices node. There is one output for each dimension of the loop.

Figure 9: Loop-generator nodes

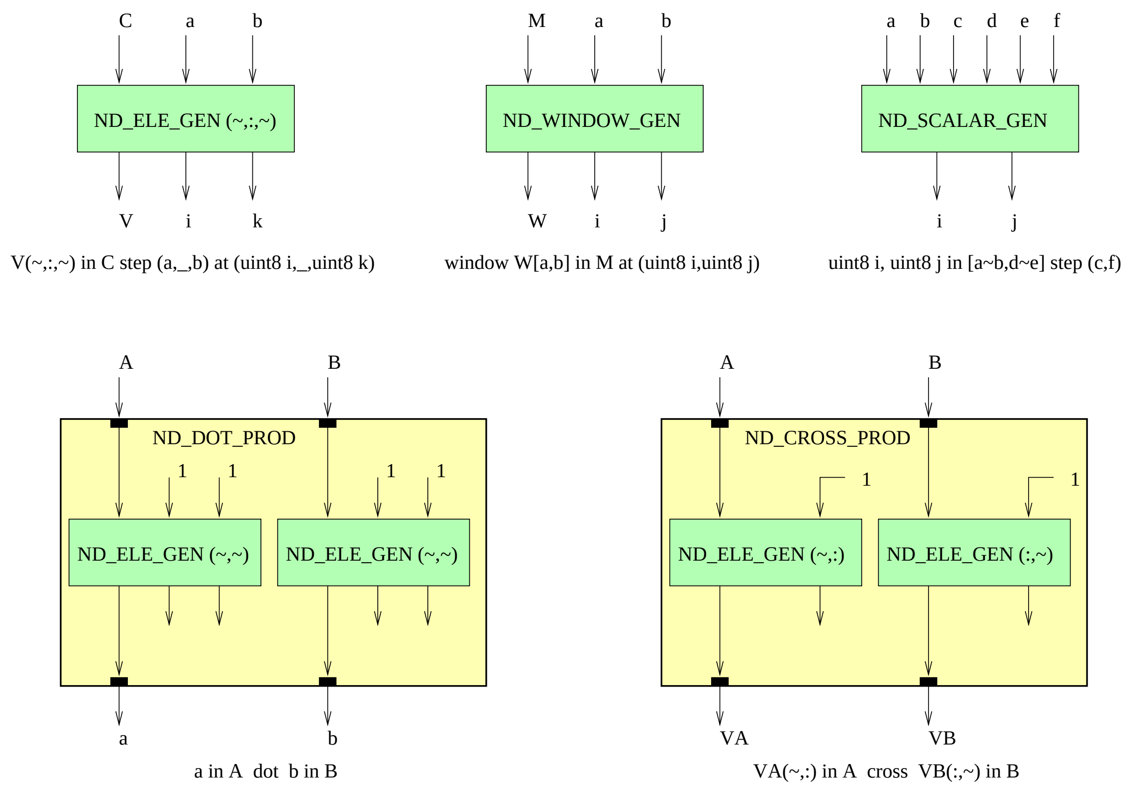


Figure 10: Examples of generator nodes.

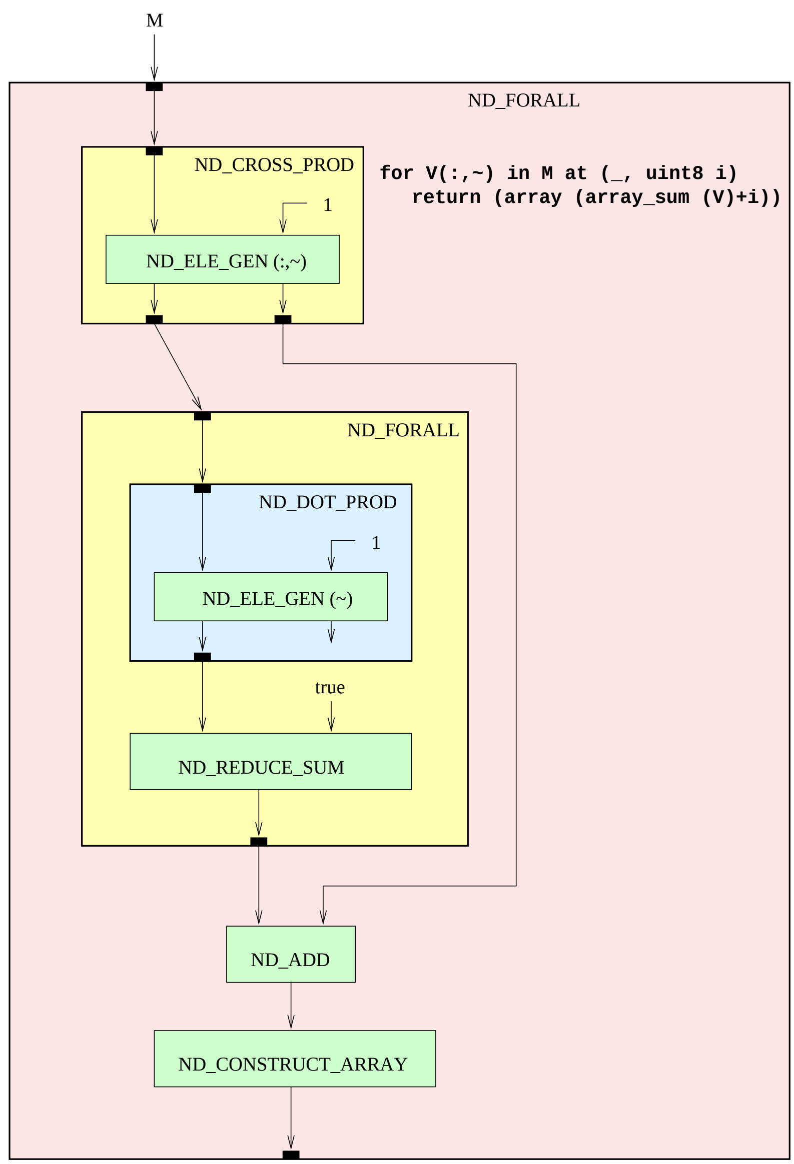
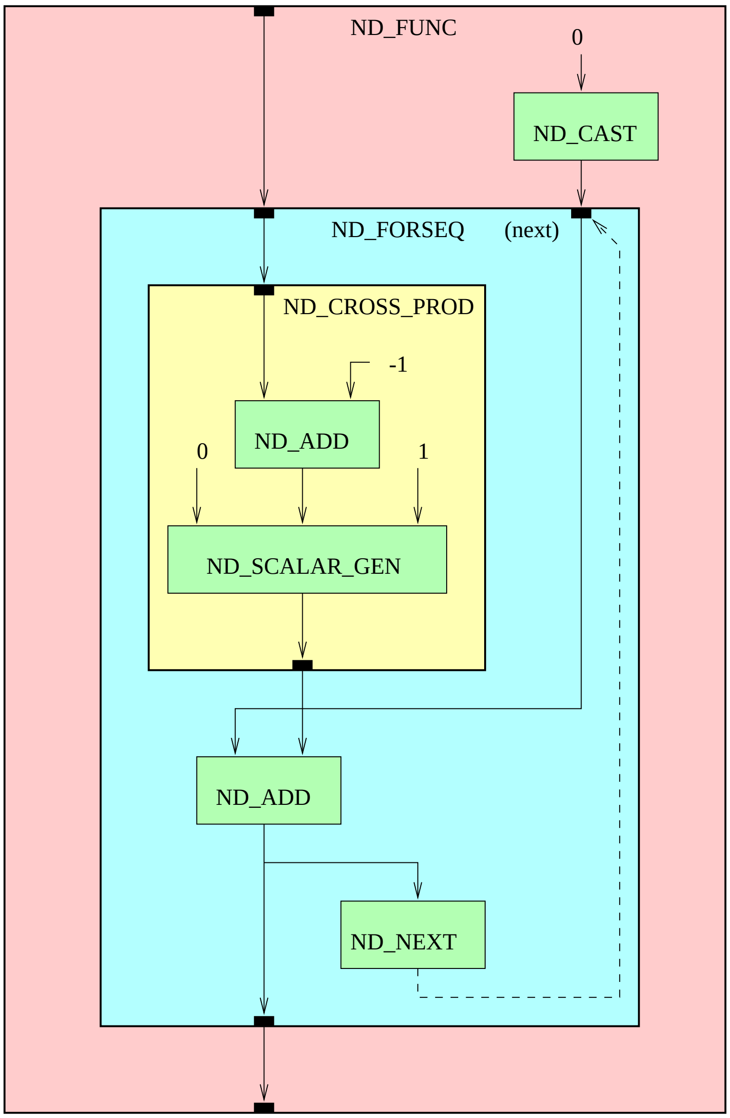


Figure 11: Examples of generator nodes.

node rather than through a ND_G_INPUT node. A value flowing into a ND_NEXT node is the value that is carried into the next iteration of a loop. It has an implied back edge to the ND_G_INPUT_NEXT node associated with the nextified variable; this back edge is represented as node-specific information for the ND_NEXT node. Figure 12 shows a ND_FORNXT example.




```
uint8 main (uint8 n) {
  uint8 ac = 0;
  uint8 res =
    for uint8 i in [n] {
      next ac = ac + i;
      } return (final (ac));
  } return (res);
```

Figure 12: Example of a for loop with a nextified variable.

7.4 While Loops

SA-C `while` loops use the same return reductions and array constructors as used by `for` loops. `While` loops have no generators, but rather use a predicate to determine whether to continue iterating. The ND_WHILE_PRED graph encloses the predicate expression. Its result goes to its output port but does not connect to anything externally. Figure 13 shows an example of a `while` loop.

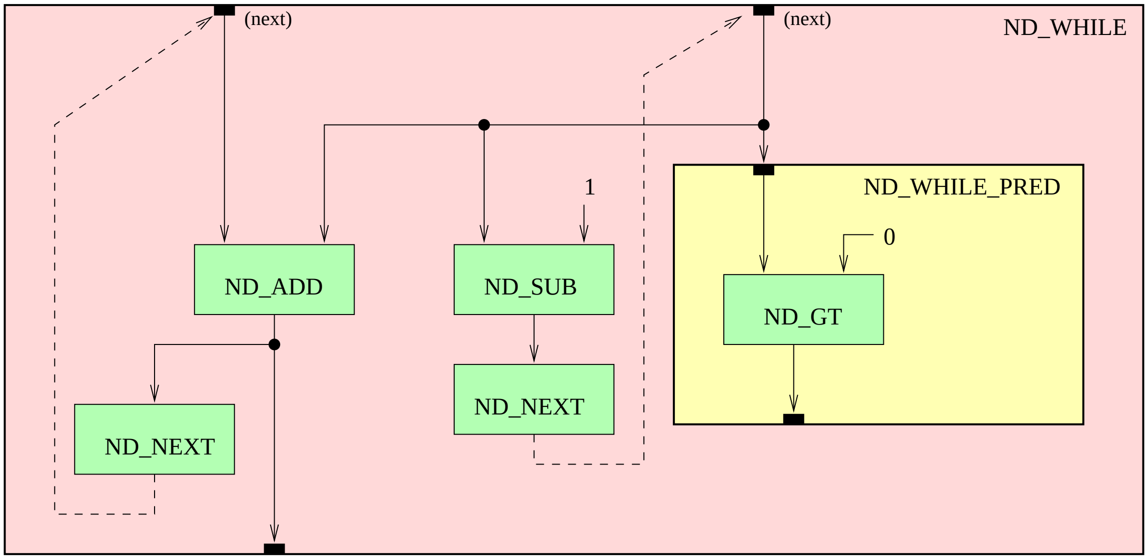


Figure 13: Example of a while loop.

8 MACRO nodes

When the SA-C compiler unrolls a loop, it uses a set of nodes related to the various REDUCE nodes to form the result. These nodes have a variable number of inputs, depending on the number of iterations the unrolled loop had. The following nodes have pairs of inputs, the first conveying a value and the second a boolean indicating whether the value should be included in the reduction:

- ND_REDUCE_SUM_MACRO
- ND_REDUCE_PRODUCT_MACRO
- ND_REDUCE_AND_MACRO
- ND_REDUCE_OR_MACRO
- ND_REDUCE_MIN_MACRO
- ND_REDUCE_MAX_MACRO
- ND_REDUCE_MEDIAN_MACRO
- ND_REDUCE_UMEDIAN_MACRO
- ND_REDUCE_IMEDIAN_MACRO

The six nodes of the VAL_AT_xxx family have input counts based on the number of values being captured, referred to here as v. There will be an input cluster for each iteration of the unrolled loop, and each input cluster will have $v + 2$ inputs, one for the compared value, one for the boolean value, and v for the capture values associated with that iteration. The outputs differ between 1D and 2D return cases. The ND_REDUCE_VAL_AT_MAXS and ND_REDUCE_VAL_AT_MINS each returns 2D a array as a single output. The other four return 1D arrays with known extent, so they have v outputs, one for each of the captured values.

The ND_REDUCE_HIST_MACRO node takes a pair of inputs for each iteration of the unrolled loop, and one more input to convey the extent of the returned array. There is one output, which is the result array.

9 DFG-related nodes

When a DDCF loop is being converted to a DFG, a variety of new nodes are introduced.

10 Implementation

Refer to appendix D for the various data structure definitions used to store DDCF graphs. There is a `FuncGraph` for each SA-C function, and each has an array of DDCF nodes. (If the function is just a propotype, the nodes array is NULL.) The `DdcfNode` structure represents a node in a DDCF graph, and has the following fields:

nodetype The kind of node it is.

loc File, function, and line number information relating back to the SA-C source.

num_inputs The number of input ports the node has.

inputs Pointer to an array of `InputPort` structures.

num_outputs The number of output ports the node has.

outputs Pointer to an array of `OutputPort` structures.

specific A union containing information that varies with node type. This is discussed in more detail in the following section.

dim_sizes Used only for FOR loop nodes. It contains loop shape information.

10.1 Node-specific information

Node-specific information applies to the following node types:

ND_NEXT An integer tells the node id of the ND_G_INPUT_NEXT node that is associated with this node. This is an implicit back edge for this value in the next iteration.

ND_INPUT An integer tells what port number this node associates with in the parent graph. Another integer tells the id of the parent graph.

ND_INPUT_NEXT Same as ND_INPUT.

ND_OUTPUT Same as ND_INPUT.

ND_FCALL A string tells the name of the called function.

ND_INTRINCALL An `Intrinsic` value specifies the intrinsic function being called.

ND_ARRAYREF A character string specifies a pattern of ':' and '_' that specify sliced and non-sliced dimensions.

name	ins	outs	cmpnd	description
ND_RC_COMPUTE	var	var	Y	This graph encloses a DFG loop and its interface code.
ND_MALL_XFER	1	1	N	This node's input takes an input array, and mallocs space for it. The output is the address of the allocated memory.
ND_SIZE	1	1	N	This node takes an input array, and returns the size (i.e. number of elements).
ND_W_LOOP_EXTENT	3	1	N	This node takes extent, size and step as inputs, and returns the number of iterations that result from those values.
ND_MALLOC_TGT_<n>_BIT_<m>D	var	1	N	This node has an input for each dimension of the array it is allocating space for. The rank is <m>. The inputs are the extents, and the output is the address of the allocated memory. The bit-width of the values is <n>.
ND_TRANSFER_TO_HOST_RET_ARRAY	var	1	N	This node transfers an array back to the host. Its first input is a trigger from the DFG, the second is the address of the source array, and the rest are the array's extents.
ND_TRANSFER_TO_HOST_SCALAR	2	1	N	This node transfers a scalar back to the host. Its first input is a trigger from the DFG, the second is the address of the scalar.
ND_RC_FORALL	var	var	Y	This is a FOR loop graph that is being turned into a DFG.
ND_RC_WINDOW_GEN_1D	4	var	N	This is a 1D window generator. Its four inputs are an array address, a window size, a window step, and the array's extent. The number of outputs is equal to the window size. The outputs are the scalar values of the window, followed by the index of the window.
ND_RC_WINDOW_GEN_2D	7	var	N	This is a 2D window generator. Its first input is an array address. This is followed by two sets of three inputs, conveying the window size, a window step, and array's extent for each of its two dimensions. The number of outputs is equal to the window size. The outputs are the scalar values of the window, followed by two outputs for the indices of the window.
ND_RC_SLICE_GEN_2D_COL	4	var	N	This is a column slicing generator. Its first input is an array address. The second is the step size. The last two are the input array's extents. The number of outputs is equal to the slice size. The outputs are the scalar values of the slice, followed by an output for the index.
ND_RC_SLICE_GEN_2D_ROW	4	var	N	This is a row slicing generator. Its first input is an array address. The second is the step size. The last two are the input array's extents. The number of outputs is equal to the slice size. The outputs are the scalar values of the slice, followed by an output for the index.
ND_RC_ELE_GEN_<n>D	var	var	N	This is a scalar-extracting generator. The first input is the address of the source array. This is followed by a pair of inputs for each dimension, conveying the step size and extent. The first output is the scalar value taken from the array. This is followed by an index output for each dimension of the array.
ND_WRITE_TILE_<n>D_<m>D	var	1	N	This node writes a tile to memory, where n is the dimension of the tile and m is the dimension of the loop. The first v inputs are the scalar values of the tile. After that there is one input for the target address of the result array, followed by m inputs for the loop extents, followed by n inputs for the tile extents. The output is a termination trigger.

Figure 14: DFG-related nodes

ND_ELE_GEN A character string specifies a pattern of ':' and '~' that specify sliced and iterated dimensions.

ND_SCALAR_GEN An integer tells the rank of the generator.

ND_ARR_DEF An integer tells the rank of the array. An array of integers tells the extent of each dimension.

ND_REDUCE_VAL_AT_MAXS An integer tells the number of values being collected.

ND_REDUCE_VAL_AT_MINS Same as ND_REDUCE_VAL_AT_MAXS.

any compound node An integer array tells the node ids of this graph's ND_G_INPUT and ND_G_INPUT_NEXT nodes. Another integer array tells the node ids of this graph's ND_G_OUTPUT nodes. An integer list tells what nodes are within this graph. The order of the nodes in this list is relevant in two ways. First, it represents a valid topological sort of the nodes in the graph, meaning that code generation can be performed simply by stepping through the list rather than following graph edges. Second, for a ND_CROSS_PROD node, the simple generators within the cross product appear in the list in left-to-right order with regard to the SA-C source that produced the loop.

10.2 Text representation

The SA-C compiler's '-ddcf' option will output a text representation of the internal DDCF structures. Appendix E shows a BNF syntax for the text file. Figures 16 and 15 show the text form and its graphic representation for the following SA-C function:

```
uint8, float f1 (uint8 A[:,:])
    return (array_min (A), array_mean ((float[:,:])A));
```

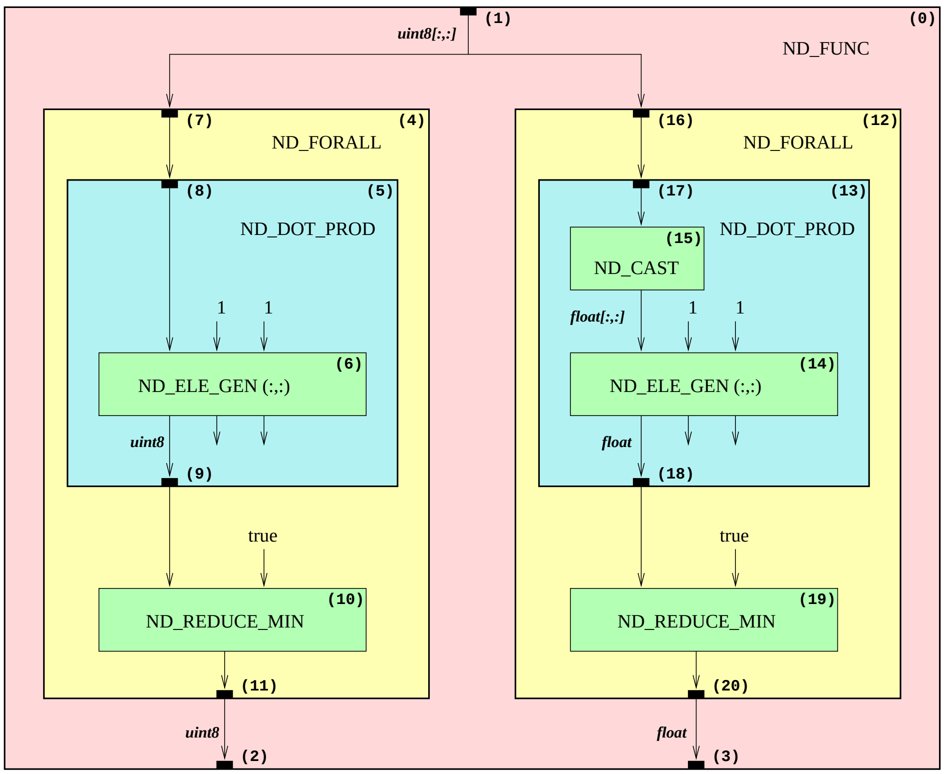
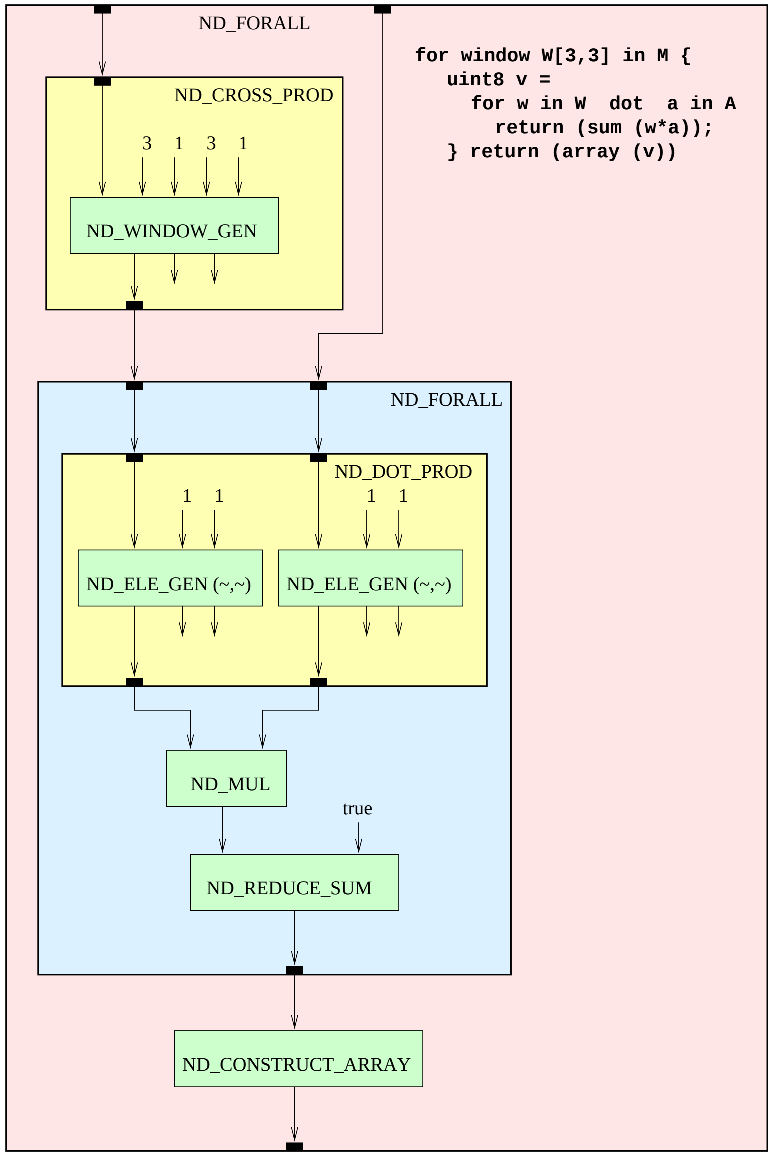


Figure 15: Graphical representation of function example in text.

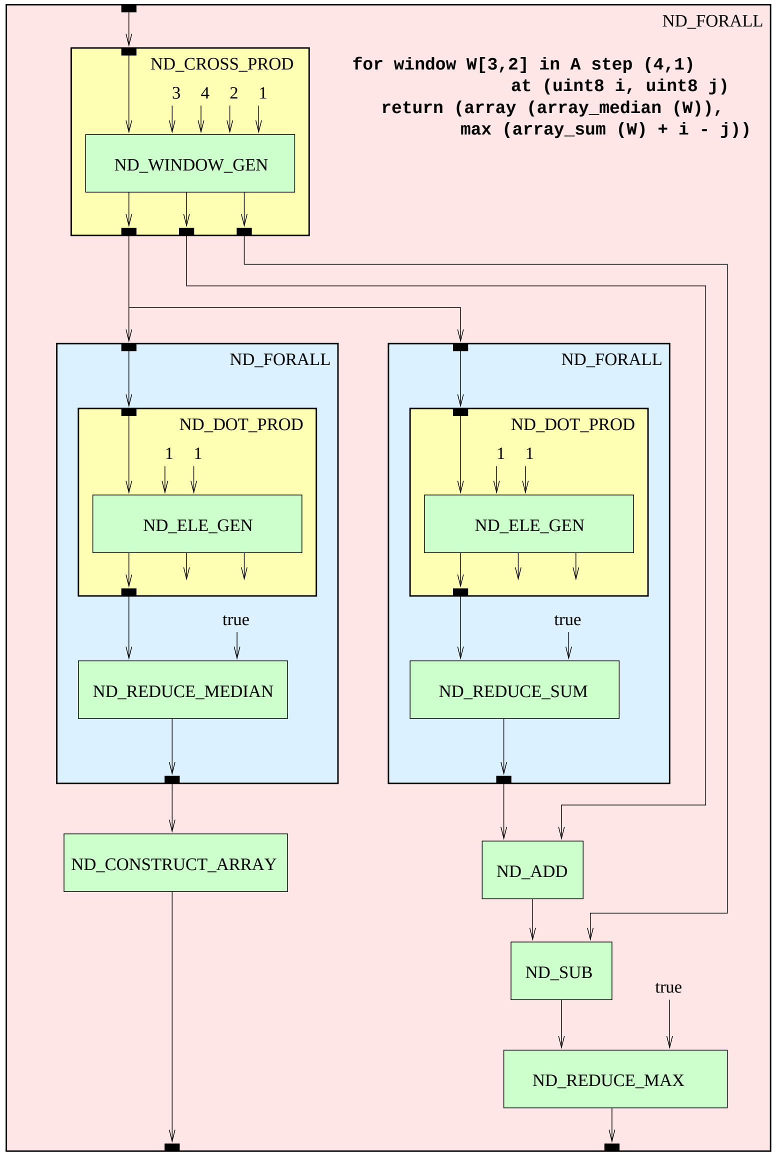
```
function "f1"                      param types : uint8[:,:]
                                   return types: uint8, float
  0 ND_FUNC (my nodes:  1 4 12) <"xx.sc", "f1", 1>
    1 inputs: (nodes  1)           port 0 <uint8[:,:]>
    2 outputs: (nodes  2 3)        port 0 <uint8>
                                   port 1 <float> ;
  1 ND_G_INPUT  (input 0 for graph node 0) <"xx.sc", "f1", 1>
    0 inputs:
    1 outputs:                     port 0 <uint8[:,:]>   12.0  4.0 ;
  2 ND_G_OUTPUT  (output 0 for graph node 0) <"xx.sc", "f1", 1>
    1 inputs:                      port 0 <uint8>
    0 outputs: ;
  3 ND_G_OUTPUT  (output 1 for graph node 0) <"xx.sc", "f1", 1>
    1 inputs:                      port 0 <float>
    0 outputs: ;
  4 ND_FORALL (my nodes:  7 5 10) extents [:,:,:,:,:,:,:,:] <"xx.sc", "f1", 2>
    1 inputs: (nodes  7)           port 0 <uint8[:,:]>
    1 outputs: (nodes  11)         port 0 <uint8>   2.0 ;
  5 ND_DOT_PROD (my nodes:  8 6) <"xx.sc", "f1", 2>
    1 inputs: (nodes  8)           port 0 <uint8[:,:]>
    1 outputs: (nodes  9)          port 0 <uint8>   10.0 ;
  6 ND_ELE_GEN  [~,~] <"xx.sc", "f1", 2>
    3 inputs:                      port 0 <uint8[:,:]>
                                   port 1 <uint1>  value "1"
                                   port 2 <uint1>  value "1"
    3 outputs:                     port 0 <uint8>   9.0
                                   port 1 <uint32>
                                   port 2 <uint32> ;
  7 ND_G_INPUT  (input 0 for graph node 4) <"xx.sc", "f1", 2>
    0 inputs:
    1 outputs:                     port 0 <uint8[:,:]>   5.0 ;
  8 ND_G_INPUT  (input 0 for graph node 5) <"xx.sc", "f1", 2>
    0 inputs:
    1 outputs:                     port 0 <uint8[:,:]>   6.0 ;
  9 ND_G_OUTPUT  (output 0 for graph node 5) <"xx.sc", "f1", 2>
    1 inputs:                      port 0 <uint8>
    0 outputs: ;
  10 ND_REDUCE_MIN <"xx.sc", "f1", 2>
    2 inputs:                      port 0 <uint8>
                                   port 1 <bool>  value "true"
    1 outputs:                     port 0 <uint8>   11.0 ;
  11 ND_G_OUTPUT  (output 0 for graph node 4) <"xx.sc", "f1", 2>
    1 inputs:                      port 0 <uint8>
    0 outputs: ;
  12 ND_FORALL (my nodes:  15 13 20 18) extents [:,:,:,:,:,:,:,:] <"xx.sc", "f1", 2>
    1 inputs: (nodes  15)          port 0 <uint8[:,:]>
    1 outputs: (nodes  19)         port 0 <float>   3.0 ;
  13 ND_DOT_PROD (my nodes:  16 14) <"xx.sc", "f1", 2>
    1 inputs: (nodes  16)          port 0 <uint8[:,:]>
    1 outputs: (nodes  17)         port 0 <uint8>   20.0 ;
  14 ND_ELE_GEN  [~,~] <"xx.sc", "f1", 2>
    3 inputs:                      port 0 <uint8[:,:]>
                                   port 1 <uint1>  value "1"
                                   port 2 <uint1>  value "1"
    3 outputs:                     port 0 <uint8>   17.0
                                   port 1 <uint32>
                                   port 2 <uint32> ;
  15 ND_G_INPUT  (input 0 for graph node 12) <"xx.sc", "f1", 2>
    0 inputs:
    1 outputs:                     port 0 <uint8[:,:]>   13.0 ;
  16 ND_G_INPUT  (input 0 for graph node 13) <"xx.sc", "f1", 2>
    0 inputs:
    1 outputs:                     port 0 <uint8[:,:]>   14.0 ;
  17 ND_G_OUTPUT  (output 0 for graph node 13) <"xx.sc", "f1", 2>
    1 inputs:                      port 0 <uint8>
    0 outputs: ;
  18 ND_REDUCE_MEAN <"xx.sc", "f1", 2>
    2 inputs:                      port 0 <float>
                                   port 1 <bool>  value "true"
    1 outputs:                     port 0 <float>   19.0 ;
  19 ND_G_OUTPUT  (output 0 for graph node 12) <"xx.sc", "f1", 2>
    1 inputs:                      port 0 <float>
    0 outputs: ;
  20 ND_CAST <"xx.sc", "f1", 2>
    1 inputs:                      port 0 <uint8>
    1 outputs:                     port 0 <float>   18.0 ;
```

Figure 16: Text representation of function example.

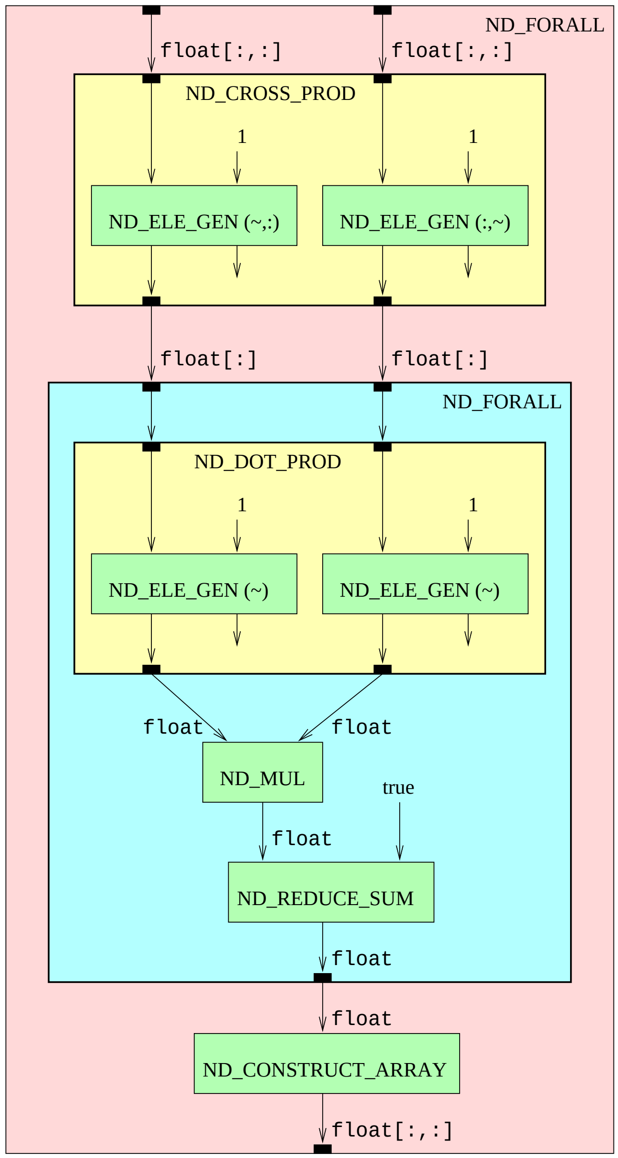
A Example of loop with window generator



B Example of loop with stepped window generator



C Matrix multiply




```
for VA(~,:) in A  cross
                VB(:,~) in B {
   float Ele =
      for a in VA  dot  b in VB
          return (sum (a*b));
   } return (matrix (Ele))
```

D Internal typedefs

```
#include <stdio.h>

extern FILE *yyin;

#define MAXRANK 8
#define FALSE 0
#define TRUE (!FALSE)

typedef struct {
    int line;
    char *file;
    char *func;
    } Location;

typedef struct intcell
    {
    int val;
    struct intcell *link;
    } IntList;

typedef enum {
    Tnone,
    Unknown,    /* type not yet known */
    Bits,       /* bit vector */
    Bool,       /* boolean */
    Uint,       /* unsigned integer */
    Int,        /* signed integer */
    Ufix,       /* unsigned fixed point */
    Fix,        /* signed fixed point */
    Float,      /* 32-bit floating point */
    Double,     /* 64-bit floating point */
    CxInt,      /* complex with signed integer */
    CxFix,      /* complex with signed fixed point */
    CxFloat,    /* complex with 32-bit floats */
    CxDouble    /* complex with 64-bit floats */
    } Type;

typedef enum {
    Knone,
    Array,      /* array */
    Scalar,     /* scalar */
    } Kind;

typedef struct tinfo
    {
    Type type;
    int totsize;
    int fracsize;
    Kind kind;
    int dims;
```

```
    int dim_sizes[MAXRANK];     /* -1 indicates no size available */
    struct tinfo *link;
    } TypeInfo;

typedef enum {
    /* compound nodes */
    ND_SWITCH,
    ND_CASE,
    ND_WHILE,
    ND_FORALL,
    ND_FORNXT,
    ND_FUNC,
    ND_CROSS_PROD,
    ND_DOT_PROD,
    ND_WHILE_PRED,

    /* 2-input, 1-output operator nodes */
    ND_ADD,
    ND_SUB,
    ND_MUL,
    ND_DIV,
    ND_MOD,
    ND_COMPLEX,
    ND_LT,
    ND_GT,
    ND_LE,
    ND_GE,
    ND_NEQ,
    ND_EQ,
    ND_AND,
    ND_OR,
    ND_BIT_AND,
    ND_BIT_OR,
    ND_BIT_EOR,
    ND_LEFT_SHIFT,
    ND_RIGHT_SHIFT,

    /* loop-related nodes */
    ND_SCALAR_GEN,
    ND_ELE_GEN,
    ND_WINDOW_GEN,
    ND_LOOP_INDICES,
    ND_REDUCE_SUM,
    ND_REDUCE_MIN,
    ND_REDUCE_MAX,
    ND_REDUCE_AND,
    ND_REDUCE_OR,
    ND_REDUCE_VAL_AT_MAXS,
    ND_REDUCE_VAL_AT_MINS,
    ND_REDUCE_PRODUCT,
    ND_REDUCE_MEAN,
    ND_REDUCE_ST_DEV,
    ND_REDUCE_MODE,
```

```
ND_REDUCE_MEDIAN,
ND_REDUCE_HIST,
ND_CONSTRUCT_ARRAY,
ND_CONSTRUCT_CONCAT,
ND_CONSTRUCT_TILE,

ND_ACCUM_SUM,
ND_ACCUM_MIN,
ND_ACCUM_MAX,
ND_ACCUM_AND,
ND_ACCUM_OR,
ND_ACCUM_PRODUCT,
ND_ACCUM_MEAN,
ND_ACCUM_ST_DEV,
ND_ACCUM_MEDIAN,
ND_ACCUM_HIST,

/* new macro nodes for unrolled loops */
ND_REDUCE_SUM_MACRO,
ND_REDUCE_PRODUCT_MACRO,
ND_REDUCE_AND_MACRO,
ND_REDUCE_OR_MACRO,
ND_REDUCE_MIN_MACRO,
ND_REDUCE_MAX_MACRO,
ND_REDUCE_MEDIAN_MACRO,

/* input and output nodes for compounds */
ND_G_INPUT,
ND_G_INPUT_NEXT,
ND_G_OUTPUT,

/* 1-input, 1-output operator nodes */
ND_REAL,
ND_IMAG,
ND_NOT,
ND_NEG,

/* other various kinds... */
ND_ARR_CONCAT,
ND_ARR_VERT_CONCAT,
ND_CAST,
ND_FCALL,
ND_EXTENTS,
ND_INTRINCALL,
ND_ARR_CONPERIM,
ND_ARRAYREF,
ND_ARR_DEF,
ND_SWITCH_KEY,
ND_SELECTORS,
ND_NEXT,
ND_PRINT,
ND_ASSERT,
ND_VOIDED,
```

```
ND_FEED_NEXT,
ND_GRAPH,

ND_RC_COMPUTE,
ND_MALL_XFER,
ND_RANK,
ND_SIZE,
ND_W_LOOP_EXTENT,
ND_MALLOC_TGT_8_BIT,
ND_MALLOC_TGT_16_BIT,
ND_MALLOC_TGT_32_BIT,
ND_TRANSFER_TO_HOST_RET_ARRAY,
ND_TRANSFER_TO_HOST_SCALAR,
ND_RC_FORALL,
ND_RC_WINDOW_GEN_1D,
ND_RC_WINDOW_GEN_2D,
ND_RC_ELE_GEN_1D,
ND_RC_ELE_GEN_2D,
ND_RC_ELE_GEN_3D,
ND_WRITE_AND_ADVANCE,
ND_WRITE_TILE_1D_1D,
ND_WRITE_TILE_1D_2D,
ND_WRITE_TILE_1D_3D,
ND_WRITE_TILE_2D_1D,
ND_WRITE_TILE_2D_2D,
ND_WRITE_TILE_2D_3D,
ND_WRITE_TILE_3D_1D,
ND_WRITE_TILE_3D_2D,
ND_WRITE_TILE_3D_3D,
ND_REDUCE_ISUM_MACRO,
ND_REDUCE_USUM_MACRO,
ND_REDUCE_IMIN_MACRO,
ND_REDUCE_UMIN_MACRO,
ND_REDUCE_IMAX_MACRO,
ND_REDUCE_UMAX_MACRO,
ND_USUM_VALUES,
ND_ISUM_VALUES,
ND_UMIN_VALUES,
ND_IMIN_VALUES,
ND_UMAX_VALUES,
ND_IMAX_VALUES,
ND_AND_VALUES,
ND_OR_VALUES,
ND_IADD,
ND_UADD,
ND_ISUB,
ND_USUB,
ND_ULT,
ND_ILT,
ND_ULE,
ND_ILE,
ND_UGT,
ND_IGT,
```

```
    ND_UGE,
    ND_IGE,
    ND_UEQ,
    ND_IEQ,
    ND_UNEQ,
    ND_INEQ,
    ND_BIT_COMPL,
    ND_CHANGE_WIDTH,
    ND_CHANGE_WIDTH_SE,
    ND_RC_SELECTOR
    } DdcfType;

typedef enum {
    IntrinSin,
    IntrinCos,
    IntrinTan,
    IntrinAsin,
    IntrinAcos,
    IntrinAtan,
    IntrinAtan2,
    IntrinSinh,
    IntrinCosh,
    IntrinTanh,
    IntrinAsinh,
    IntrinAcosh,
    IntrinAtanh,
    IntrinSqrt,
    IntrinCbrt,
    IntrinPow,
    IntrinModf,
    IntrinExp,
    IntrinFrexp,
    IntrinLdexp,
    IntrinLog,
    IntrinLog10,
    IntrinExpm1,
    IntrinLog1p,
    IntrinCeil,
    IntrinFabs,
    IntrinFloor,
    IntrinFmod,
    IntrinCopysign,
    IntrinHypot,
    IntrinRint
    } Intrinsic;

typedef struct edge {
    int node;
    int port;
    struct edge *link;
    } Edge;

typedef struct {
```

```
    int id;
    TypeInfo ty;
    int is_const;
    char constval[128];
    Edge *back_edges;
    } InputPort;

typedef struct {
    int id;             /* port number for its node */
    int unique_id;      /* unique id within its function */
    TypeInfo ty;
    Edge *targets;
    } OutputPort;

#define In_next_id specific.in_next_id
#define Io_num specific.io_info.io_num
#define My_graph specific.io_info.my_graph
#define My_inputs specific.g_info.my_inputs
#define My_outputs specific.g_info.my_outputs
#define My_nodes specific.g_info.my_nodes
#define Sym specific.sym
#define Intrin_func specific.intrin_func
#define Reftypes specific.reftypes
#define DefDims specific.arr_def_info.dims
#define DefRank specific.arr_def_info.rank
#define Rank specific.rank
#define VecSize specific.vecsize

typedef struct {
    DdcfType nodetype;
    Location loc;
    int num_inputs;
    InputPort *inputs;
    int num_outputs;
    OutputPort *outputs;
    union {
        int in_next_id;         /* for ND_NEXT nodes */
        struct {
            int io_num;
            int my_graph;
            } io_info;          /* for ND_G_INPUT and ND_G_OUTPUT nodes */
        char *sym;              /* for ND_FCALL nodes */
        Intrinsic intrin_func;  /* for ND_INTRINCALL nodes */
        char *reftypes;         /* for ND_ARRAYREF and ND_ELE_GEN nodes */
        int rank;               /* for ND_SCALAR_GEN nodes */
        struct {
            int rank;
            int *dims;
            } arr_def_info;     /* for ND_ARR_DEF nodes */
        struct {
            int *my_inputs;
            int *my_outputs;
            IntList *my_nodes;
```

```
            } g_info;           /* for any compound graph node */
        int vecsize;            /* for ND_REDUCE_VAL_AT_MxxS */
        } specific;
    int dim_sizes[8];           /* to hold extents of loop nodes */
    } DdcfNode;

typedef struct funcgraph {
    char *name;
    TypeInfo *params;
    TypeInfo *rets;
    DdcfNode *nodes;
    int nodes_used;
    struct funcgraph *link;
    } FuncGraph;

typedef struct loop_strings {
    int id;
    DdcfType ty;
    int n;
    int var[8];
    char init[8][256];
    char incr[8][256];
    char terminate[8][256];
    char size[8][256];
    char totsize[256];
    struct loop_strings *link;
    } GenStrings;
```

E BNF of DDCF file format

```
rule 1    program -> functions
rule 2    functions -> function
rule 3    functions -> function functions
rule 4    function -> TOK_FUNCTION TOK_STRING params returns nodes
rule 5    params -> TOK_PARAM TOK_TYPES ’:’ types
rule 6    params -> TOK_PARAM TOK_TYPES ’:’
rule 7    returns -> TOK_RETURN TOK_TYPES ’:’ types
rule 8    types -> type
rule 9    types -> type ’,’ types
rule 10   type -> scalar_type dims
rule 11   scalar_type -> TOK_UINT
rule 12   scalar_type -> TOK_INT
rule 13   scalar_type -> TOK_UFIX
rule 14   scalar_type -> TOK_FIX
rule 15   scalar_type -> TOK_FLOAT
rule 16   scalar_type -> TOK_DOUBLE
rule 17   scalar_type -> TOK_BITS
rule 18   scalar_type -> TOK_BOOL
rule 19   scalar_type -> TOK_COMPLEX TOK_INT
rule 20   scalar_type -> TOK_COMPLEX TOK_FIX
rule 21   scalar_type -> TOK_COMPLEX TOK_FLOAT
rule 22   scalar_type -> TOK_COMPLEX TOK_DOUBLE
rule 23   dims -> /* empty */
rule 24   dims -> ’[’ places ’]’
rule 25   places -> place
rule 26   places -> place ’,’ places
rule 27   place -> ’:’
rule 28   place -> TOK_UINTNUM
rule 29   nodes -> /* empty */
rule 30   nodes -> nodes node
rule 31   node -> TOK_UINTNUM TOK_NODETYPE special_info loop_extents ’<’
          TOK_STRING ’,’ TOK_STRING ’,’ TOK_UINTNUM ’>’ opt_prags inputs outputs ’;’
rule 32   node -> TOK_UINTNUM TOK_NODETYPE ’;’
rule 33   opt_prags -> /* empty */
rule 34   opt_prags -> TOK_PRAGMAS ’(’ prag_list ’)’
rule 35   prag_list -> prag
rule 36   prag_list -> prag_list ’,’ prag
rule 37   prag -> TOK_NO_INLINE
rule 38   prag -> TOK_NO_UNROLL
rule 39   prag -> TOK_NO_FUSE
rule 40   prag -> TOK_LOOKUP
rule 41   prag -> TOK_NO_DFG
rule 42   prag -> TOK_STRIPMINE ’(’ numlist ’)’
rule 43   numlist -> TOK_UINTNUM
rule 44   numlist -> numlist ’,’ TOK_UINTNUM
rule 45   loop_extents -> /* empty */
rule 46   loop_extents -> TOK_EXTENTS ’[’ lval ’,’ lval ’,’ lval ’,’ lval
          ’,’ lval ’,’ lval ’,’ lval ’,’ lval ’]’
rule 47   lval -> ’:’
rule 48   lval -> TOK_UINTNUM
```

```
rule 49   special_info -> /* empty */
rule 50   special_info -> '(' TOK_INPUT TOK_UINTNUM TOK_FOR TOK_GRAPH
          TOK_NODE TOK_UINTNUM ')'
rule 51   special_info -> '(' TOK_OUTPUT TOK_UINTNUM TOK_FOR TOK_GRAPH
          TOK_NODE TOK_UINTNUM ')'
rule 52   special_info -> '(' TOK_MY TOK_NODES ':' numbers ')'
rule 53   special_info -> '[' gen_pattern ']'
rule 54   special_info -> '(' TOK_BACK TOK_TO TOK_NODE TOK_UINTNUM ')'
rule 55   special_info -> TOK_STRING
rule 56   special_info -> TOK_RANK TOK_UINTNUM
rule 57   special_info -> TOK_VECSIZE TOK_UINTNUM
rule 58   numbers -> /* empty */
rule 59   numbers -> numbers TOK_UINTNUM
rule 60   gen_pattern -> gen_place
rule 61   gen_pattern -> gen_place ',' gen_pattern
rule 62   gen_place -> '~'
rule 63   gen_place -> ':'
rule 64   gen_place -> '_'
rule 65   gen_place -> TOK_UINTNUM
rule 66   inputs -> TOK_UINTNUM TOK_INPUTS ':' opt_identify input_ports
rule 67   input_ports -> /* empty */
rule 68   input_ports -> input_ports input_port
rule 69   input_port -> port_spec opt_value
rule 70   port_spec -> TOK_PORT TOK_UINTNUM '<' type '>'
rule 71   port_spec -> TOK_PORT TOK_UINTNUM '<' '>'
rule 72   opt_value -> /* empty */
rule 73   opt_value -> TOK_VALUE TOK_UINTNUM
rule 74   opt_value -> TOK_VALUE TOK_STRING
rule 75   opt_value -> TOK_VALUE TOK_TRUE
rule 76   opt_value -> TOK_VALUE TOK_FALSE
rule 77   outputs -> TOK_UINTNUM TOK_OUTPUTS ':' opt_identify output_ports
rule 78   output_ports -> /* empty */
rule 79   output_ports -> output_ports output_port
rule 80   output_port -> port_spec targets
rule 81   targets -> /* empty */
rule 82   targets -> targets target
rule 83   target -> TOK_UINTNUM '.' TOK_UINTNUM
rule 84   opt_identify -> /* empty */
rule 85   opt_identify -> '(' TOK_NODES numbers ')'
```