INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI®

DISSERTATION

COMPILING SA-C TO RECONFIGURABLE COMPUTING SYSTEMS

Submitted by
Jeffrey P Hammes
Department of Computer Science

In partial fulfillment of the requirements
for the Degree of Doctor of Philosophy
Colorado State University
Fort Collins, Colorado
Summer 2000

COLORADO STATE UNIVERSITY

April 14, 2000

We hereby recommend that the dissertation COMPILING SA-C TO RECONFIGURABLE COMPUTING SYSTEMS prepared under our supervision by Jeffrey P Hammes be accepted as fulfilling in part requirements for the degree of Doctor of Philosophy.

Committee on Graduate Work

Committee Member

Committee Member

Committee Member

Adviser

Department Head

ABSTRACT OF DISSERTATION

COMPILING SA-C TO RECONFIGURABLE COMPUTING SYSTEMS

Field Programmable Gate Arrays (FPGAs) have been available for approximately fifteen years and have experienced speed and density improvements similar to those of microprocessors. Current FPGAs can be reprogrammed in a matter of milliseconds, making them interesting candidates for reconfigurable computing, where specialized circuits can be produced for specific programs to execute more efficiently than a sequential program. Algorithms that are highly regular and exhibit parallelism may benefit from the use of FPGAs.

A significant roadblock to this use of FPGAs is the difficult nature of programming them. Hardware description languages have been the predominant tools for creating FPGA circuit configurations, but these languages are low level and require digital circuit expertise as well as explicit handling of timing. To bring FPGAs into mainstream use by conventional programmers, familiar algorithmic language paradigms must be available, with compilers that can convert high level codes to FPGA configurations.

This research presents SA-C (derived from "Single-Assignment C"), a pure functional algorithmic language intended for the expression of image processing (IP) applications. SA-C's functional nature makes the compiler's job easier, as compared with imperative languages: parallelism is easy to detect, and analysis and transformations are more straightforward. Perhaps the most important part of the language is its loop *window generators*, which not only express many IP operations in an elegant way but are highly useful in expressing optimizing transformations within the compiler.

A Data Dependence and Control Flow (DDCF) hierarchical graph form is also presented, as an

intermediate form with which the SA-C compiler performs its optimizations. These optimizations fall into two broad categories: graph simplifying and loop restructuring. The former are primarily conventional optimizations such as common subexpression elimination and constant folding. The loop restructuring optimizations include loop unrolling, stripmining and fusion, applied as DDCF-to-DDCF transformations using window generators. The compiler, after performing optimizations, is able to convert many inner loops to a low-level, flat dataflow graph designed for translation to VHDL and finally to FPGA configurations. The effects of the compiler's optimizations have been measured on some small kernel codes, and the loop restructuring optimizations are shown to be highly effective.

Jeffrey P Hammes
Department of Computer Science
Colorado State University
Fort Collins, Colorado 80523
Summer 2000

ACKNOWLEDGEMENTS

Thanks are hereby expressed to the entire Cameron Project team. Not only has the SA-C compiler been a pleasure to write, but collaboration with the team members and their enthusiasm have made the work all the more satisfying.

Special thanks go to my advisor, Wim Böhm. He understands research as well as anyone I have met, and his suggestions and guidance have always been on target.

DEDICATION

To my parents.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction and background

Advances in conventional microprocessor technology continue at a rapid pace, bringing some CPU-intensive applications, previously feasible only on supercomputers, into the realm of conventional workstations and PCs. Nevertheless, there remain important applications that consume significant resources on these machines, limiting their usefulness. Improvements in reconfigurable hardware technology have paralleled those in CPU technology: field programmable gate arrays (FPGAs) are steadily gaining both in circuit density and in ease of reprogramming. The research described in this dissertation focuses on the problem of compiling a pure functional, algorithmic language to a machine consisting of a conventional personal computer (PC) supplemented with a FPGA-based accelerator board.

FPGAs have existed for more than fifteen years, being used commonly as "glue" logic to interconnect functional units in digital systems. Since their introduction, they have become increasingly easy to reprogram, opening the door to a use for which they were not intended: as reconfigurable logic for an executing program. This makes it possible to create FPGA configurations specifically designed and optimized for an individual program. Prior to running the program, or even during its execution, these configurations can be downloaded into the FPGAs with the intent of performing computations faster than the host processor could perform them on its own.

FPGAs enjoy the same growth in circuit density that has made traditional microprocessors more and more powerful over time. However, an important stumbling block for the user who may wish to use FPGAs is the difficulty of programming them. The usual method of programming has involved

the use of hardware description languages (HDLs) that require extensive circuit knowledge to use. If a more conventional algorithmic language can be compiled successfully to FPGAs, the door is opened for the ordinary programmer to use these devices.

The remainder of this chapter presents some background for this research, including a description of FPGAs, dataflow and functional languages. The chapters that follow describe the functional language SA-C, the optimizations that have been implemented in its compiler, and performance measurements that test the effectiveness of these optimizations.

1.1 Field Programmable Gate Arrays (FPGAs)

Field programmable gate arrays were invented in the mid-1980s as devices that resembled Application Specific Integrated Circuits (ASICs) but could be programmed after the chip was manufactured. They have become a well-established market since then. Xilinx, a major manufacturer of FPGAs, recently announced that its Virtex™ series of FPGAs yielded an accumulated revenue of more than $100 million in its first year of production [29]. The company also shipped its first two-million system-gate FPGA (in its Virtex-E™ family) in late 1999 [28].

FPGAs represent one of a number of programmable logic devices currently available. An FPGA consists of a matrix of programmable logic cells, with a grid of interconnect lines running between them (see figure 1.1) [5]. In addition, I/O cells exist around the perimeter, providing an interface between the interconnect lines and the chip's external pins. The exact functionality of a logic cell varies with the manufacturer of the chip, but it is typically comprised of a small amount of functional logic and/or some register store capability. Programming an FPGA consists of specifying the logic function of each cell, interconnecting the cells by programming their connections to the interconnects, and programming the I/O cell functions and their interconnects.

There are two predominant methods of configuring an FPGA. The first uses *antifuses*, where the configuration process consists of creating connections between interconnect wires at the desired locations within the chip. Once such a connection has been made, it is permanent, so this technique is not useful for reconfigurable computing. The second configuration approach uses SRAM-controlled

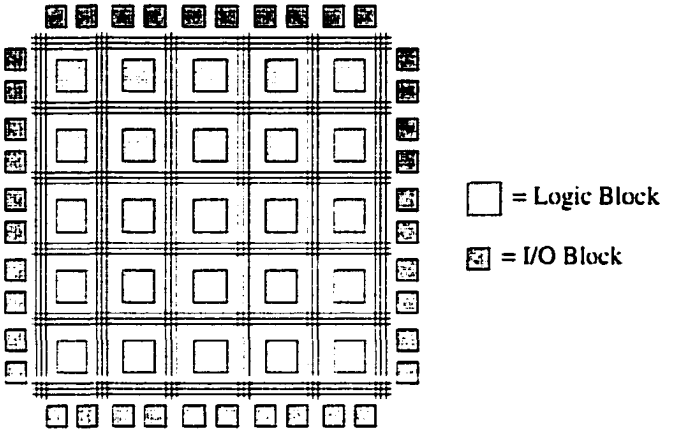


Figure 1.1: Structure of an FPGA

internal switches that pass or block signals at the appropriate locations. This approach allows relatively fast reprogramming, simply by writing new data into the SRAMs. This ease of configuration makes SRAM-based FPGAs a good potential choice for use in reconfigurable computing [25].

The Xilinix XC4036XL is an example of a commonly used FPGA. It is made up predominantly of 1,296 Configurable Logic Blocks (CLBs) and 288 user I/O Blocks (IOBs). Each CLB has two 16-bit static random access memories (SRAMs), two D flip-flops, and a small number of miscellaneous multiplexors and gates. A CLB can function as RAM or as combinational logic. It has 13 inputs and four outputs that meet with the FPGA's internal interconnects. Each IOB controls one external pin of the chip, and contains two D flip-flops and associated buffer-driver circuits. It has five inputs and two outputs that meet the FPGA's internal interconnects. The chip supports system synchronous clock rates of up to 80 MHz [43].

It is useful to relate an FPGA, used as a computing device, to a conventional RISC microprocessor. At a low level, both consist of gates that implement boolean logic. The conventional processor uses its logic to implement a fixed instruction set, in which each instruction has been chosen based on its usefulness to compilers in generating code for general purpose computing. The instructions are very simple, and performance is achieved by devising ways in which they can be rapidly loaded and executed.

The configurability of an FPGA makes it possible to depart from the idea of a fixed instruction set. Instead, application-specific instructions can be specified, and these instructions can be orders of magnitude more complex than those found in RISC processors. The instructions can make heavy use of both breadth-wise and pipeline parallelism. The tradeoff is in the load time of the instruction. Whereas a RISC instruction on a conventional processor can load in a few nanoseconds, the load time of an FPGA instruction can be measured in milliseconds. This makes FPGAs suitable only for parts of applications that are highly regular and work on large data sets. Andre DeHon [11] defines *computational density* as the number of bit operations a device can perform per unit of area-time, and calculates that "reconfigurable architectures provide roughly an order of magnitude higher raw computational density than the programmable architectures." This provides hope that reconfigurable systems can achieve good performances for applications that are suited to them.

The architectural combining of FPGAs with CPUs is an area of active research. Compton and Hauck describe four possible levels of CPU/FPGA coupling [9]. In the tightest, there may be one or more small FPGAs serving as functional units on the CPU's datapath. Somewhat less coupled is an FPGA coprocessor, which can perform operations concurrently with the CPU. A third level is an attached FPGA-based processing unit that acts like an additional processor in the system. The last is an external standalone unit. The level of coupling affects the granularity of the CPU/FPGA program partitioning, as well as the expense required to design and produce the system.

Virtually all FPGAs presently are programmed using one of the various hardware description languages, such as VHDL (VHSIC Hardware Description Language) [36], or directly with logic circuit diagrams. In such low-level programming, where signal timing and synchronization issues must be dealt with explicitly, code development can be slow and tedious. Furthermore, some useful optimizations, such as loop unrolling, must be performed explicitly by the programmer, making it difficult to explore implementation alternatives and their performances.

1.2 Functional languages

The history of functional languages dates back to 1958 when John McCarthy designed the Lisp language [32]. Since then, a variety of functional languages have been created, all built around the central idea of specifying a computation purely with expressions and function calls, and eliminating explicit memory transactions from the semantics of the language. The term *referential transparency* refers to the idea that evaluations of expressions, including function calls, cannot cause side-effects. Thus a function call's result is based only on its arguments, and not on *when* the evaluation takes place. This leads to inherent concurrency. For any pure functional program, all terminating evaluation orders are guaranteed to produce the same result [14].

One of the chief differentiating characteristics among the modern strongly-typed functional languages is *strictness*, which has to do with order of evaluation as well as with mechanisms for triggering evaluations and accessing their results. An argument expression to a function is called "strict" if, prior to the call, the argument is evaluated and its result is used in the call. A strict language is one in which all arguments are strict. Sisal [33] is a strict language, except for its streams. On the other hand, *non-strict* languages can come in at least two forms: *lazy* and *lenient*.

Lazy evaluation postpones the evaluation of function arguments: the function is called and a given argument will be evaluated only if the function discovers that the argument is needed; it is possible that some arguments are *never* evaluated. Also, since a function's argument may itself be in an expression that is an argument to another function call, the *demands* for evaluation of an argument expression may ripple through many calls. In fact, the lazy evaluation of a program begins with a demand for the program's output, which causes evaluation through backward propagation of demands, subsequent evaluations, and forward flowing values. Lazy evaluation avoids computing values that are not needed during an execution, but requires the implementation of unevaluated expressions, called *thunks*, as well as a demand mechanism. Strictness analysis in a compiler may reveal arguments that are statically guaranteed to be evaluated, in which case the argument can be treated as strict. A modern example of a lazy language is Haskell [27].

The *lenient* approach evaluates arguments concurrently with a function's call. When it is implemented, there must be an appropriate mechanism so that, when the called function tries to read an argument's value, it can detect whether the value is present; if it is not present, that part of the function execution must suspend until the value is available. An example of a lenient language is Id [35].

A function is called *higher-order* if any of its arguments or return values is a function. The combination of laziness and higher-order functions encourages a style of programming in which functions are often *composed* in pipeline-like way. A higher-order function captures a pattern of computation and allows the computation itself to be specified by a function passed as one of its arguments. Laziness allows these general purpose higher-order functions to ignore termination issues, since unneeded values will not be computed. This lazy, higher-order approach also allows the use of unbounded data structures, since only the necessary parts of the structure will actually be computed.

A language, or a program, is called *pure* if it is free of side effects, i.e. it does not explicitly manipulate memory. Some functional languages have been extended with one or more *impure* layers, that give the programmer the ability to explicitly allocate, read and write memory locations.

Early functional languages did not have arrays, but rather used lists and trees as data structures for storing data. This made these languages poor candidates for use in scientific or data-intensive codes, due to non-constant time access into these structures. Arrays pose problems for functional languages, both in the specification of an array and in the "updating" of an existing array. (Strictly speaking, an array cannot be updated since a pure functional language cannot express the idea of writing to a memory location.) In a pure language, an array must be specified monolithically. If the language is non-strict, the values for some elements of the array may be expressions referring to other elements of that array. Updating an array really means creating a new array, with the new elements specified, and the remaining elements copied from the previous array. However, compiler analysis may be able to determine that some updates can be done in place on the existing array, if the pre-updated array can be shown not to be referenced after the update [7]. Functional language

arrays typically are dynamically sized and "know" their extents; i.e. the language has a function that returns the extents of an array argument.

Most functional languages rely on recursive functions, rather than loops, to implement iteration. However, loops are possible in functional languages, though the expression-oriented nature of these languages requires that a loop produce one or more return values rather than act through side effects. Lazy, higher order languages often have library functions that are themselves recursive, but hide the recursion and allow the programmer to program at a higher level, where neither loops nor recursion are visible.

While some proponents of functional languages value them simply for their expressiveness, researchers in parallel computing find them interesting because of the inherent parallelism in pure functional programs. This parallelism can is derived from the freedom from side effects. As soon as side effects are introduced, parallelism may become more difficult to detect. Imperative languages represent the extreme case, where the very nature of a program's execution involves side effects, and detection of parallelism presents a significant challenge.

While the promise of "free parallelism" from functional languages is an attractive one, a programmer who wishes to get good parallel performance must have a thorough understanding of the evaluation orders that arise from the language's semantics and implementation [21, 24]. Further, when performance is important, the programmer needs both relevant feedback from the compiler/run-time system to show where problems may be occurring, and mechanisms to control and tune the system based on that feedback. These control mechanisms should be at as high a level as possible, while still allowing sufficient control [3].

1.3 Dataflow

A dataflow graph expresses a computation using nodes for operations and directed edges between nodes to convey values, called *tokens*, from a node output to one or more node inputs. The dataflow model has been used as a virtual machine description for use as compiler intermediate forms, as well as for actual target machines for parallel computation. Dataflow graphs are attractive for expressing

parallel computations because the only constraints they place on parallelism and order of evaluation are those caused by data dependencies.

Jack Dennis, a pioneer in dataflow research, identifies two principal forms of dataflow [12]: *static architecture* and *tagged token*. The latter is closer to the semantics of high-level programming languages: It has mechanisms to support recursive function calls and concurrent execution of loop iterations. The token taggings allow the appropriate tokens to be matched when multiple invocations of a graph are concurrently active. Two well-known tagged token dataflow architectures are the Manchester Dataflow Machine [17] and the Monsoon machine developed by MIT and Motorola [26].

The static approach does not support function calls and concurrent graph invocations. Because of this, resources do not require dynamic (run time) allocation. The targeting of static dataflow from a high-level language requires compiler analysis that can remove function calls and convert to an execution model that is stream or pipeline oriented. Parallel execution of loop iterations is achieved either by pipelining or by creating multiple copies of the loop body and steering tokens among the copies.

The Sisal [33] compiler has used a hierarchical dataflow graph form called "IF1" as an intermediate representation [40]. It has been found to be an excellent form for the application of many optimizations. A graph in IF1 is acyclic, and has four parts:

- *Nodes* are operations that take place within the graph. Nodes may be *simple* or *compound*: the latter contain subgraphs.
- *Edges* carry values between nodes.
- *Types* are associated with edges and functions.
- *Graph boundaries* enclose groups of nodes and edges, and have input and output ports.

One important kind of graph is the **forall**, which has three subgraphs: the *generator*, the *body*, and the *returns*. The generator describes the values associated with the loop's iterations. The body describes the computations performed with those values for an iteration. The returns describe the

values that are accumulated and returned by the loop. (Since this is a functional language, a loop is an expression and therefore must return one or more values.)

The Sisal language has been targeted to the Manchester Dataflow Machine, with various optimizations applied to the dataflow graph representation [4]. They discuss standard optimizations (e.g. common subexpression elimination), function inlining, array copy avoidance, and vectorization.

1.4 The Cameron Project

The Cameron Project [34, 22] began in 1998 as a DARPA-funded[1] effort between the Computer Science Department of Colorado State University and Khoral Research, Inc. to "shift the programming paradigm of adaptive and reconfigurable systems from hardware centered to software centered." The project focuses on image processing applications, and consists of the following parts:

1. Design and implement a language that supports image parallel programming and that can be effectively compiled to reconfigurable hardware.

2. Develop a set of domain-specific library modules for use as building blocks in the design of FPGA-based programs.

3. Integrate the design environment with Khoros [31], a commonly used program development environment in the image processing community.

4. Provide a rapid prototyping adaptive computing capability based on pre-compiled libraries of FPGA programs.

The Khoros environment is a graphical user interface that allows a user to connect together, in a dataflow-like way, image processing modules. The nodes are called *glyphs*, which are selected from menus and correspond to basic image processing routines such as convolution, median filter, etc. The user develops an application by selecting glyphs and interconnecting them.

---

[1]This work is supported by DARPA under US Air Force Research Laboratory contract F33615-98-C-1319.

An important part of the Cameron effort involves the creation of a functional language, designed both to allow the straightforward expression of image processing algorithms and with FPGA-based target architectures in mind. The language is called "Single Assignment C," or SA-C. In spite of the performance problems that many function languages have exhibited, they remain attractive for such a project since their semantics make certain kinds of compiler analysis much easier than those for imperative languages. An important emphasis in making this effort succeed is to provide the user with better control and feedback on performance-related issues.

In the image processing community, *Vector/Signal/Image Processing Library* API, referred to as *VSIPL* has been developed with the goal of standardizing many of the routines needed by the community [10]. The VSIPL library has been written in SA-C and integrated into the Khoros environment, and efficient compilation of these routines, as well as compilation of programs that use these routines, is an ongoing focus of the Cameron project.

1.5 Related work

In addition to the many reconfigurable hardware projects currently underway at various research institutions, there are also some attempts at compilation of conventional languages to FPGA-based targets.

As part of the "Brass" project at the University of California at Berkeley, the *garpcc* compiler has been developed [6]. Their goal is to compile "dusty deck" C programs for execution on the Garp processor. Since this processor is a tightly-coupled CPU and reconfiguarble logic block, it is feasible for them to direct execution to the CPU or the FPGA on an iteration-by-iteration basis. They use trace scheduling to identify the most commonly executed paths and compile these to hardware configurations by merging basic blocks into "hyperblocks".

As part of the PipeRench project at Carnegie Mellon University, the *DIL* programming language (Data-Flow Intermediate Language) has been created [15]. It is an intermediate-level, single assignment language targeting the PipeRench CVH (Configurable Virtual Hardware) chip. A DIL program is a description of a synchronous circuit.

The Oxford University Computing Laboratory has created a language called Handel-C, designed specifically with the intent of compiling to hardware [16]. Handel-C lets the user specify integer bit-widths in variable declarations as well as for subexpressions. The language has a strong hardware orientation: it includes RAM and ROM constructs and explicit timing in the form of clock cycles.

The David Sarnoff Research Center and the Supercomputing Research Center have created a data parallel C language called *dbC*, similar to C* and others, for compilation to reconfigurable hardware. It has parallel control constructs and variable bit-widths. They compile *dbC* to a generic three-address SIMD form, and allow the programmer to specify the partitioning of the problem between the host and the RCS.

The MATCH Project at Northwestern University uses MATLAB as a source language. They target hardware made up of FPGAs and digital signal processors (DSPs). Their approach is to identify and pre-compile commonly-used primitives. Compilation of an application consists of composing the primitives by generating "glue code".

Chapter 2

The SA-C language

When the Cameron Project was created, with the goal of compiling a high-level computer language to reconfigurable hardware, the group had two alternatives: an existing language could be used, or a new language could be created. There are obvious advantages to using an existing language: There is an existing base of knowledgeable programmers for many current languages, as well as a base of existing compilers that could be used as a starting point for the Cameron project's development. Also, programmers are much more willing to develop codes in a familiar language than in a new language that requires a significant effort to learn.

Nevertheless, a new language was created for Cameron, called SA-C [20] (derived from "Single Assignment C" but pronounced "Sassy"). The paramount consideration in creating a new language was the ability to focus the group's efforts solely on the aspects of compilation that are interesting when compiling to an RCS. If the C language [30], for example, had been chosen as the project language, some kind of support would have had to be given for all parts of the language, even though many of those parts not only have nothing useful to offer when compiling to reconfigurable hardware, but may even make the job of compilation significantly more difficult. Using C would require dealing with that language's primitive treatment of arrays, as well as with the extremely difficult issue of explicit pointers and aliasing. It would also require doing extensive analysis to uncover the array access patterns of loops. Since C does array accessing at a very low level, programmers may use many methods in expressing array-manipulating loops. A compiler that tries to derive array access patterns from a C program may recognize some idioms while not recognizing others, with resulting

confusion on the part of programmers as they try to understand how to write code that will optimize well. Finally, most mainstream languages are imperative and have sequential semantics, both of which complicate the compiler's job of finding parallel work.

SA-C, on the other hand, has been narrowly defined to focus on the application area of interest (image processing) and to make the compiler's job more feasible. For example, it has powerful array capability and loops that express array access patterns in a straightforward way. SA-C is a strict, pure functional language, so it eliminates not only explicit pointers but also side effects. The language's strictness combined with its lack of side effects makes it easy for a compiler to find parallelism in a program. Also, the pure functional nature of the language makes source-to-source program transformations easy to design and validate.

This section does not provide a complete description of SA-C, but rather highlights the language aspects that are important for understanding the research that is later described.

2.1 SA-C types

SA-C has scalar base types, complex types, and arrays composed of scalars or complex numbers. There are eight base types in SA-C: **bool** (boolean), **bits<n>** (non-numeric bit vectors), **int<n>** (signed integers), **uint<n>** (unsigned integers), **fix<n.m>** (signed fixed point numbers), **ufix<n.m>** (unsigned fixed point numbers), **float**, and **double**. The n value is a total bit width, and the m value is a fractional bit width. Some example base types are:

```
bool
bits5
uint6
fix16.4
ufix8.8
float
```

The **complex** types in SA-C have signed numeric subtypes **int**, **fix**, **float**, or **double**. Example complex types are:

```
complex int8
complex fix8.8
complex double
```

SA-C has multidimensional arrays with components limited to scalars and complex numbers. The language emphasizes array operations and includes mechanisms for taking array *slices* (or *sections*) and creating arrays through special loop constructs. All arrays are rectangular. The following array-related terms are defined, drawn from Fortran 90 [13] terminology. The *rank* of an array is the number of dimensions it has. Thus, what we commonly call a 2D-array, or matrix, is an array of rank two. An array has an *extent* for each of its dimensions. Multiplying all the extents of an array together yields the number of elements, or the *size* of the array. An array's rank and extents together comprise its *shape*. One or more of an array's extents can be zero, which would imply that the array's size is zero.

An array type in SA-C describes two characteristics of an array: its component type and its rank. The type does *not* define an array's extents. Instead, during program execution each array carries its extents with it, and these extents can be accessed explicitly through the **extents** operator, and implicitly through the loop generators. As in C, SA-C array index ranges always start at zero. An array type specifier consists of a base type followed by a comma-separated list of either colons or integer constants enclosed in square brackets. The colon indicates that the array extent is defined dynamically, whereas an integer constant statically declares the extent. Examples of array types are:

```
int8[:]          // 1D array of 8-bit signed integers
uint4[8,:]       // 2D array (8 rows) of 4-bit unsigned integers
bool[2,2,2]      // 3D "hypercube" of booleans
```

Because an array's extents are not part of its type, arrays of different sizes can have the same type.

2.2 Expressions

Expressions play a dominant role in SA-C programs. The language's integer scalar arithmetic and bit operators are the same as those in C, with the same associativities and precedences. Signed integer and fixed point operations are performed using two's complement arithmetic.

As in C, the inferred type of an expression is derived from the expression's operands, and SA-C uses rules similar in spirit to C, in that the result type is the same as the "widest" or most general

of the types of the operands. Because of SA-C's variety of type and bit widths, the exact rules for this are somewhat more complex than those for C: If either of the operands is signed, the result is signed. An operation on integer or fixed operands yields a result with the maximum integral and fraction operand sizes. Neither the integral, nor the fractional size can exceed 32. If the total size of the result exceeds 32, the size of the fraction will be reduced so as to make the total size 32. As an example, a **fix16.10** combined with a **fix16.14** will result in a **fix20.14**, whereas a **fix32.10** combined with a **fix32.20** will result in a **fix32.10**. An operation on a **float** and an integer or fixed point results in a **float**. An operation on a **double** and a **float** or integer or fixed point results in a **double**.

Even though these rules are similar to those of C, a SA-C programmer must take special care to understand them because of SA-C's variable bit widths. For example, if four **uint8** variables are being added, they will in general require a **uint10** variable to hold the result. Furthermore, a cast will be necessary to force the computation to be done with ten bits:

```
uint10 R = (uint10)V0 + V1 + V2 + V3;
```

Early definitions of SA-C applied similar rules to the intrinsic functions. For example, a call to **sqrt** would return a value having the same type as the call's argument. Users quickly discovered that this did not work well. Consider the situation where a **uint32** value has been computed, and the user wishes to take its square root with a resulting **ufix32.16**, that is to say a value with 16 whole number bits and 16 fractional bits. The user's first attempt might be

```
uint32 v = ...
ufix32.16 vsq = sqrt (v);
```

but this would perform a **uint32**-to-**uint32** square root, which can't hold fractional bits. By the time it is implicitly cast to **ufix32.16** it is too late. If the user attempts to fix the problem by forcing the square root to be done as a **ufix32.16**-to-**ufix32.16** operation

```
uint32 v = ...
ufix32.16 vsq = sqrt ((ufix32.16)v);
```

the cast on v discards the high-order bits of v. Casting it to **ufix48.16** is not an option because SA-C's scalar type bit-widths are limited to 32 bits.

This problem was solved by recognizing that intrinsic functions are *functions*, not operators; the output types of functions are not inferred by their input types. So SA-C now sees **sqrt** as able to accept a variety of input types, but always returning a type **double**. The user can cast that return value to any desired SA-C type, so that

```
uint32 v = ...
ufix32.16 vsq = sqrt (v);
```

does exactly what the user wished. Section 5.4.2.3 discusses how this is handled when the call is transferred into a dataflow graph.

Casting a value of some type to another type, either explicitly or through assignment, can have two effects: the value can be numerically *converted*, i.e. its bit representation can be changed, or the value can be *interpreted* in terms of the new type without change in its bit representation (apart from truncating leftmost bits or adding zero bits on the left to adjust size.) Interpretation occurs when casting any type to and from a **bits** type. Conversion occurs when any non-**bits** is cast to any non-**bits**. Array casts are also allowed in SA-C. For example, the expression

```
(ufix12.4[:,:])A
```

takes the rank-two array A and creates a new array in which each element has been cast to the type **ufix12.4**.

2.3 Array operations

Arrays can be created in a number of ways. First, an array can be explicitly typed, sized by integer constants, and given an expression for each of its elements in an assignment, as in:

```
uint8 A[3,4] =  { {3,6,7,2},
                  {4,3,2,1},
                  {7,1,0,8}}
```

The size of the array must be specified on the left hand side of the assignment, and the patterns within the curly braces must match the size specifiers. Another way of creating an array is through a

loop return value, as discussed later. The **array_concat** operator creates an array by concatenating its two operand arrays, which have to be of equal type and of equal extents in all but the rightmost dimension.

Individual array elements can be referenced using a single set of square brackets, with comma-separated integer expressions. Array slices can be taken using colon notation, similar to that of Fortran 90. A lone colon in a given dimension signifies taking the entire index range of the dimension. A subrange can be specified with integer expressions on either or both sides of the colon, and an optional step as a third parameter. Here are some examples, assuming the above definition of array A:

```
A[1,2]          // returns scalar value '2'
A[:,1]          // returns vector {6,3,1}
A[0,:]          // returns vector {3,6,7,2}
A[2,1:3]        // returns vector {1,0,8}
A[0:1,0:1]      // returns matrix {{3,6},{4,3}}
A[0,2:]         // returns vector {7,2}
A[0,:2:2]       // returns vector {3,7}
```

An array can be referenced through a variety of reduction operators built into the language. For example, **array_sum** takes an array and returns the sum of its elements. It can be given an optional second argument of an array of boolean values that determine which values are summed. The summing operation and the result of the return value are the same type as the array elements, so it is often necessary to cast the array argument to a type suitable for holding the result value. Many of the array reductions also have **accum** versions, in which another array of labels is used to partition the values being reduced; the operator performs a separate reduction and returns a value for each partition.

2.4 Loops in SA-C

SA-C has two loop constructs: **for** and **while**. The **while** loop is the more general, and is designed for situations where, upon encountering the loop, the executing program cannot determine how many iterations will take place. The **for** loop, in contrast, is able to determine, when it is about to begin execution, exactly how many times it will iterate. It is not possible to break out of a **for** loop

early. Both loop forms allow values to be carried from one iteration to the next through the use of "nextified" variables, discussed later. The compiler optimizations and dataflow graph generation in this research concentrate solely on **for** loops, so this document does not deal with **while** loops.

SA-C **for** loops occur in two ways: with and without loop-carried dependencies; i.e. with and without nextified variables. A loop without nextified variables can understood as having completely independent, parallel iterations. The presence of nextified variables, on the other hand, enforces a sequence to the iterations, though the iterations may allow some overlap, and hence parallelism, during execution. A nextified variable in SA-C must be given an initial value before the loop, and a **next** assignment within the loop. For example,

```
uint8 ac = 0;
uint8 rs =
    for .... {
        ...
        next ac = ... ;
        } return (...);
```

A **for** loop has three parts: a *generator*, a *body*, and one or more *return* expressions. The generator produces values that are used in the loop body, and it determines how many iterations the loop will perform. The loop-returns combine, in various ways, values that are produced in the iterations of the loop. When a SA-C loop is executed, it has a *shape*: a rank and extents. This shape is determined by the behavior of the loop's generator, and it determines the shapes of any arrays that are created by the loop's returns. This is described more fully in the following sections.

2.4.1 Loop generators

A SA-C **for** loop has one generator, made up of one or more *simple generators*. Each simple generator is one of four kinds: *scalar*, *array-element*, *array-slice* or *window*. If there are multiple simple generators, they are combined with a *dot* or *cross* operator, described later. Each simple generator has a shape, which contributes to the overall shape of the generator and hence the shape of the loop.

A *scalar* generator produces sequences of integers in one or more dimensions. For example, the generator

```
uint10 v in [3~6]
```

will produce four values (3, 4, 5 and 6), assigned to the variable v. This gives rise to four loop iterations, one for each of the four generated values: the generator is rank-one with an extent of four. Scalar generators can be multidimensional. For example,

```
uint10 v0, uint10 v1 in [1~2,1~3]
```

will produce six pairs of values (<1,1>, <1,2>, <1,3>, <2,1>, <2,2>, <2,3>) assigned to the variables v0 and v1. There are six iterations of the loop: its rank is two, with a shape of [2,3]. Of course, the range expressions in general need not be static constants, allowing the extents of the loop to be determined by the execution of the program. Scalar generators also allow an optional step value for each of the loop's dimensions. Finally, SA-C allows a shorthand syntax for scalar generators that start at zero. For example, these two generators are equivalent:

```
uint8 v in [n]                    uint8 v in [0~n-1]
```

An *array-element* generator extracts scalar values from a source array. Its syntax looks like

```
a in A
```

where a is the capture variable and A is the source array. Here a type is not needed for a since its type is assumed to be the same as the component type of A. This generator will extract all values from A, one per iteration, and give rise to a loop whose shape is identical to that of A. An optional **step** value may be specified for each dimension of the source array. (Non-unit steps means that the generator's extents will be smaller than the extents of the source array.)

An *array-slice* generator extracts sub-arrays from a source array, in which each slicing dimension is the full width of the source array in that dimension. For example,

```
V(~,:) in A
```

extracts row vectors from A into the capture variable V. The pattern inside the parentheses determines which dimensions are being iterated (~) and which are capturing slices (:). In this example, column vectors would be extracted if the two symbols in the parentheses were interchanged. The

rank of an array slice generator is equal to the number of ~ symbols in the parentheses, and the extent for each dimension is the extent of the source array in that dimension. An optional step value can be supplied for each ~ dimension.

A *window* generator extracts sub-arrays from the source array, but unlike the array slice generator, its rank is equal to the source array's rank, and in each dimension the sub-array's extent is typically smaller than the source array's extent. (If it is equal to the source array extent, then the generator has an extent of one. If it is larger, the generator's extent is zero: no iterations are produced.) The window generator specifies the size of the sub-array. For example,

```
window W[3,2] in A
```

will extract 3x2 sub-arrays from source array A into capture variable W. All possible sub-arrays are taken, one per iteration, meaning that the windows overlap each other. The loop extent in a given dimension is the number of windows that fit the source array in that dimension. For example, if A had extents `[10,20]`, then the above window generator would give rise to loop extents of `[8,19]`. (In the vertical dimension, eight size-three windows fit in an extent of ten, and 19 size-two windows fit in an extent of 20.) Window generators may be given optional step sizes in each of their dimensions, which can leave some array elements untouched in either of two ways: the window may skip over elements if its step is larger than its size, and there may be an untouched fringe at the end of the array.

The three array-accessing generators may also capture the indices of the current source array access via an **at** specification. In the case of the window generator, the indices are the lowest indices of the window's elements in each dimension (e.g., the indices of the upper left window element in a rank-two source array).

Any combination of the these simple generators may be formed with **dot** or **cross** operators. The **dot** operator runs the simple generators in lock step, advancing each generator with every loop iteration. The language requires the simple generators in a **dot** to have identical shapes. The generator's shape is the same as the shape of each of its simple generators. A **cross** operator combines simple generators by producing all combinations of values from each of the generators,

varying the rightmost generator the fastest. This is equivalent to a nest of loops, each containing one of the simple generators. (The innermost loop contains the rightmost generator.) The rank of a loop with a **cross** product of simple generators is the sum of the ranks of those generators, and the extents are derived by concatenating the extents of each generator.

In addition to the generators already described, there is a related operator called **loop_indices**. It produces loop iteration indices, and though it seems in that respect to be another generator, it is different because it does not *cause* loop iterations, but merely reports indices of iterations that have been caused by the loop's true generators. Because of this, it does not occur with the generators but rather as a call in the loop body. Here is an example:

```
for window W[3,3] in A step (3,3) cross uint16 v in [m] {
    uint16 i, uint16 j, uint16 k = loop_indices ();
    ...
```

The **loop_indices** operator contrasts with the **at** specification of array-extracting generators in that the **at** values will stride if there is a non-unit step in the generator, whereas the **loop_indices** will *always* produce contiguous values. Also, the **loop_indices** operator's effect is derived from the combined (dotted or crossed) generators, whereas **at** is associated with an individual generator.

There is a significant capability missing in the current SA-C generators: a hybrid between the window and the array-slicing generators. Such a hybrid would behave like a window in one dimension and like a slicing generator in another dimension. Such generator behavior can be expressed less elegantly using existing window generators, but there are two drawbacks: it requires explicitly taking the extents of the source array, and it produces a result array of a rank larger than it should be, making it necessary to take a sub-array to convert it to the desired rank. Here is a simple example of a proposed syntax at left, and the way it must be expressed currently at right:

```
uint8 f (uint8[:,:]);
...
uint8 R[:] =
    for W[:,5] in A
        return (array (f (W)));
```

```
uint8 f (uint8[:,:]);
...
uint16 d0, _ = extents (A);
uint8 RTmp[:,:] =
    for window W[d0,5] in A
        return (array (f (W)));
uint8 R[:] = RTmp[0,:];
```

2.4.2 Loop returns

Because SA-C is a functional language, all loops return one or more values. The language has a variety of return operators, which fall into three kinds: structuring, reduction and final value.

2.4.2.1 Structuring returns

The structuring operators are **array**, **tile** and **concat**. The **array** operator collects scalar values or sub-arrays from the loop iterations and builds them into a return array. If the collected values are scalars, then the shape of the return array is the same as the shape of the loop itself. If the collected values are themselves arrays, then they are required to have identical shape from iteration to iteration, and the return array has a shape formed by concatenating the loop's shape and the component's shape. For example, consider

```
for a in A {
    uint8 T[2,3] = {{a-1,a,a+1},{a-2,a-1,a}};
    } return (array (T))
```

The return component T has a shape of [2,3]. If A has a shape of [5], then the return of the loop has a shape of [5,2,3].

The **tile** operator applies only to array components, and requires that the components have identical shape from iteration to iteration. The return array's rank is the max of the rank of the loop and the component, and its extents are obtained by left-padding the shape expression from the lesser rank with ones and then multiplying them independently in each dimension. For example, a loop with shape [3,4,6] and tiles of shape [2,2] will produce an array of shape [3,8,12]. While this sounds complicated, it's quite easy to visualize. Figure 2.1 shows this example.

The **concat** operator is related to the **tile** operator, but may be used only in a rank-one loop, and unlike tiling, the components that are concatenated may have different rightmost extents from iteration to iteration. (A component's rightmost extent in a given iteration may be zero, meaning that the iteration does not contribute any values to the result array.)

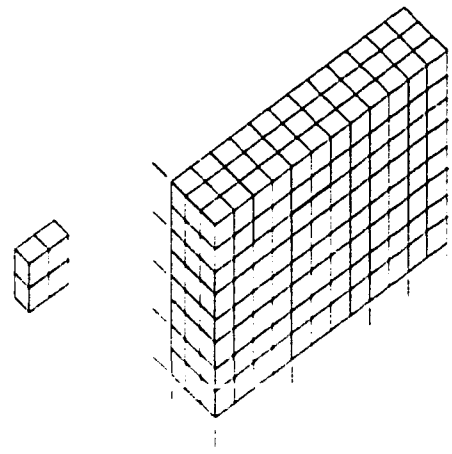


Figure 2.1: Example of **tile** operator on [2,2] tiles (at left) by a [3,4,6] loop.

2.4.2.2 Reduction returns and final values

SA-C loops may perform a variety of reductions, such as **sum**, **product**, **min**, **max**, **and** and **or**. Since these reductions are associative and commutative, they allow parallelism as well as various loop order restructuring. (A notable exception is the **median** reduction, which is not associative: a median-of-medians is not the same as the median.) Reductions also may be given optional boolean mask values that determine for each value whether or not it should be included in the reduction. The **histogram** reduction differs from the above in that it returns an array rather than a scalar. It requires the user to specify an expression for the extent of the return array. All of these reductions also have **accum** counterparts that reduce into arrays. Each value of the return array corresponds to a label value that is used to determine which index the reduction value should be folded into. For histograms, the **accum** version produces a rank-two array.

An interesting family of reductions is the **vals_at_...** operators: **vals_at_maxs**, **vals_at_mins**, **vals_at_first_max**, **vals_at_first_min**, **vals_at_last_max** and **vals_at_last_min**. The first two locate every loop iteration that produced a max (or min) of the specified value, and they return specified capture values from those iterations in the form of an array. The number of maxs (or mins) of a loop, and therefore the size of the return array, is not known until after the loop is run. Sometimes the user wants a value from just one iteration. The other four operators accommodate this by selecting from the first, or last, iteration that produced the min or max.

The last value of a nextified variable may be returned using the **final** keyword, as in this example:

```
for a in A {
    next ac = ac * (a-1);
    } return (final (ac))
```

2.4.3 A loop example

Here is an example of a SA-C loop that runs two dotted windows across two source arrays, computes a value v and a boolean b in each iteration, and returns an array of v and the max value of v from the iterations where b is true:

```
for window WA[3,5] in A dot WB[2,2] in B {
    uint8 v = f (WA) + g (WB);
    bool b = (W[0,0] + W[1,1] - W[1,0] - W[0,1]) >= 0;
    } return (array (v), max (v, b))
```

If the shapes of arrays A and B are [12,16] and [11,13] respectively, then each of the window generators will have a shape of [10,12], meaning that they are consistent. (It is a run-time error for generators of a **dot** operator to have different shapes.) The result array takes its shape from the loop, which in turn has taken its shape from the generator.

2.5 Conditionals and switches

SA-C has the usual conditional **if-else** expression, that can return multiple values. There is also an **elif** that allows expressing a cascade of conditional tests. The return value syntax is similar to that of the loop: a code body may exist within curly braces, followed by the keyword **return** and a parenthesized, comma-separated list of return expressions. The language also has **switch** expressions, which also may return multiple values. Here are examples of a conditional and a switch:

```
if (a>b) {
    uint8 a = x*x+3;
    } return (a)
elif (a<b)
    return (x+1)
else
    return (x-1)
```

```
switch (y+3) {
case 2, 5 : return (y-1)
case 3 : return (y)
default : return (42)
}
```

2.6 Functions

Each SA-C function takes zero or more arguments and returns one or more values. The syntax is derived from that of C. For example.

```
int8, uint4[:,:] f1 (uint8 A[:,:], int8 z, bool M[:]) {
```

is the beginning of function f1's definition. It returns two values: a scalar 8-bit signed integer and a 2D-array of 4-bit unsigned integers. It takes three arguments. A SA-C program consists of one or more function definitions. The function body and return values are syntactically like those of loops: the body is enclosed in curly braces and is followed by the keyword **return** and a parenthesized, comma-separated list of expressions. The function named "main" is considered the top-level function. All function names are in scope throughout the entire program. To make type checking possible and still allow separate compilation, function prototypes can be declared, as in C.

A variety of intrinsic functions are available in SA-C, taken from the standard C math library but adapted slightly: Some of the C math functions return more than one result by passing an additional result via a pointer argument. Since SA-C allows multiple return values, these functions have been implemented with multiple returns. Many of the C math functions take and/or return type **double** values. Since SA-C has a **double** type as well, they have the same type signatures. If a SA-C programmer wants to apply an intrinsic function on a different type, a cast can be used to convert the **double** value to the desired type.

Chapter 3

Data Dependence Control Flow (DDCF) graphs

Data-Dependence-Control-Flow (DDCF) graphs are used as an intermediate representation in the SA-C compiler, suitable for performing a variety of optimizations [18]. 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. The DDCF graph representation has pure functional semantics; a memory model is not exposed at this level. Edges have SA-C data types.

DDCF graphs are not suitable for token-driven execution or simulation because there is implicit information shared between certain kinds of nodes in the graph. For example, there are no explicit edges between a loop's generator and its loop-return nodes. If token-driven simulation were to take place, the return nodes would have to receive information from the generator conveying shape information for any arrays that are produced and therefore would require additional edges to convey this information. But since the DDCF graph is used only for structural representation and transformation, there is no need for these edges; there is an implicit relationship among the generator and the return nodes by virtue of the fact that they exist together in a loop's graph.

3.1 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. Each port of a simple node is associated with a specific piece of information, based on node type. In contrast, a compound node's ports have no particular inherent semantic meaning; the ports are merely means by which to connect outside edges to the internal graph, and those internal connections determine the meaning of the data that flow through a given port. (There is an exception to this: a FUNCTION node's inputs and outputs correspond to its parameters and return values, and the order of the ports correspond to the parameter and return value orders in the SA-C source program.)

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, an FCALL node will hold the name of the function being called.

Compound nodes contain INPUT, INPUT_NEXT and OUTPUT nodes (called I/O nodes) that serve as the interface between the node's internal structure 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" the identities of 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 3.1 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 INPUT_NEXT is distinguished from a INPUT by the fact that there is a dashed implicit edge from a NEXT node back to its associated 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.

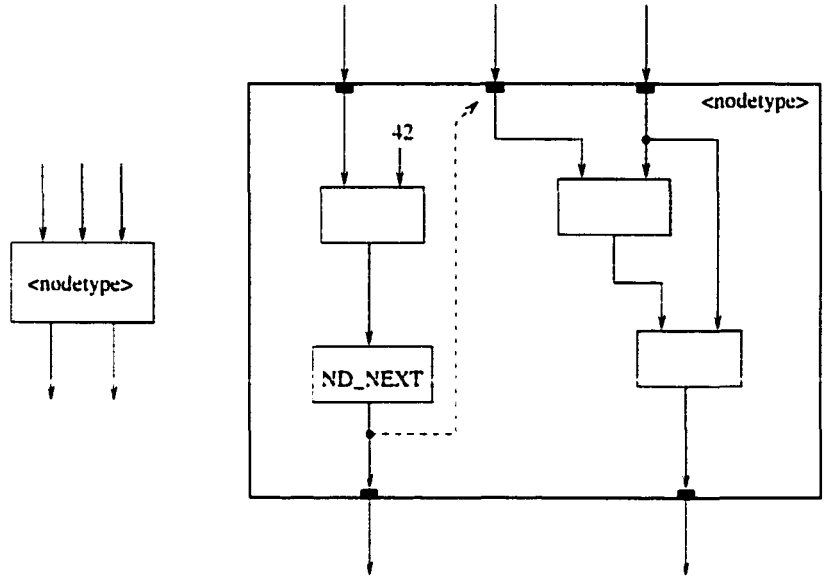


Figure 3.1: An example of a simple node (left) and compound node (right.) Both have three input ports and two output ports. The compound node contains nine internal nodes (including its I/O nodes), and its middle input is "nextified".

Every port has a SA-C *type* tag which tells

- the scalar type (**uint**, **float**, etc.)
- the total bit width
- the fractional bit width
- whether it is scalar or array
- the rank
- the extent, if known, for each dimension

A '-1' value in the extent information indicates that a dimension's size is not known. In general, when an edge connects an output port to an input port, the type tags of the ports must match. There is one exception: when edges from multiple 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 and is therefore not statically known. Since an edge connects

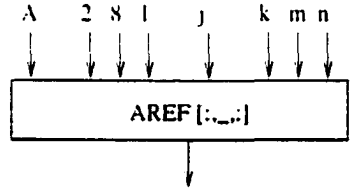
ports with matching types, DDCF figures that show SA-C types are usually drawn with the types alongside the graph's edges.

A complete listing of DDCF nodes is found in the reference document [18]. Nodes that are particularly relevant to this research are now described.

3.1.1 Array-related nodes

A variety of simple nodes are associated with array creation and referencing.

The AREF node is used to extract a scalar or a slice from an input array. It has node-specific information that defines the kind of access it makes: a ':' dimension indicates a sliced dimension, for which three inputs exist to convey the low and high indices of the slice and a step value. A non-colon dimension indicates one for which an index value is provided, as a single input. The first input is the array itself. For example, the SA-C expression A[2:8,j,k:m:n] would produce the node:

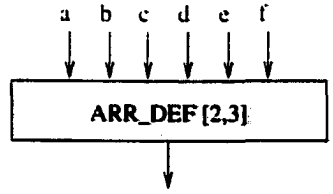


An AREF node that takes a scalar value from an array is simply a special case of this general form, in which all dimensions are designated as '_' and there is one index input for each dimension.

The ARR_DEF node defines an array of statically known size. It has one input for each array value, and node-specific information that specifies the array's shape. For example, the SA-C assignment

```
uint8 A[2,3] = {{a, b, c},{d, e, f}};
```

will produce the node:



An EXTENTS node takes a single array as input and returns the extents of the array; the number of outputs equals the rank of the array. The ARRAY_CONCAT node corresponds to the SA-C operator **array_concat**. It takes two arrays and concatenates them in the rightmost dimension. The source arrays must have the same rank and must have the same extents except for the rightmost dimension. The DDCF graph also has a related node that has no SA-C counterpart: ARRAY_CONCAT_VERT concatenates a pair of rank-two arrays in the left dimension, and is produced during one graph transformation.

3.1.2 DDCF for loop graphs

There are two graph nodes produced by SA-C **for** loops: FORALL and FORNXT. The latter has nextified variables; the former does not. A loop graph has one generator graph node, one or more loop-return nodes and various nodes representing the loop's body.

3.1.2.1 Generator graphs

There are two kinds of generator graph: DOT and CROSS. If a loop has only one simple generator, it is contained in a DOT graph. Within the generator graph are the simple generators, which come in three kinds.

The *scalar* generator of the language produces a SCALARGEN node. For each dimension of the generator there are three inputs and one output. The inputs are the low and high range values, and the step.

The *array-element* and *array-slice* SA-C generators produce ELEGEN nodes. As with AREF nodes, the scalar-extracting version is a special case of the more general form. Node-specific information indicates for each dimension whether it is a slicing dimension (designated by ':') or an iterating dimension (designated by '~'). No input information is needed for a slicing dimension. For each non-slicing dimension, a single input conveys the step size. The leftmost input is the source array. The leftmost output is the array component, and there is an additional output for the **at** index for each non-':' dimension.

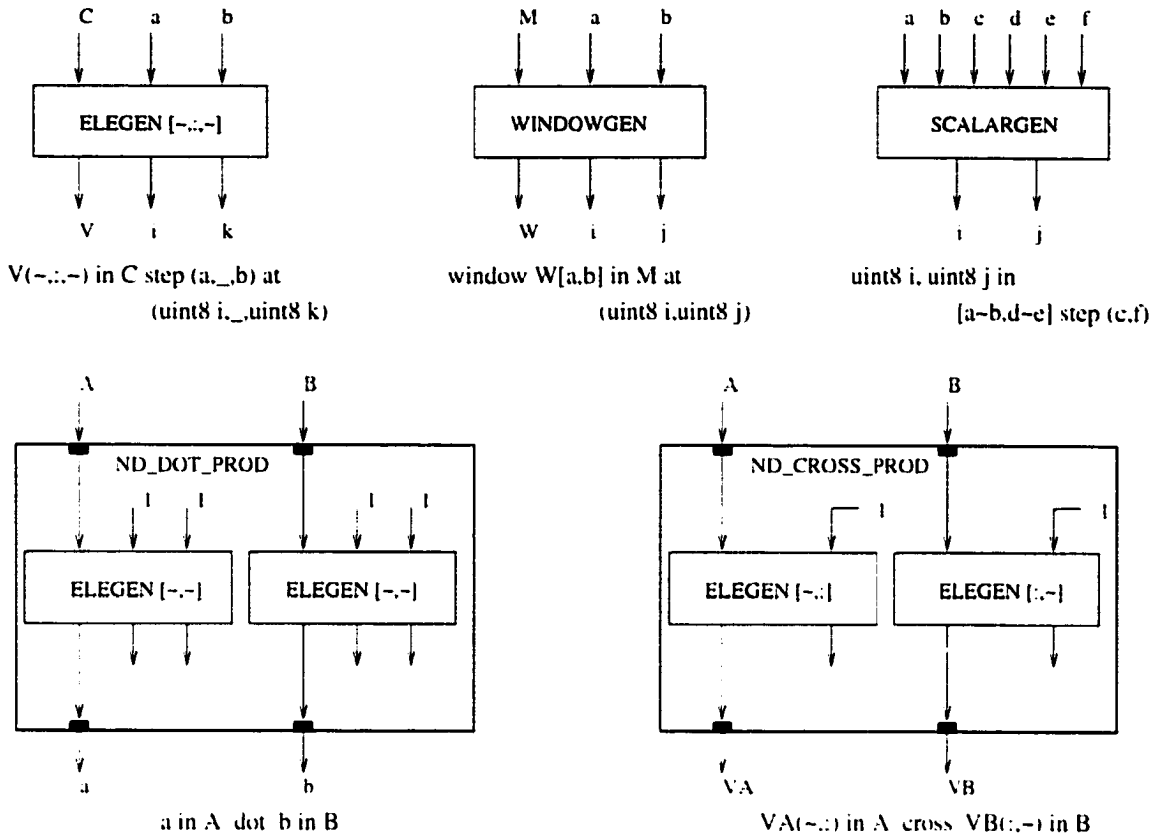


Figure 3.2: Examples of generator nodes.

The *window* generators produce WINDOWGEN nodes. Each dimension of the source array gives rise to two inputs, conveying the window's size and step in that dimension. The leftmost input is the source array. The leftmost output is the generated window sub-array, and there are additional outputs for the **at** index for each dimension.

When multiple simple generators exist in a CROSS graph, they are drawn in a left-to-right configuration that matches the SA-C source. This left-to-right relationship is captured in the DDCF graph internal data structures by the ordering of the CROSS graph's internal node identifier list.

A loop's generator graph is often visualized as sending streams of values out of its output ports. The output edges are typed with the SA-C types of the generator's capture variables in the source program. However, since the DDCF graph is not executed or simulated, this streaming behavior is simply a convenient way of thinking about the meaning of the generator. Figure 3.2 shows an example of each of the three simple generator nodes, as well as cross- and dot-product examples.

3.1.2.2 Loop-return nodes

All of the various loop-return nodes are simple (i.e. not compound) and have direct associations with the language's return nodes. The constructing return operators **array**, **tile** and **concat** give rise to CONSTRUCT_ARRAY, CONSTRUCT_TILE and CONSTRUCT_CONCAT nodes. Each has a single input (the component) and a single output (the accumulated result). When compared with the simple generators, these nodes have a kind of inverse behavior: the array-referencing generators take arrays as inputs and emit array components as outputs, whereas the constructing return nodes take components and emit complete arrays as outputs.

The SA-C loop reduction operators have direct counterparts as well, giving rise to nodes such as SUM, AND, MAX and HISTOGRAM. Each produces one output, and most have two inputs, one for the value and the other for the boolean mask value. (HISTOGRAM, however has an additional input for the size of the result array.) The reductions also have their **accum** counterparts, such as ACCUM_SUM, ACCUM_AND, ACCUM_MAX, ACCUM_HISTOGRAM, etc. Each has an additional input to indicate the range of the label values, specifying the extent of the resulting array.

If a **final** value of a nextified variable is returned, it goes directly from the output of its associated NEXT node to an output port of the loop's graph.

3.1.2.3 Loop example

Figure 3.3 shows an example of a parallel **for** loop graph corresponding to the SA-C loop:

```
for V(:,~) in M at (_, uint8 idx) {
    uint8 s = array_sum (V) + idx;
    } return (array (s))
```

The inner loop is produced by the **array_sum** call.

3.1.2.4 Nodes derived from unrolled loops

The SA-C compiler does full loop unrolling when it is able to determine how many iterations the loop will execute. Loop unrolling requires new nodes that are associated with loop reductions after unrolling. These nodes are xxx-MANY versions of the reductions. For example, consider the following loop:

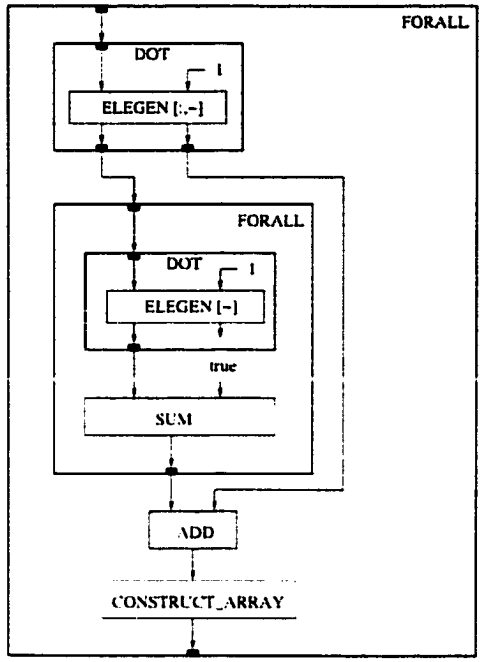
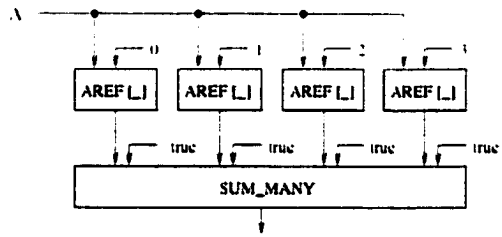


Figure 3.3: Example of a FORALL loop.

```
for a in A
    return (sum (a))
```

If A has a shape of [4] and the loop is unrolled, it will produce a SUM_MANY node as shown here:



There is a pair of inputs for each iteration, one for the value and one for the optional boolean mask.

3.1.3 Switch graphs

The SWITCH node is a compound node corresponding to switch and conditional expressions in SA-C. Each SWITCH graph contains exactly one SWITCH_KEY node that acts as a sink for the switch select expression. The SWITCH graph also contains multiple CASE graphs, each of which in turn contains a SELECTORS node that sinks multiple constant value inputs. (There is an exception: one CASE graph may lack a SELECTORS node: this is a default CASE.) SA-C conditional expressions

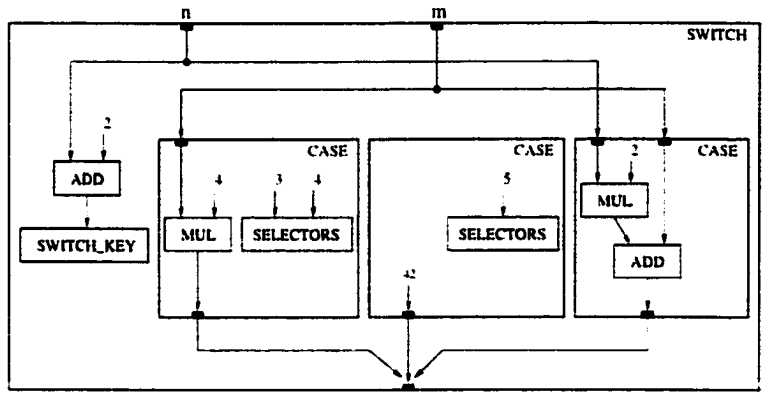


Figure 3.4: Example of a switch expression's DDCF graph.

(**if-else**) are represented by SWITCH graphs since the conditional is a special case of the more general switch. The OUTPUT nodes of a SWITCH graph are the single exception to the rule that multiple DDCF graph edges may not target the same port.

As an example, figure 3.4 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) }
```

Chapter 4

SA-C compiler optimizations

The optimizations in the SA-C compiler are designed and implemented with the goal of improving performance of the FPGA-related code. While the host code can also benefit from optimization, no effort has been expended in that direction yet. (However, some of the optimizations that have been implemented may incidentally improve host code as well.) Optimization refers to transformation of code to improve performance, either in space or time.

In conventional architectures, space usually refers to memory space, but in FPGA-related systems space can also refer to the amount of internal FPGA logic space that a loop requires in its implementation. Execution time in an architecture with FGPA-based coprocessor boards can be broken into a number of sometimes overlapping categories:

- **Configuration download time** is the time taken to transfer the bitmaps that define the FPGAs' internal logic. This is somewhat analogous to loading a program into memory on a conventional computer.
- **Data transfer to RCS** refers to the time taken to copy data from the host to the memories of the RCS board.
- **RCS compute time** is the time spent by the RCS doing the computation. It has a number of parts, which may overlap:
  - **RCS read from local memory** is part of the RCS compute time. It is the time spent reading values from RCS local memory into the FPGA(s). Because windows in SA-C's

loop generators often overlap each other, the number of reads may change after a code transformation.

  - **Loop iteration overhead** refers to the amount of time required by a loop iteration, and is part of the RCS compute time. Because the number of loop iterations may change when a code is transformed, this time can change.

  - **RCS write to local memory** is part of the RCS compute time. It is the time spent writing values to RCS local memory from the FPGA(s). Unlike the **RCS read from local memory** where generator overlap reduction can change the number of reads, the number of writes for a given loop and its transformation must always be the same.

- **Data transfer from RCS** refers to the time taken to transfer results from the RCS board back to the host.

Optimizations are designed to improve performance in one or more of the above categories, but often a performance improvement in one area produces a degradation in another area, leading to performance tradeoffs. Also, some costs can be hidden by concurrency. The following descriptions of SA-C compiler optimizations include brief motivations, but a more complete performance analysis is saved for the later chapter on performance measurement on benchmarks. All the optimizations take place as DDCF-to-DDCF transformations, meaning that they stay within the pure functional realm and do not expose most implementation details, including memory use.

In the next section, the currently-implemented SA-C compiler optimizations are described. Following that is a section describing some optimizations that are not implemented, but which can nevertheless be tested since they are expressible in SA-C. The last section talks about the order in which the optimizations are performed and iterated.

4.1 Currently implemented optimizations

The optimizations described in this section are implemented in the SA-C 1.0.242 compiler.

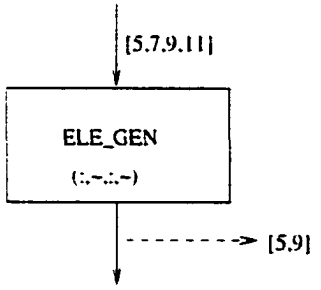
4.1.1 Array and loop size propagation

When compiling SA-C to FPGAs, static size information is vitally important. For example, full loop unrolling can take place if the number of loop iterations is statically known, allowing the loop to spread over chip space rather than over time. Section 4.1.2 discusses the benefits of full loop unrolling.

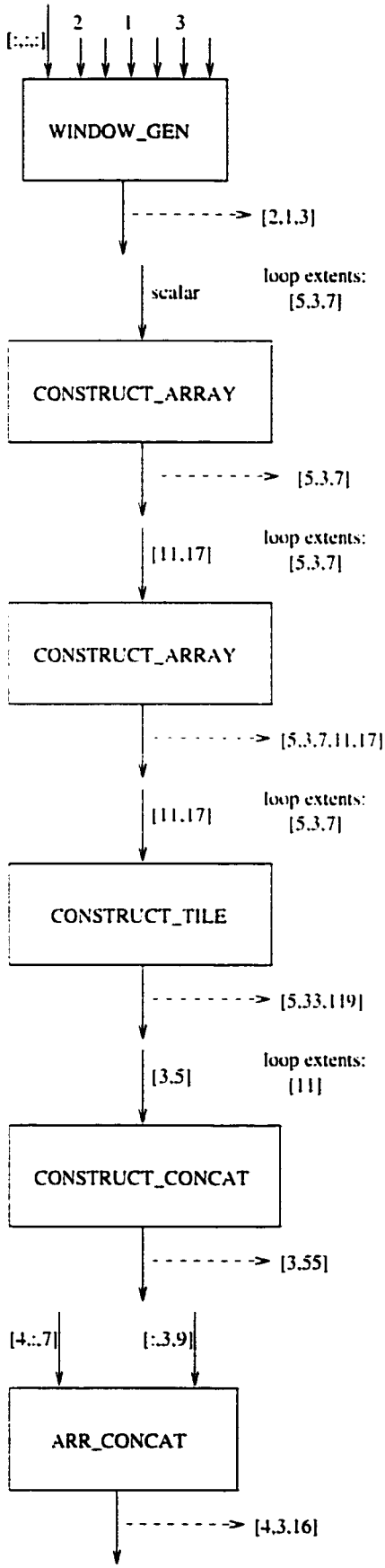
SA-C has a close association between arrays and loops. Both have *shapes*, i.e. ranks and extents. When a loop generator traverses an array, that array's size plays a part in determining the shape of the loop. The loop's shape, in turn, plays a part in determining the shapes of the arrays it creates. This means that extent information at one place in the program can propagate to parts of the program that are associated with it. If viewed from the perspective of the DDCF graph, this information can flow downwards, upwards and laterally among the individual generators within a DOT graph. The lateral information flow takes place through the shape information of the loop itself (see section 2.4.1). The size propagation implementation has two parts: a downward sweep puts array extent information on the outputs of nodes, whereas an upward sweep puts array extent information on the inputs of nodes. During both sweeps information is transferred to and from the loops' extents. These sweeps are iterated until no new information is inferred.

4.1.1.1 Inferring array sizes on node outputs

This section defines the rules used to infer array extents at the output ports of various node types, using input extent information and/or loop extents. The dashed arrows show the extents that are computed.



The ':' extents are passed through to the output's extents. (Note that step specifications do not exist for ':' dimensions.) Loop extents are not relevant here.



The output extents are not related to the input array's extents, but rather come from the window-size inputs to the node. Step sizes do not matter. Loop extents are not relevant here.

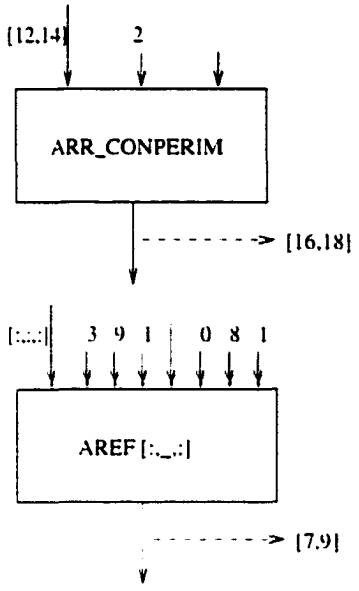
For scalar input, the output extents are the same as the loop's extents.

For non-scalar input, the output extents are the loop extents concatenated with the input extents.

The loop extents and the input extents are right justified, and the shorter one is left-padded with ones. The output extent in each dimension is the product of the loop extent and the input extent.

The loop rank is guaranteed to be one. The rightmost output extent is the product of the input extent and the loop extent. The other extents are carried down directly from the input extents.

The rightmost output extent is the sum of the rightmost extents of the inputs. Each remaining output extent can be carried down directly from either of the corresponding input extents. An error can be reported here if the input extents are not consistent.



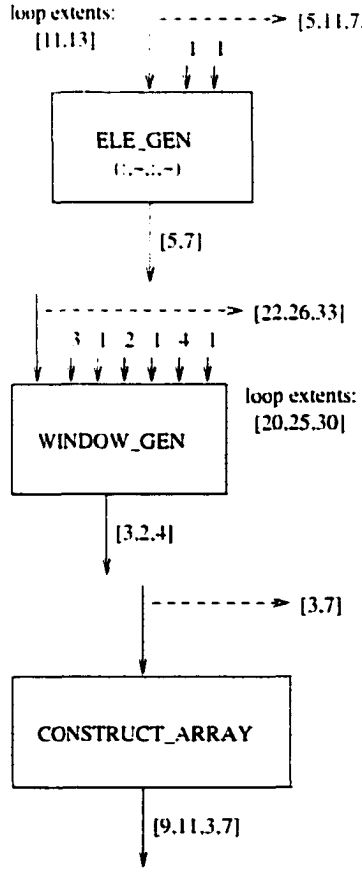
Each output extent is the sum of the corresponding input extent and two times the perimeter width.

The input extents are not relevant. The ':' dimensions are slice dimensions, whereas each '_' dimension has a specified index expression. Each output dimension corresponds to a ':' in the extract pattern. Note that for each ':' dimension, there are three input edges, carrying the low index, high index and step size. An output extent is computed by the C expression:

```
hi-lo<0?0:1+(hi-lo)/step
```

4.1.1.2 Inferring array sizes on node inputs

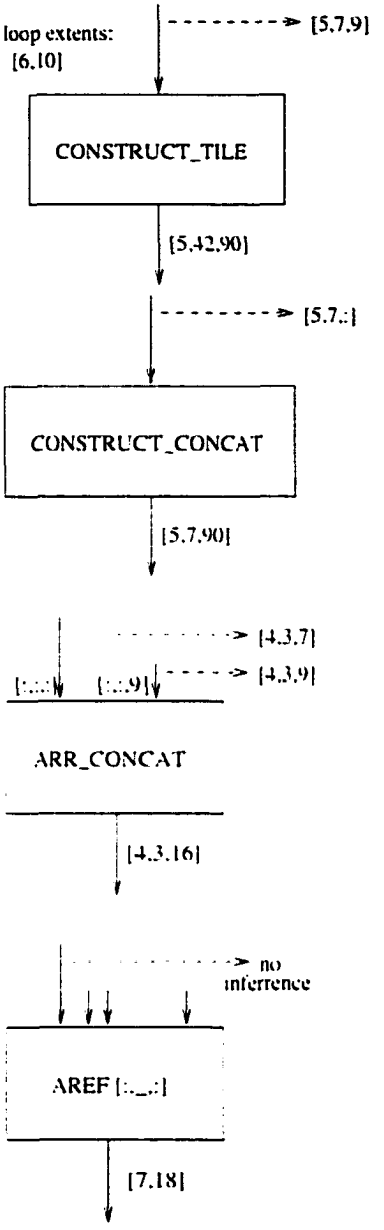
This section defines the rules used to infer array extents at the input ports of various node types, using output extent information and/or loop extents. The dashed arrows show the extents that are computed.



The input extent for a ':' dimension comes from the corresponding output extent. The input extent for a '~' dimension comes from the corresponding loop extent, and can take place only if the step size is one. Non-unit step sizes don't allow size inference because a leftover "fringe" could be present in the generator's source array.

An input extent can be inferred only if its loop extent is greater than zero and the step size is one. The input extent is the loop extent plus the window size minus one.

The input rank has been determined by the type system. Its rank is guaranteed to be less than the rank of the output. The input extents are taken from the rightmost output extents.



The loop extents and output extents are right justified. Each input extent is computed by dividing the output extent by the loop extent. An error can be reported here if the output extent *mod* loop extent is not zero. For input dimensions where only one extent exists (either loop extent or output extent), the input extent is set to that extent.

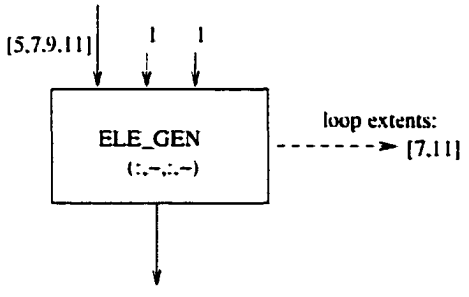
The loop rank is guaranteed to be one. The rightmost extent of the input can never be inferred, since its size may vary from iteration to iteration of the loop. The remaining extents can be inferred directly from the output extents.

All but the rightmost extents of both inputs can be inferred directly from the output extents. The sum of the rightmost input extents must equal the rightmost output extent, so if the output and one input extent is known, the other can be inferred. An error can be reported here if the input extents are inconsistent with each other, or with the output. An error can be reported if the rightmost output extent and one input extent infer a negative extent for the other input.

No input extent information can be directly inferred. This can be understood by looking at the downward propagation rule, shown earlier. The output extents are associated with the slice size, not the source array's size. Even in the case where the user has specified a full slice, e.g. `A[:,3]`, the fact that a full slice is being taken in a given dimension is not directly evident in the DDCF graph.

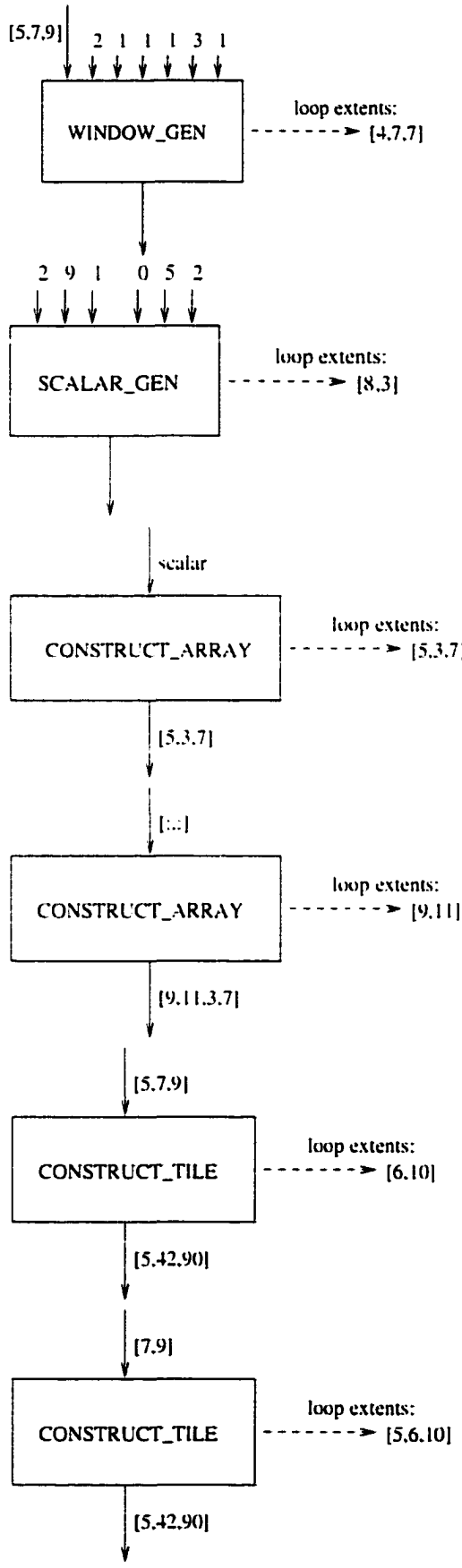
4.1.1.3 Inferring loop extents

This section defines the rules used to infer loop extents from the input and/or output extents of various node types. The dashed arrows show the extents that are computed.



For a '~' dimension, the loop extent can be inferred directly from the input extent and the step size. The loop extent is computed by the C expression

```
in<0?0:1+in/step
```



Each loop extent can be inferred from the input extent, the window size and the window step. The extent is computed by the C expression

```
in-wsz<0?0:1+(in-wsz)/step
```

Each loop extent can be inferred from the lo, hi and step values. The extent is computed by the C expression

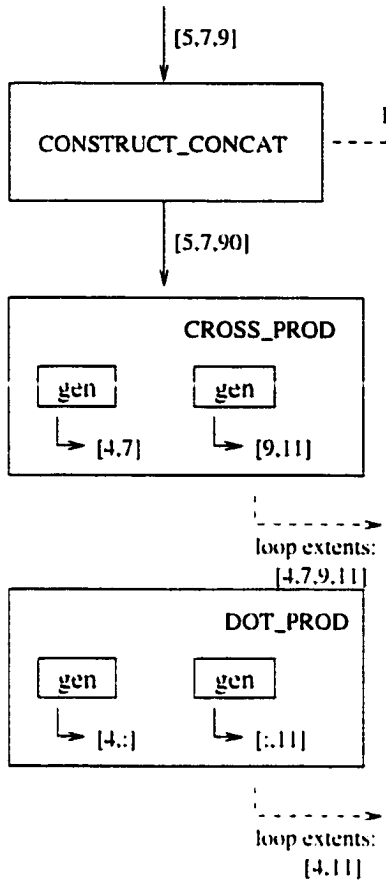
```
hi-lo<0?0:1+(hi-lo)/step
```

When the input is scalar, the loop extents can be inferred directly from the output extents.

When the input is an array, the loop extents can be inferred directly from the leftmost output extents. Note that the output rank equals the input rank plus the loop rank, and both ranks have been determined by the type system.

If the loop rank is less than or equal to the output rank, the loop, output and input extents are right justified and each loop extent can be computed by dividing the output extent by the input extent. If the output extent *mod* input extent is not zero, an error is reported.

If the loop rank is greater than the input rank, the loop, output and input extents are right justified and each loop extent can be computed by dividing the output extent by the input extent. If the output extent *mod* input extent is not zero, an error is reported. The leftmost loop extents are inferred directly from the output extents.



The loop rank is guaranteed to be one. The loop extent can be computed by dividing the output extent by the input extent. If the output extent *mod* input extent is not zero, an error is reported.

The loop extents produced by each simple generator in a cross product are concatenated to produce the complete set of loop extents.

For dot products, a loop extent can be inferred from the corresponding extent of any of its simple generators. Errors can be reported of the extents of the simple generators are not consistent.

4.1.1.4 An example

Figure 4.1 shows a small SA-C program (encountered while restructuring a 7x7 convolution) that demonstrates many of the propagation rules. Note that size propagation does not take place across function calls, but rather occurs after functions have been inlined. However, for clarity, this example is shown before inlining. The size "7" in this code originates from the user's specification in the window generator of the main function. It propagates downward through other generators and slices, until it reaches the **dot** operator in the dprod function. Here it crosses to the other array in the **dot** and it travels all the way up, finally inferring that the left dimension of the kernel parameter of main must be size "7".

4.1.2 Full loop unrolling

Full unrolling of a loop means eliminating the loop entirely and replacing it with explicit loop bodies, each corresponding to one iteration of the loop [42]. This can be done only if the compiler

```
uint22[:,:] main (uint8 Image[:,:], uint8 kernel[:,:]) {
  _, uint16 d1 = extents (Image);

  uint22 res[:,:,:] =
    for window W[7,d1] in Image {
      uint22 res_row[:] = process_stripe (W, kernel);
      } return (array (res_row));
    } return (res[:,0,:]);

uint22[:] process_stripe (uint8 W[:,:], uint8 kernel[:,:]) {
  _, uint16 d1 = extents (W);

  uint22 intermediate[:,:] =
    for kernel_col(:,~) in kernel at (_, uint8 c) {
      uint8 slice[:,:] = W[:,c:d1+c-7];
      uint22 rw[:] =
        for win_col(:,~) in slice
          return (array (dprod (kernel_col, win_col)));
      } return (array (rw));

  uint22 res[:] =
    for V(:,~) in intermediate
      return (array (array_sum (V)));
  } return (res);

uint22 dprod (uint8 V0[:], uint8 V1[:]) {
  uint22 res =
    for v0 in V0  dot  v1 in V1
      return (sum ((uint22)v0*v1));
  } return (res);
```

Figure 4.1: An example of array and loop size propagation.

can determine the exact number of iterations and the sources of the inputs that will drive them. This optimization in the SA-C compiler has a number of motivations. First, since the SA-C compiler currently places only a bottom-level loop (i.e. a loop that contains no loops) on the RCS, unrolling an inner loop may allow a higher-level loop containing it to be placed on the RCS. This in turn means that more work per iteration may be performed by the RCS, better amortizing the cost of moving data between RCS and host. Also, since loop unrolling distributes the loop's iterations over space (i.e. FPGA logic blocks) instead of time, the computation may be speeded up. In SA-C, a loop often can be completely unrolled if its extents, inferred by the size propagation pass, are known by the compiler. There are some generator and return-operator limitations on unrolling, which are noted in the following discussion.

4.1.2.1 Unrolling loop generators

SA-C has four kinds of loop generators, which can be combined as **dot** or **cross** products: scalar, array-element, array-slice and window. It also has a loop-extents generator that produces the loop's indices based on the other generators and the way they are combined. Each of these generators must be handled appropriately when a loop is unrolled.

To unroll a scalar generator, the compiler must produce constant values corresponding to the scalar values the generator produces as inputs to the loop bodies. A loop-extents generator is unrolled in a similar way, but the scalar values start with zero and are unit-stride. Figure 4.2 shows an example of each.

Array-element, array-slice and window generators require AREF nodes to extract appropriate parts of the source array. An array-slice generator also requires an EXTENTS node to determine the extent(s) of the source array. Figure 4.3 shows examples of these unrolled generators.

The current implementation of loop unrolling requires that the generator parameters be constants. This restriction could be eased: the compiler would have to build expression sub-graphs where it currently computes and installs constants for inputs to nodes.

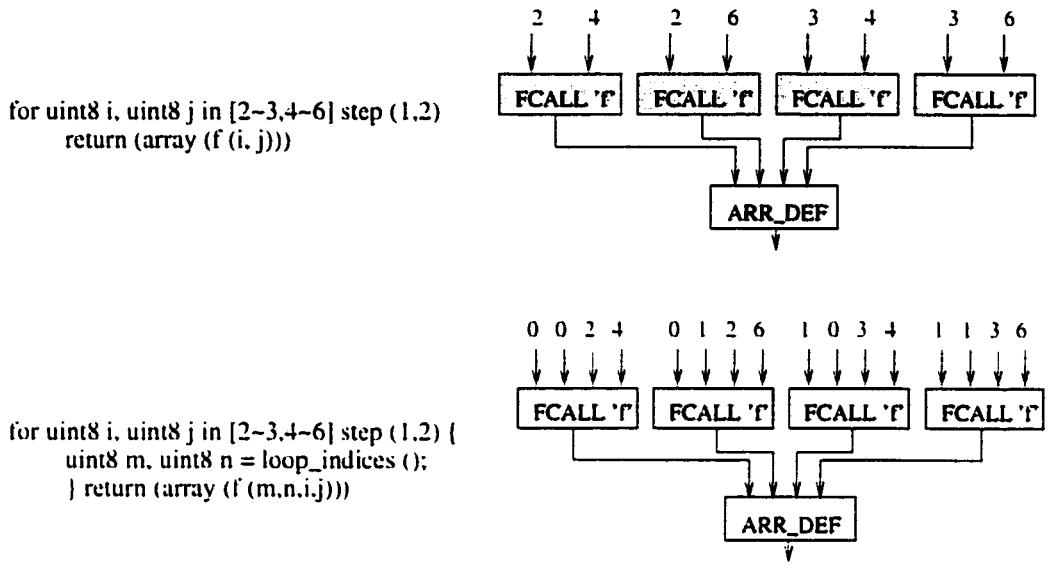


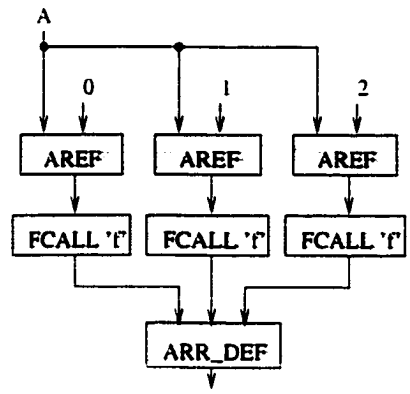
Figure 4.2: Examples of unrolled scalar generator and loop-indices generator.

4.1.2.2 Unrolling loop return operators

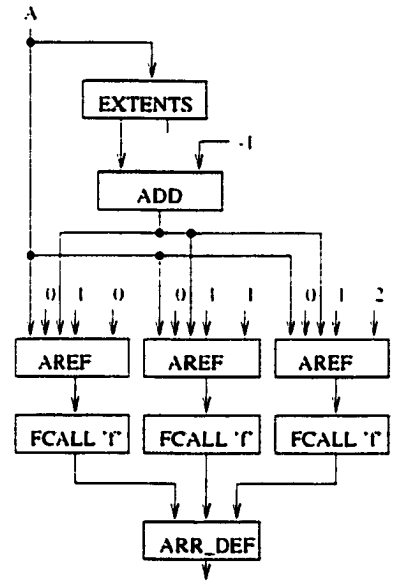
Most loop return operators allow a loop to be unrolled, with each operator having an associated node type that takes in explicit value edges rather than the loop's implicit stream of values. The number of inputs is variable. As an example, a SUM loop return operator, which has two inputs for the value and boolean mask streams, will produce a SUM_MANY node with two inputs (value and mask) for each loop body that is created. It is left to implementations to choose an appropriate evaluation strategy for the node (e.g., tree of adders.) Table 4.1 lists the loop return operators that allow loop unrolling, and their unrolled counterparts.

Since the goal of full unrolling is to generate DFGs for placement on the RCS, some loop return operators have not been supported for unrolling due to the fact that their unrolled counterparts are not implementable in FPGAs under the current compilation process. For example, the various **accum** versions of the reductions produce arrays as outputs. In general, array-producing nodes cannot exist in a loop body on the RCS (see chapter 5 for a full discussion of the translation to DFGs), so the "MANY" counterparts of the **accum** return operators have not been implemented. The **mean** and **st_dev** operators have not been implemented since they require division, which is currently not supported on the RCS.

```
for a in A
    return (array (f (a)))
```




```
for V(:,-) in A
    return (array (f (V)))
```




```
for window W[3] in A
    return (array (f (W)))
```

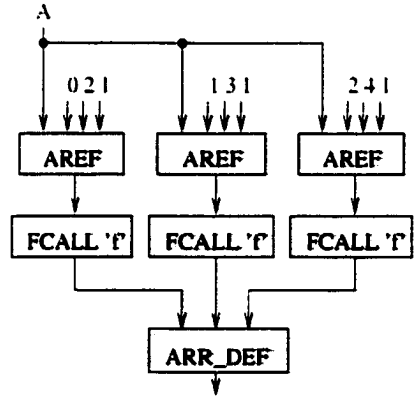


Figure 4.3: Examples of unrolled array-element generator, array-slice generator, and window generator.

loop return operator	unrolled counterpart
VAL_AT_FIRST_MAX	VAL_AT_FIRST_MAX_MANY
VAL_AT_FIRST_MIN	VAL_AT_FIRST_MIN_MANY
VAL_AT_LAST_MAX	VAL_AT_LAST_MAX_MANY
VAL_AT_LAST_MIN	VAL_AT_LAST_MIN_MANY
VAL_AT_MAXS	VAL_AT_MAXS_MANY
VAL_AT_MINS	VAL_AT_MINS_MANY
SUM	SUM_MANY
HIST	HIST_MANY
MIN	MIN_MANY
MAX	MAX_MANY
AND	AMANY
OR	OR_MANY
MEDIAN	MEDIAN_MANY
CONSTRUCT_ARRAY	ARR_DEF

Table 4.1: Loop return node types capable of unrolling, with their unrolled counterparts.

4.1.3 Stripmining and blocking

The window generators in SA-C allow loop stripmining and array blocking [42] to be expressed in an elegant way. These two optimizations are pragma-directed and they have nearly identical implementations.

Stripmining is used, along with full inner loop unrolling, to achieve the effect of partial unrolling over multiple dimensions. This should have a number of useful effects. First, when overlapping windows are being generated, this should reduce the amount of data moved from the local memories to the FPGAs. Second, the number of loop iterations is reduced, thus reducing loop overhead. Finally, stripmining allows multiple loop bodies to be executed concurrently on an FPGA.

On the other hand, *blocking* is motivated by size limitations of the RCS local memories. Since a source array for a loop may be too big to fit in the local memory, blocking breaks the loop into a number of chunks, each with a sub-array that will fit the local memory. The partial results from these chunks are then combined by the host into a complete result.

Loop stripmining, in multiple dimensions, is used to get the effect of partial unrolling of loops. For example, consider

```
// PRAGMA (stripmine (4,5))
for window W[3,3] in A
```

A new loop will be created, with a 4x5 window generator. The existing loop will be nested inside the new loop, and will traverse the 4x5 arrays created by the outer loop. The inner loop is then fully unrolled, leaving one loop that has six parallel loop bodies. As with any partial loop unrolling, this has the potential for shared computation among the loop bodies, saving work. It also may save loop iteration overhead by reducing the number of iterations.

While the motivations of array blocking are different than those of stripmining, it is accomplished in an almost identical way. The difference is that the original loop is not unrolled. For example, consider

```
// PRAGMA (block (80,80))
for window W[3,3] in A
```

As with stripmining, a loop nest is created by enclosing the original loop in a new loop with, in this case, an 80x80 window generator, but the inner loop is *not* unrolled. Since only an inner loop can run on FPGAs, the inner loop with the 3x3 generator will run on the RCS, and the outer loop will run on the host. The hosts's loop blocks the source array into chunks; each chunk is an argument to the inner loop that is called on the RCS. Since a user may want both stripmining and array blocking, it is possible to put both pragmas on a loop. The stripmine transformation is applied first, yielding one loop that replaces the original. Then blocking is applied, creating a loop nest.

Implementation of these transformations requires dealing with three issues. First, the step sizes for the newly formed outer loop must be determined. Second, the values returned by the inner loop must be combined by the new outer loop into final values for the loop nest. Finally, since the window of the outer loop may leave "fringes" of the source array untouched, these fringes must be handled separately and their results grafted onto the array returned by the main loop. In the implementation discussion, the following variables are used, subscripted by the dimension being referenced:

- v_i is the original window's extent in dimension i
- s_i is the original window's step in dimension i
- a_i is the stripmine window's extent in dimension i

- t_i is the stripmine window's step in dimension i
- e_i is the source array's extent in dimension i

4.1.3.1 Computing the new step size and the fringe array slices

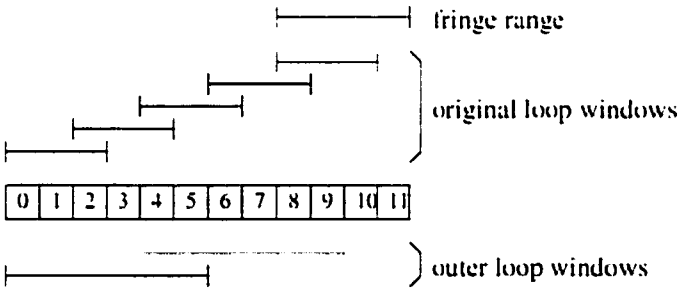
The outer loop's step is computed by the equation

$$t_i = ((a_i - v_i + s_i)/s_i) \times s_i$$

using integer arithmetic. This equation is designed to ensure that, in the event of non-unit step in the original loop, each window of the outer loop starts at an index that would be visited by the original unstripmined loop. For example, the stripmined loop

```
// PRAGMA (stripmine (6))
for window W[3] in A step (2)
```

on a source array of extent 12 produces an outer loop step of four. This can be shown visually:

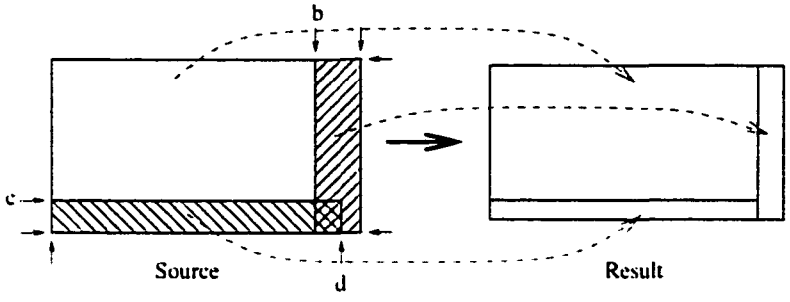


Each outer-loop window (below) starts on an index that an original iteration started on, and each succeeding outer iteration starts where the preceding one failed to contain an original iteration. This diagram also shows that the sub-array used to compute the fringe must start at the first original-loop window that fails to fit in the outer-loop windows. This starting index is computed from the expression

$$(e_i - (((e_i - a_i) \bmod t_i) + a_i)) + t_i$$

which evaluates to eight in this example.

The fringes are more complicated for a rank-two source array. A right sub-array and a bottom sub-array are created to compute the fringes, as shown:



The right source array's horizontal start index is computed by the equation:

$$b = (e_1 - (((e_1 - a_1) \bmod t_1) + a_1)) + t_1$$

Its vertical range is identical to the vertical range of the source array, making the array slice:

$$A[0 : e_0 - 1, b : e_1 - 1]$$

The bottom source array's slice requires computing the value of a start index c and an end index d:

$$c = (e_0 - (((e_0 - a_0) \bmod t_0) + a_0)) + t_0$$

$$d = b + (e_1 - s_1 - 1)$$

Its array slice is:

$$A[c : e_0 - 1, 0 : d]$$

The values of a_i are always statically known since they are specified by the pragma. The values of e_i and s_i need not be statically known, but in the present implementation they are required to be. This allows the step sizes of the outer loop to be computed by the compiler. To ease this restriction, the compiler would have to build a small expression graph that would compute the step size dynamically using the above equations.

4.1.3.2 Combining the inner loop results

Since this transformation computes chunks of results produced by the inner loop, the outer loop must be able to combine these chunks into a correct result. Stripmining has been implemented for those operators that are associative in the following sense:

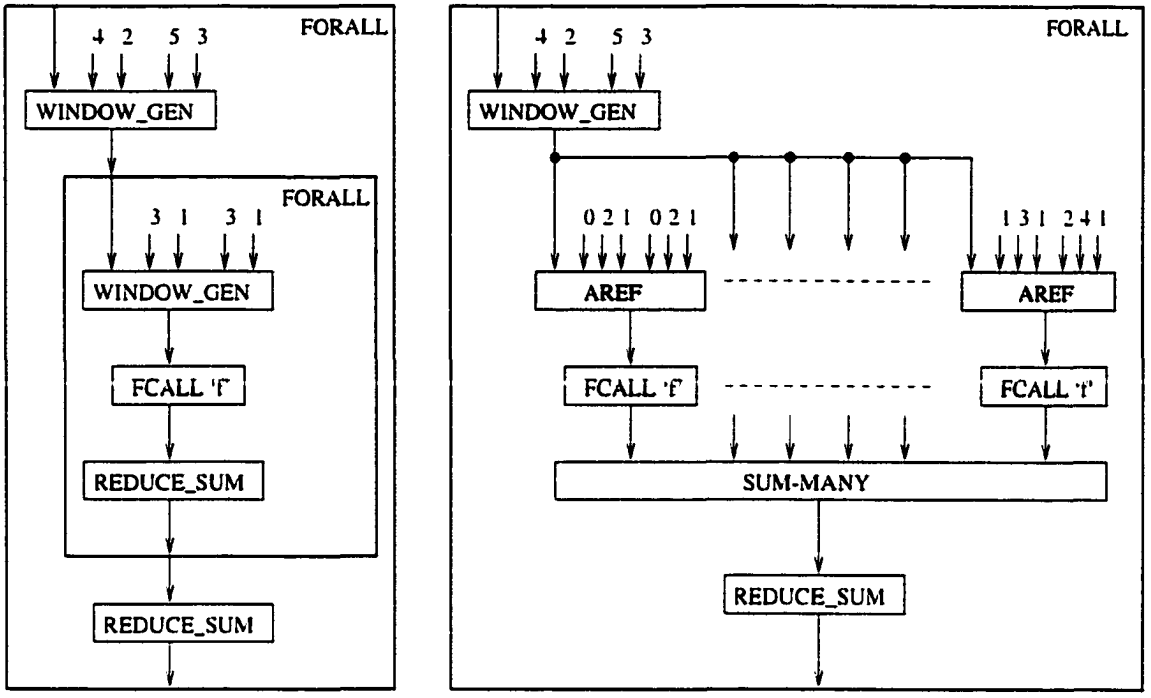
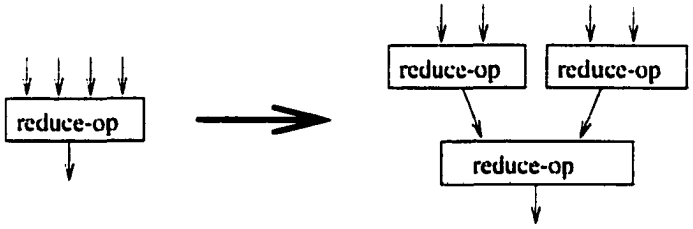


Figure 4.4: Stripmined window generator loop, before and after inner loop unrolling.



The presence of mask values can make some operators, such as **mean**, nonassociative since the number of values being averaged in the various subcomputations may not be equal. (With some significant additional work, **mean** could be accommodated by bringing a second value out of the inner loop to convey the number of values that were used in the mean. However, it would be better to use SUM reductions in the inner loop, and do only one division after all results are combined.)

Figure 4.4 shows DDCF graphs, before and after inner loop unrolling, for the following loop:

```
// PRAGMA (stripmine (4,5))
for window W[3,3] in A
    return (sum (f (W)))
```

Before unrolling, the outer loop is summing a stream of values, each of which was produced by the inner loop's summing; in this case the inner loop is summing six values. After unrolling, the inner loop produces six loop bodies, each fed by an `AREF` node that takes a 3x3 slice of the 4x5 outer loop window. As discussed in section 4.1.2, the sum reduction node is replaced by its `SUM-MANY` counterpart.

A variety of loop return node types will prevent this transformation from taking place. The **mean** and **st_dev** could be handled with additional work as mentioned earlier. The **mode** and **median** operators are not associative and therefore will not allow stripmining or blocking. The **histogram** operator and many of the various **accum** operators are associative but return sub-arrays, which would require the creation of new DDCF nodes to combine the outputs of the inner loop. For example, **histogram** would require a node that summed arrays on an index-by-index basis. The **vals_at_first_min** operator (and its related nodes) could have been stripmined if the operator had returned the min along with the captured values, but since only the captured values emerge from the inner loop, a combining node for the outer loop does not have the information needed to select among the values it receives.

4.1.3.3 Other restrictions

There are a few other restrictions on this transformation:

- The current implementation deals only with rank-one and -two generators. This can be extended to higher ranks, but has been deferred since the computation of the fringes gets more and more complex as the rank increases.
- Generators with "at" specifications currently are not able to be transformed. To do this properly, the inner loop would have to be given a scalar generator, combined with its window generator as a dot product, that would produce the range of indices appropriate for the particular outer loop iteration.
- If any of the pragma's extents is less than its corresponding window generator extent, the transformation will not take place.

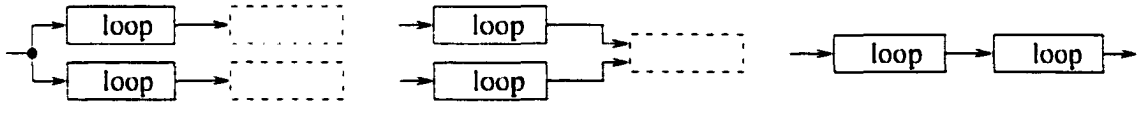
- The current implementation allows only one generator in a loop that is to be stripmined, because with more than one generator it is not clear to which generator the stripmine pragma's extents should be applied. (Keep in mind that a pair of generators may apply different window sizes with different steps.) However, since in a vast majority of cases the generators in a DOT operation have identical sizes and steps, this restriction could be eased to allow multiple generators in that case.

- The current implementation cannot stripmine a loop that has an array-slice generator, since doing so would require the hybrid window/slice generator as discussed in section 2.4.1.

- The current implementation does not operate on loops with nextified variables. In general a nextified loop cannot be stripmined, since there are dependencies between successive loop iterations and vertical stripmining, for example, has the effect of rescheduling the iterations. Stripmining in the rightmost dimension, however, could be done.

4.1.4 Loop fusion

There are various ways to fuse loops [42]. Three possibilities are associated with these structural forms:



In the form on the left, if the input array is traversed by identical generators in both loops, they can be fused easily. In the middle form, the two inputs arrays in some cases may be inferred to have the same shape (by analysis of the generators they feed) and it may be possible to fuse the two array-producing loops. The first two forms are not difficult for a programmer to fuse, and the present SA-C compiler does not fuse them. (However, in a programming environment where these loops come from a predefined component library it would be useful for the compiler to fuse these forms.)

Fusion in the form on the right (a producer/consumer form) can be expressed in SA-C under many conditions, but it is not easy for the programmer to do, as will become apparent in the discus-

sion below, so this kind of fusion has been implemented in the SA-C compiler. This transformation is intended to reduce communication, both between the host and RCS, and perhaps more importantly between the FPGAs and their local memories. Fusion should also reduce the number of loop iterations, reducing loop overhead.

4.1.4.1 Development of an approach to fusion

The goal of loop fusion is to replace a pair of loops with one new loop. This is partially accomplished in this optimization stage by transforming the loop pair to a loop nest. It is left for the loop unrolling optimization to complete the job by unrolling the inner loop, leaving the desired result: one loop that replaces the two original loops.

The essential idea behind the SA-C compiler's loop fusion can be seen with a simple example. Consider the SA-C loops:

```
uint8 A[:] =
    for window W[3] in Source
        return (array (f1 (W)));
uint8 Res[:] =
    for window W[3] in A
        return (array (f2 (W)));
```

If these loops are to be transformed into a single loop, the resulting loop's window must be wide enough to hold all the elements of Source needed to compute one value of Res, in this case a width of five. This is shown graphically in figure 4.5. One value of the result requires three intermediate values, and each of these in turn requires three values from the source, but they have some overlap.

This transformation also is designed such that the new loop produces the result values in the same order as they were produced by the original lower loop. This means that the lower loop body and its return operator (or operators) can be transferred directly to the new loop. It should be clear that if the new loop is to be an equivalent of the original pair, then the new loop must have a shape (rank and extents) that is identical to the lower loop. (This observation will be important when reasoning about the handling of multiple generators that may be present in the lower loop.) Figure 4.6 shows, at left, a DDCF graph for the above example, and in the center shows the transformation using the reasoning developed so far. It has a generator of the required width, as well as the loop body and return of the lower loop.

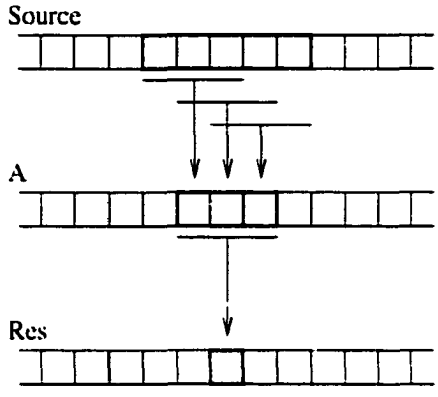


Figure 4.5: A size-three window generator, followed by a second size-three generator, requires five values from the source array.

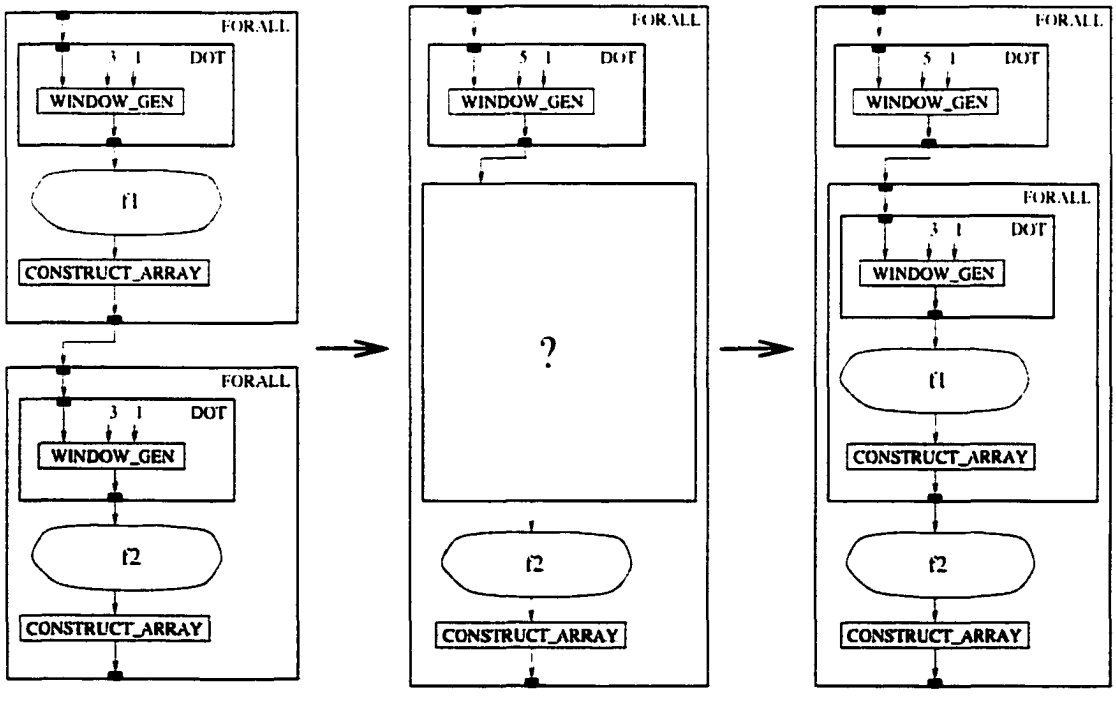


Figure 4.6: The development of a loop fusion transformation. At left is the original pair of loops. At center is a partial transformation, with a generator sufficiently wide to take in all values needed for one result element, and the loop body and return to create that element. At right is the completed transformation. The upper loop is moved into the new loop, and creates width-3 tiles that feed the body of the original lower loop.

To complete the transformation, code must be defined and placed between the generator and the body that was transferred from the lower loop. To understand what should go here, consider iteration i of the new loop. The undefined code body must produce the same sub-array as the window read by iteration i of the original lower loop, i.e. a width-3 window of the intermediate array. The width-5 window of the new generator produces all the values needed to compute this width-3 array, and the required block of code is most easily defined by creating an inner loop identical to the original upper loop. However, now it traverses a width-5 array rather than the source array and therefore produces a width-3 result each time it is executed. This is shown in the right part figure 4.6, and its corresponding SA-C code is as follows:

```
uint8 Res[:] =
    for window WT[5] in Source {
        uint8 T[:] =
            for window W[3] in WT
                return (array (f1 (W)));
            } return (array (f2 (T)));
```

Since the inner loop operates on an array of statically known size, and its window size is also statically known, it will be fully unrolled. The resultant code is then:

```
uint8 Res[:] =
    for window WT[5] in Source {
        uint8 V0 = f1 (WT[0:2]);
        uint8 V1 = f1 (WT[1:3]);
        uint8 V2 = f1 (WT[2:4]);
        uint8 T[3] = V0, V1, V2;
        } return (array (f2 (T)));
```

This approach to loop fusion creates redundant computation. For example, in the above loop the evaluations of `f1 (WT[0:2])` and `f1 (WT[1:3])` were also performed in previous iterations. If FPGA space is plentiful, this is probably not a problem since the three evaluations of f1 happen in parallel. However, another optimization can transform the above loop to a nextified version in which the redundant computations are eliminated and their values are carried from one iteration to the next, thereby saving space (see section4.2.1).

4.1.4.2 Generalizations and Limitations

The fusion approach developed above can be generalized in a number of ways. There are also inherent limitations in it.

4.1.4.2.1 Generators with different sizes and steps The two original generators can run windows of different sizes, each with its own step size. If c_U and c_L represent the upper and lower window sizes, and s_U and s_L represent their steps, then the window generator of the new loop will require window and step sizes as follows:

$$c = (c_L - 1) \times s_U + c_U$$

$$s = s_U \times s_L$$

This approach generalizes to multiple dimensions, each dimension being treated independently as shown above. Array-element generators (which extract scalar values) can be handled by first transforming them into window generators of size one.

4.1.4.2.2 Multiple generators in the upper loop If the upper loop has multiple window generators in a DOT graph, fusion can still be performed. Each window generator will yield a corresponding one in the resulting loop, according to the same rules as described above. For example, consider the SA-C loops:

```
uint8 A[:] =
    for window W1[3] in S1 step (2) dot window W2[4] in S2
        return (array (f1 (W1, W2)));
uint8 Res[:] =
    for window W[2] in A
        return (array (f2 (W)));
```

For the left upper loop generator and the lower loop generator,

$$c_U = 3,\ s_U = 2 \quad \text{and} \quad c_L = 2,\ s_L = 1 \quad \text{yielding} \quad c = 5,\ s = 2$$

For the right upper loop generator and the lower loop generator,

$$c_U = 4,\ s_U = 1 \quad \text{and} \quad c_L = 2,\ s_L = 1 \quad \text{yielding} \quad c = 5,\ s = 1$$

This produces the following loop after transformation:

```
uint8 Res[:] =
    for window WT1[5] in S1 step (2) dot window WT2[5] in S2 {
        uint8 T[:] =
            for window W1[3] in WT1 step (2) dot window W2[4] in WT2
                return (array (f1 (W1, W2)));
                } return (array (f2 (T)));
```

4.1.4.2.3 Multiple generators in the lower loop Additional generators of any kind, in a DOT graph, may coexist in the lower loop alongside of the window generator that traverses the output of the upper loop. These generators are transferred to the new loop, and their outputs feed the lower loop body, i.e. f2 in the examples. To understand why this is valid, recall that the new loop has a shape that is identical to that of the original lower loop, and it produces values in the same order. This means that the other generators of the lower loop, if transferred to the new loop, will have correct shapes and will produce their values in the correct sequence.

4.1.4.3 Multiple return values from the loops

Multiple values of any kind can be returned by the lower loop without affecting the fusion implementation, since the lower loop body that feeds them is copied directly into the new loop and it is fed by the same values it would have received in the unfused execution. However, there are significant limitations for the outputs of the upper loop and the ways in which they are used.

First, no output of the upper loop may feed a node other than the lower loop. To see why, note that in general the shapes of the arrays created by the upper loop are different than the shapes of those created by the lower loop. If an intermediate array is referenced outside of the loop pair, it would have to be produced by the new loop after fusion, but any array-returning node in the lower loop will not have the correct shape for the intermediate array.

Second, the upper loop may not send a reduction to the lower loop. This is understood by noting that the lower loop needs the value of the reduction while it traverses its source array, but the value is not available until the upper loop has completely traversed its sources. Thus a fused loop will not have the reduction's value when it is needed.

Third, if multiple return values of the upper loop feed multiple generators of the lower loop, they must either agree on the sizes and steps of the fused loop's generators, or separate generators must be formed for each. Consider an example (admittedly somewhat contrived since the dot product in the lower loop works only because of the fringes left due to non-unit steps):

```
uint8 A[:], uint8 B[:] =
    for window W[3] in Src
        return (array (f (W)), array (g (W)));
uint8 Res[:] =
    for a in A step (4) dot b in B step (5)
        return (array (h (a, b)));
```

The left generator of the lower loop combines with the upper generator to produce a generator in the fused loop of size 3 and step 4. The lower loop's right generator however suggests that the fused loop should have a generator of size 3 and step 5. Thus the fused loop must have two window generators, both traversing array Src but with different steps. In general, if both loops have multiple generators, the fused loop will need a generator for every possible pair of upper loop-lower loop generators. In practice, however, it is highly likely that the generators of the lower loop will have identical sizes and steps, meaning that the lower loop generators agree on the sizes and steps of the generators of the fused loop. It is probably not worth the effort to write the fusion optimization to handle the extreme general case. Furthermore, the case where the lower loop generators have identical window sizes and steps can be accommodated by a separate transformation that combines the return arrays of the upper loop into one array and gives the lower loop a single generator along with code that unpacks the values (see section 4.2.4). Such a transformation would make it unnecessary for the fusion optimization to handle multiple edges between a loop pair.

4.1.4.4 DDCF view of loop fusion

Figure 4.7 shows a loop fusion opportunity in DDCF form. It demonstrates that each of the loops may have additional generators, and that various other values entering the loops and used in various ways must be handled appropriately in the transformation. Figure 4.8 shows the result of the transformation.

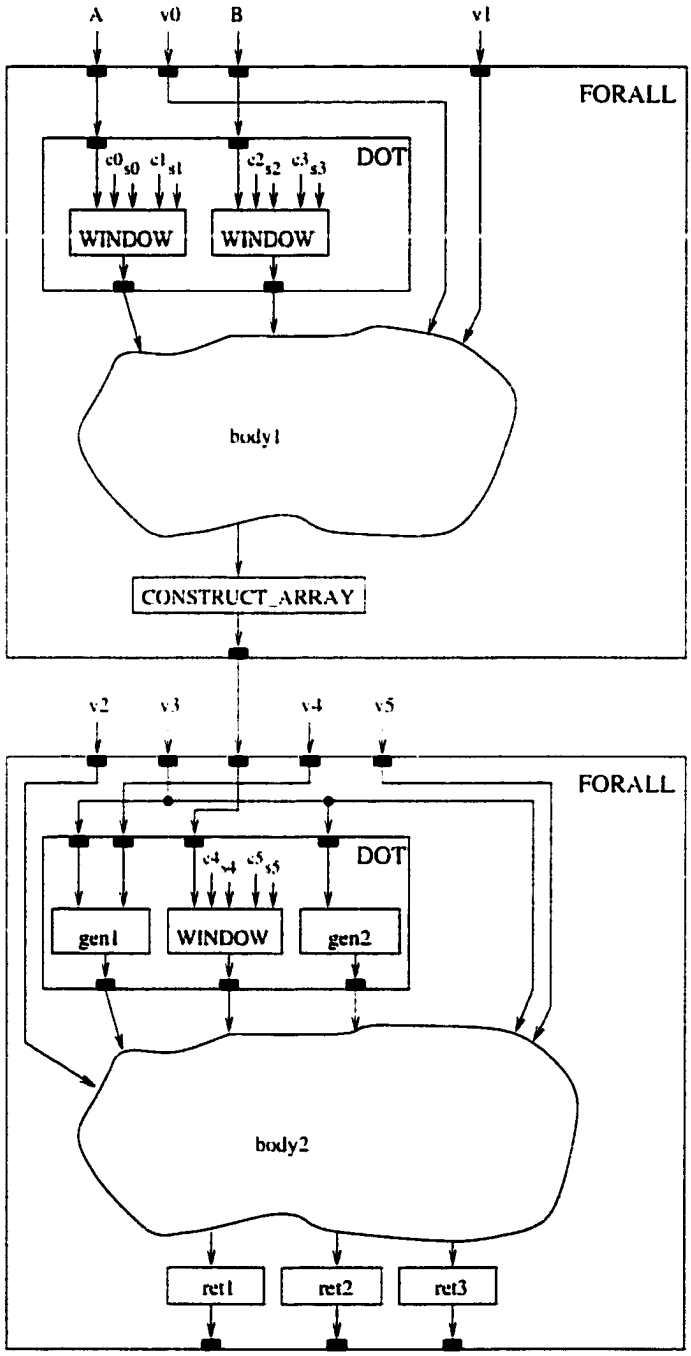


Figure 4.7: DDCF graph of loops before fusion.

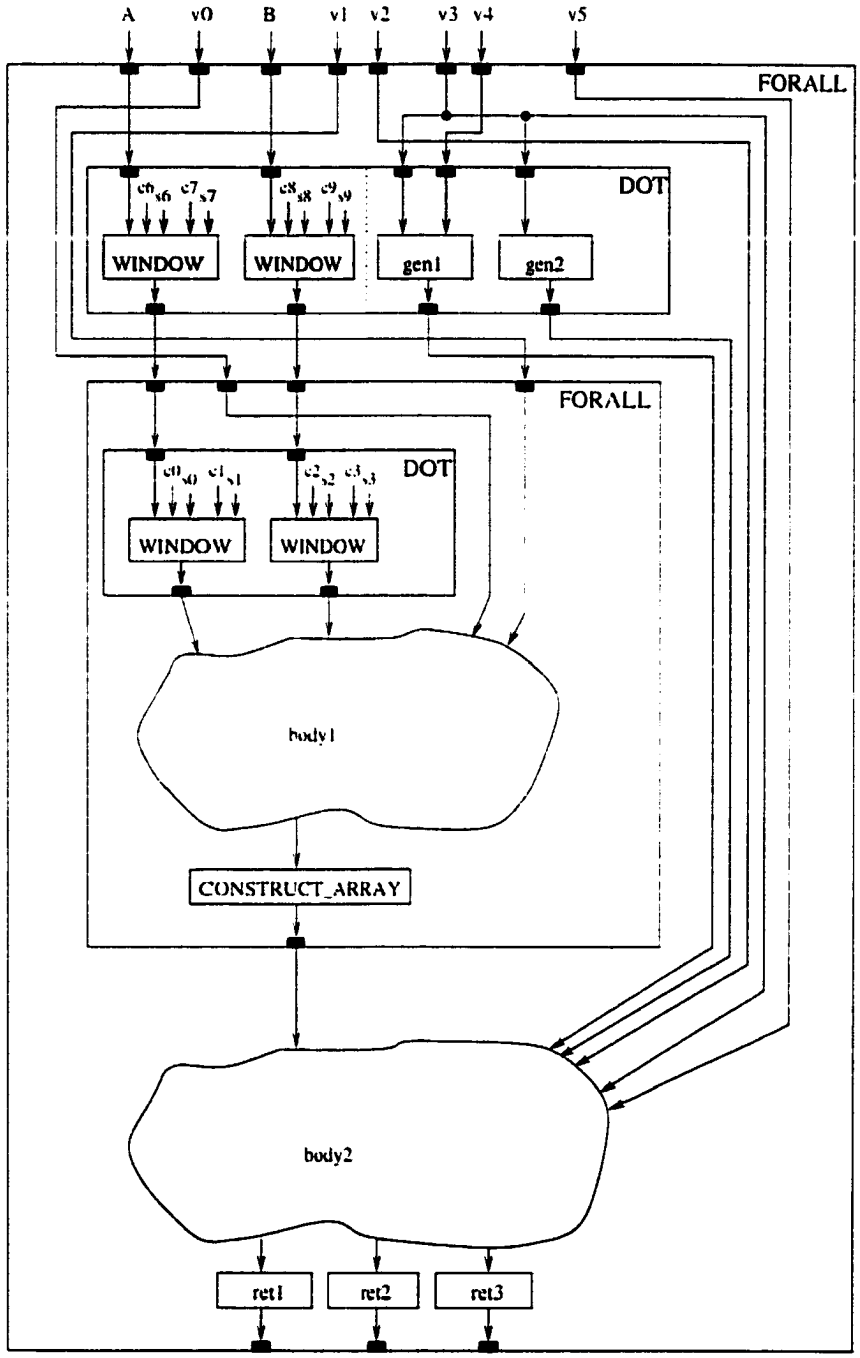


Figure 4.8: DDCF graph of loops after fusion.

The current implementation's criteria that allow loop fusion are now detailed. The upper loop requires that:

- it produces exactly one return value, formed by a CONSTRUCT_ARRAY node and feeding only the lower loop
- the input to the CONSTRUCT_ARRAY node is scalar
- the generator graph is DOT
- each simple generator is WINDOW, or ELEGEN with scalar output
- the window and step sizes are statically known

The lower loop requires that:

- the generator graph is DOT
- the input from the upper loop feeds only the generator
- the generator fed from the upper loop has statically-known window and step sizes

4.1.5 Lookup tables

Because SA-C has scalar types with user-specified bit-widths, functions sometimes have parameters consisting of relatively few bits, making it feasible to implement them as lookup tables. This means that the function's return values are computed by the compiler for all possible parameter values and saved as an array, and calls to the function are replaced by references to the array. The conversion of a function to a lookup table is controlled by the user through a pragma.

The first step in converting a function to a lookup table is the creation, by the compiler, of a new SA-C program that consists of one function called "main". This function has no parameters, and returns an array whose rank is equal to the number of parameters to the original function. The array's type is the type of the return value of the original function. The body of "main" is a loop that generates all possible values of the original function's parameters and whose loop body is the original function's body. Consider the following example:

```
// PRAGMA (lookup)
ufix10.5 magnitude (ufix4.1 a, ufix4.1 b)
    return (sqrt ((ufix9.2)a*a + (ufix8.2)b*b));
```

The new program created by the compiler is

```
ufix10.5[:,:] main () return (
    for uint32 i, uint32 j in [16,16] {
        ufix4.1 a = (bits4)i;
        ufix4.1 b = (bits4)j;
        ufix10.5 v = sqrt ((ufix9.2)a*a + (ufix8.2)b*b);
        } return (array (v)));
```

The newly created program is next written to a temporary file in DDCF format. The compiler then calls itself, compiling the new program to an executable. Once compiled, the executable is run and its output saved in binary format. The original function is then altered by creating a constant array from values read from the binary file, and producing a lookup as the function's return. Continuing the above example, the function now looks like this:

```
ufix10.5 magnitude (ufix4.1 a, ufix4.1 b) {
    ufix10.5 A[16,16] = { - - - <computed values> - - - };
    } return (A[(uint4)(bits4)a,(uint4)(bits4)b]);
```

The casts through **bits4** to **uint4** preserve the bit pattern of the parameters when used as array indices.

At this point the lookup table conversion is complete. Other optimizations carry on to produce the desired results: Function inlining will replace calls to the function with the function body (which includes the constant array definition). Typically the call occurs within a loop. If this is the case, the invariant code motion optimization will lift the array definition out of the loop. Also, if there were multiple calls to the function within the loop, the multiple array definitions will be replaced, during common subexpression elimination, by one definition. For example, consider the following loop using magnitude from the earlier example:

```
ufix10.5 R0[:], ufix10.5 R1[:] =
    for s0 in S0 dot s1 in S1 dot s2 in S2 dot s3 in S3 {
        ufix10.5 v0 = magnitude (s0, s1);
        ufix10.5 v1 = magnitude (s2, s3);
        } return (array (v0), array (v1));
```

After function inlining, code motion and constant subexpression elimination, the code will have been transformed to the following:

```
ufix10.5 A[16,16] = { - - - <computed values> - - - };
ufix10.5 R0[:], ufix10.5 R1[:] =
    for s0 in S0 dot s1 in S1 dot s2 in S2 dot s3 in S3 {
        ufix10.5 v0 = A[(uint4)(bits4)s0,(uint4)(bits4)s1];
        ufix10.5 v1 = A[(uint4)(bits4)s2,(uint4)(bits4)s3];
        } return (array (v0), array (v1));
```

The criteria for allowing conversion to a lookup table are

- The function must have only one return value.
- The return value must be scalar.
- The return value, and all parameters, may not be Float, Double, or any complex type.
- The total bit-width of all the parameters must not exceed 14 bits (a somewhat arbitrarily chosen limit to prevent extremely large lookup tables).
- The function may not have a function call in its body.

To accommodate lookup tables, function inlining is performed twice, once before and once after the lookup table conversion. The first pass inlines functions but skips those that are designated as lookup tables. (This means that the last criterion above is not really a problem unless there is a call to a function that has been given a **no_inline** pragma.) The second function inlining pass takes place after lookup table conversion, setting the stage for the code motion and common subexpression steps as described above.

4.1.6 Array-related optimizations

The SA-C compiler performs three array-related optimizations.

4.1.6.1 Array cast elimination

SA-C follows rules similar to C in the way it infers the size and type of an operation by the sizes and types of its operands. If this is applied in a consistent way to loop reduction operators, it requires

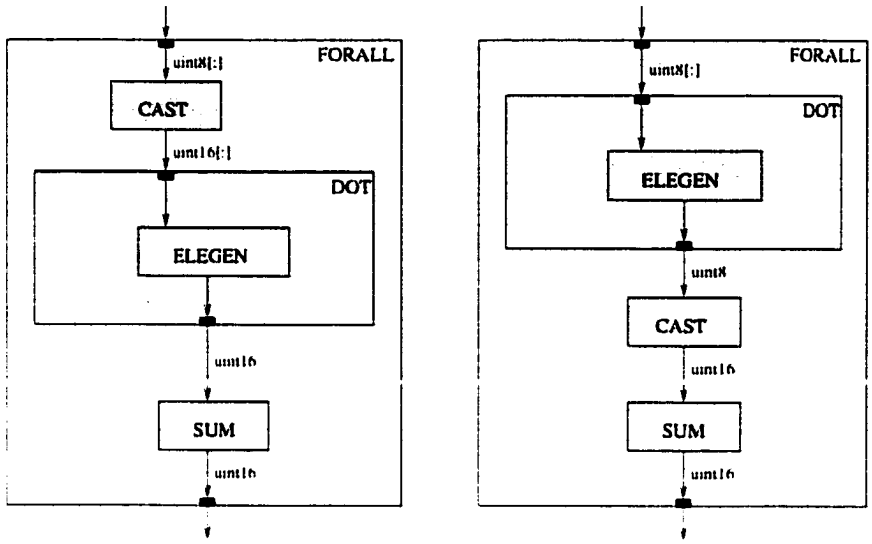


Figure 4.9: DDCF graph before and after array cast elimination.

the user to specify the size and type of the reduction through the use of a cast. For example, if the user wants to sum **uint8** values into a **uint16** result, the loop return can be written

```
return (sum ((uint16)v))
```

Unfortunately, the **array_sum** operator, which is turned into an explicit loop during translation into a DDCF graph, gives the user no way to express this except to cast the array to the desired width:

```
uint8 A[:] = ... ;
...
uint16 s = array_sum ((uint16[:])A);
```

This produces a DDCF graph as shown at left in figure 4.9. A straightforward translation of this graph into host code creates a new array of 16-bit elements, with all values copied from the old to the new array. Array cast elimination moves this cast to the output of the generator, where it is casting scalar values rather than an array, thus eliminating array allocation and copying. This is shown at right in figure 4.9.

4.1.6.2 Array value propagation

The application of other optimizations, especially full loop unrolling, often creates situations in which an array of statically-known size is created with individually computed values, feeding array

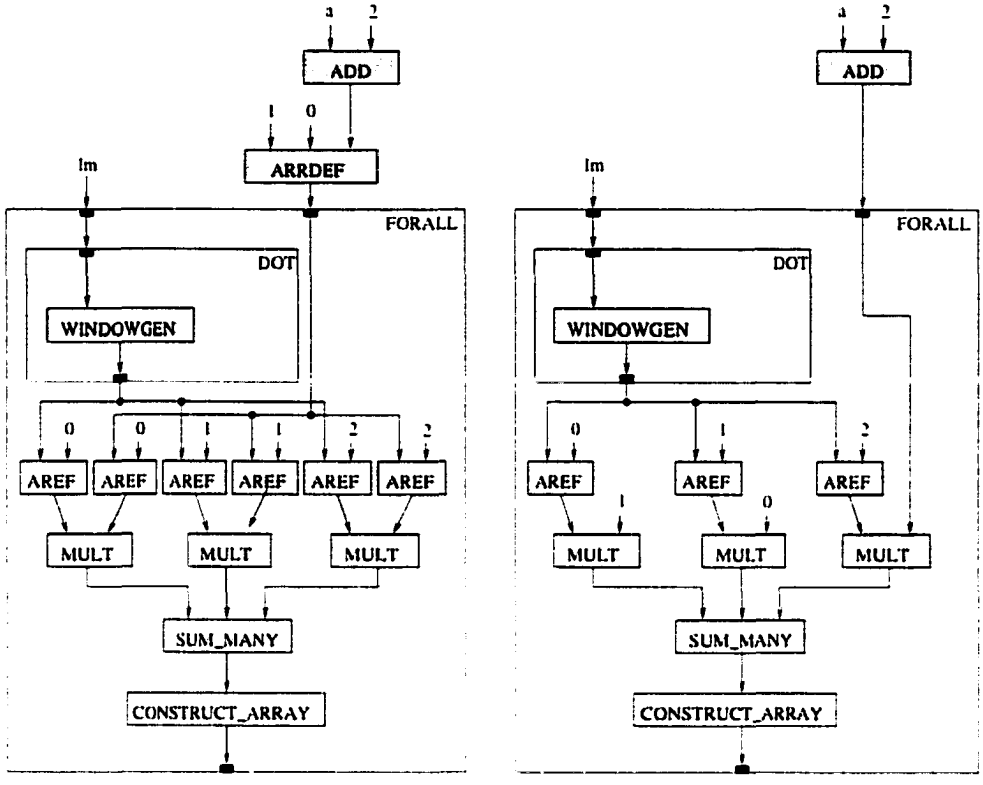


Figure 4.10: DDCF graph before and after array value propagation.

reference nodes that read values at statically-known indices. For example, consider the following SA-C code:

```
uint1 M[3] = {1, 0, a+2};
uint8 R[:] =
    for window W[3] in Im {
        uint8 v =
            for w in W dot m in M
                return (sum (w*m));
        } return (array (v));
```

After the inner loop is unrolled, this produces the DDCF graph shown at left in figure 4.10. Array value propagation moves the two constant array values directly to their points of use, and brings the a+2 expression into the loop to its point of use.

4.1.6.3 Array reference chain elimination

SA-C allows array slice operations, similar to those in Fortran, to be expressed, giving rise to AREF nodes in DDCF graphs. In some situations, an AREF node taking a slice may feed another AREF node or an EXTENTS node. The graphs in these cases can be simplified.

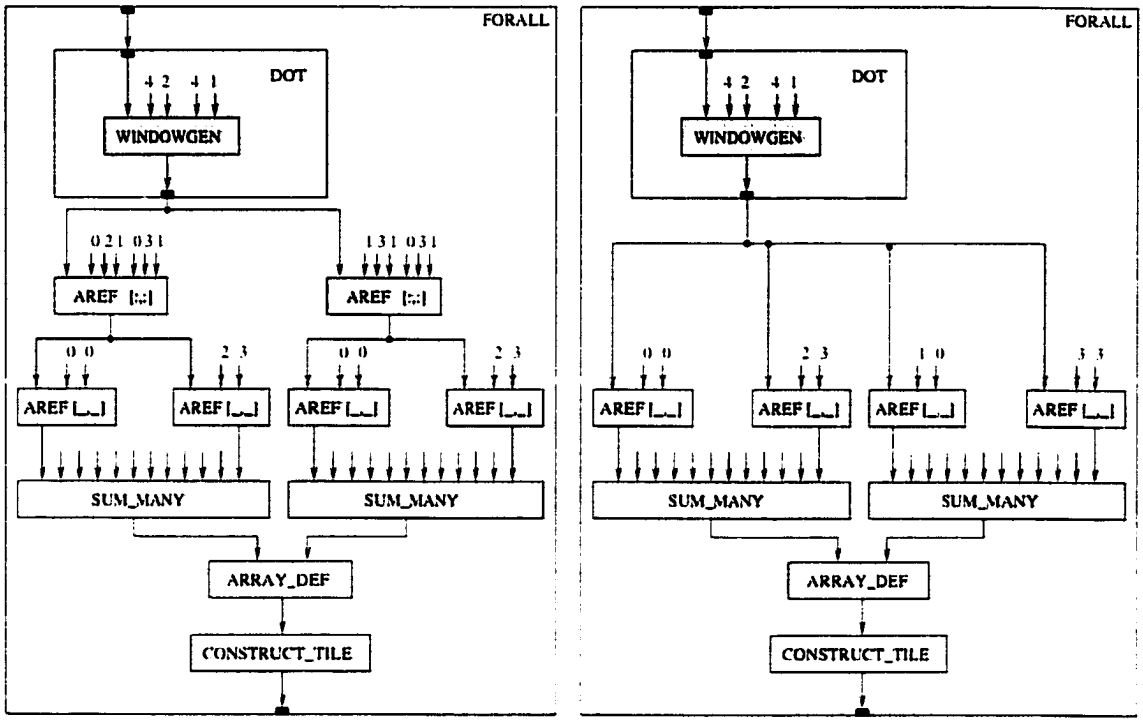


Figure 4.11: DDCF graphs, before and after array reference elision.

The following code shows a piece of SA-C code that gives rise to pairs of AREF nodes:

```
uint8[:,:] main (uint8 A[:,:]) {
    uint8 R[:,:] =
        // PRAGMA (stripmine (4,4))
        for window W[3,4] in A
            return (array (array_sum (W)));
    } return (R);
```

The loop is stripmined, creating an outer loop with a 4x4 window generator. The inner loop is then unrolled, creating two loop bodies, each of which takes a slice of the 4x4 window. AREF nodes extract elements from these slices. Figure 4.11 shows the DDCF graph at this stage of the optimization process. After array reference chain elimination, the two slicing AREF nodes are eliminated by dead-code elimination.

AREF nodes can be eliminated by the following transformations. Consider first a pair of AREF nodes in a producer/consumer relationship:

```
uint8 B[:] = A[a:b:c];
uint8 C[:] = B[d:e:f];
```

can be replaced with

```
uint8 C[:] = A[c*d+a:c*e+a:c*f];
```

This can be seen as a static computation of dope vectors. A related situation can occur if the second AREF node takes a scalar:

```
uint8 B[:] = A[a:b:c];
uint8 C = B[d];
```

can be replaced with

```
uint8 C[:] = A[c*d+a];
```

These two situations can be generalized to multidimensional arrays, where each dimension is treated individually according to the above transformations.

A related situation occurs when an AREF taking a slice is followed by an EXTENTS node. The EXTENTS node can be eliminated using the transformation from

```
uint8 B[:] = A[a:b:c];
uint8 r = extents (B);
```

to

```
uint8 B[:] = A[a:b:c];
uint8 r = (b-a+1)/c;
```

Again, this generalizes to multiple dimensions by treating each dimension individually.

4.1.7 Bit-width narrowing

The ability in SA-C to specify the bit widths of the scalar types opens up opportunities for space optimization in FPGAs by inferring that certain subexpressions can be computed with fewer bits than specified without changing the result of the expressions. Such situations arise, for example, in loops with loop-carried dependencies ("nextified" loops in SA-C) where values grow or shrink from one iteration to the next.

The SA-C function in figure 4.12 computes the square root of a six-bit unsigned value. This loop can be fully unrolled, producing three loop bodies, shown in figure 4.13. There are four nextified

```
uint3 sq_root (uint6 vsqn) {
    bits6 vsq = vsqn;
    bits6 asq = 0;
    bits3 a = 0;
    bits6 tvsq = 0;

    uint3 v =
        for uint3 i in [3] {
            bits6 nasq = ((bits6)((uint6)asq+(uint3)a)<<2) | 0b1;
            bits3 sa = a<<1;
            next tvsq = (tvsq<<2) | ((bits2)(vsq>>4));
            next vsq = vsq<<2;
            next a, next asq =
                if (nasq <= tvsq) return (sa|0b1, nasq)
                else return ( sa, asq<<2);
            } return (final (a));
    } return (v);
```

Figure 4.12: SA-C code to compute the square root of a six-bit value.

variables, three of which are initialized to zero, allowing constant folding to remove eight nodes in the first iteration. Since three of the nextified variable values are not needed after the loop completes, dead code elimination removes three of the nodes in the last iteration. It is important to note that bit shifts by statically known distances do not consume FPGA logic, since they are simply interconnection. The same is true of CHANGE-WIDTH nodes, which truncate or pad with zeroes; they are shown in the diagram as small rectangles. The small circles show the bit widths of the logic required for the node. In the case of BIT-OR, the required logic is the lesser of the two input widths. The widths of the types specified by the programmer are sufficient to hold the widest of each of the values, even though not every iteration needs that much width. For example, the UADD nodes are six bits wide, even though the UADD node in the middle iteration is adding two 1-bit values (the constant inputs of the SELECTOR nodes feeding it) and therefore needs a width of only two bits.

A first approach to narrowing bit-widths might apply straightforward rules based on the worst case for each given operator. For example, and ADD node with input bit-widths of w_0 and w_1 requires $1 + \max(w_0, w_1)$ bits, and a LSHIFT of a value with bitwidth w, shifted by distance d, is $w + d$. However, these rules sometimes produce widths that are wider than necessary. Consider these two examples:

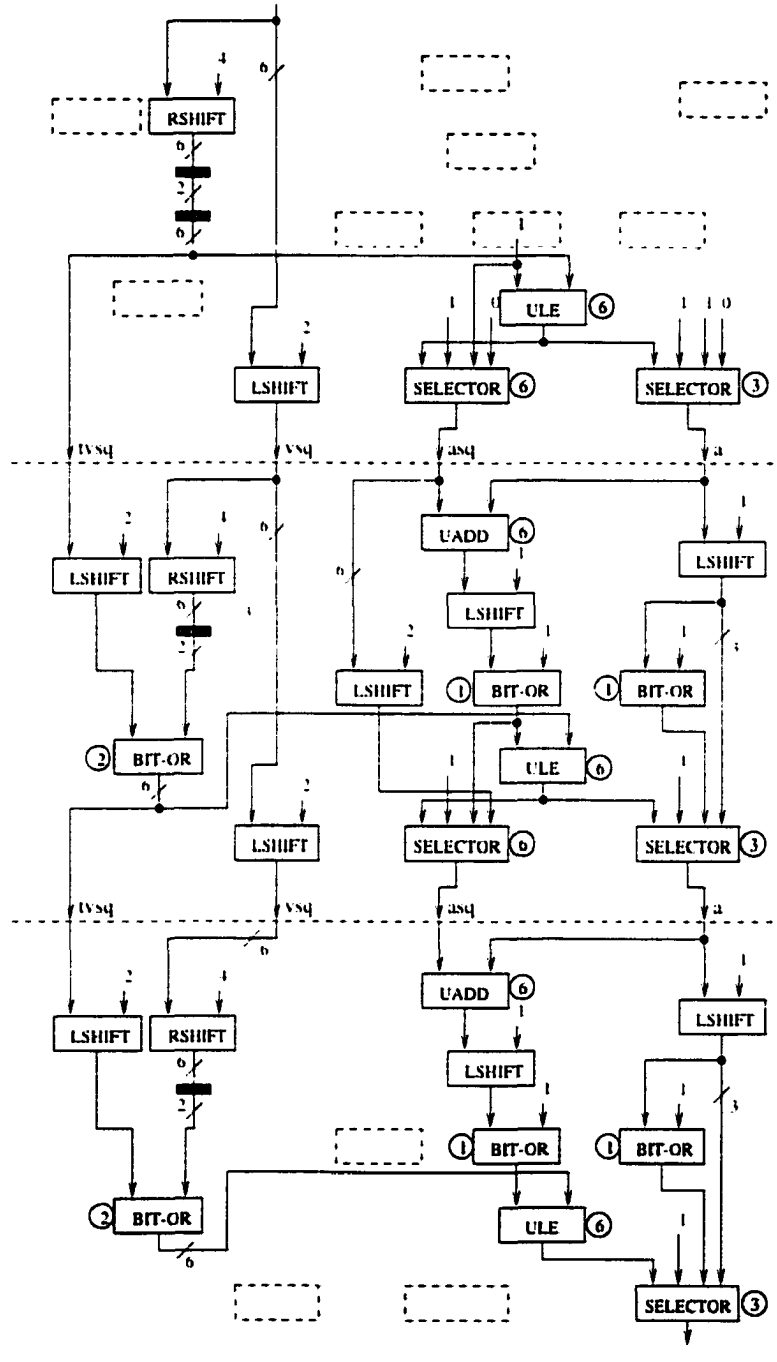


Figure 4.13: Three iterations of square root loop, before narrowing. The dashed boxes in iteration one were removed by folding. The dashed boxes in iteration three were removed as dead code.

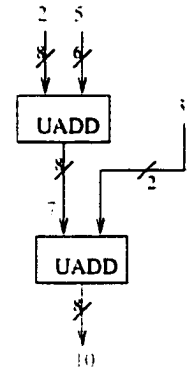
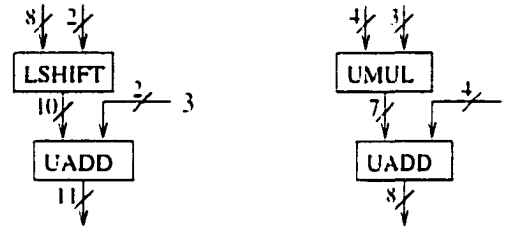


Figure 4.14: Example of semantic problem with downward bit width propagation. Red values are maximum values. If the upper node's output width were inferred to be three bits, based on its max value of seven, the addition in the lower node would be too narrow.



At left, the left shift to ten bits leaves two zeroes on the right, so adding three is guaranteed to fit in ten bits, even though the simple rules dictate eleven. At right, the multiply requires seven bits, but the largest value that can come out of the UMUL node is 15×7, or 105. The four-bit input to the UADD node cannot be greater than 15, so the result of the UADD cannot be larger than 120, which fits in seven bits, even though the rules dictate eight bits.

A better approach to narrowing bit-widths uses the maximum values on edges, rather than just the numbers of bits. It is possible to establish, for each node output, a maximum unsigned bit value. In the case where only the bit width of the output is known, the maximum value will be $2^w - 1$ where w is the width in bits. However, a tighter bound is often possible, as in the above examples. It is not difficult to establish the rules, for each node type, by which the maximum output values can be computed, given their maximum input values. This information is propagated through the dataflow graph in a downward pass.

The maximum values are of interest since they can be used to determine the number of bits each edge needs to convey its information. However, an optimization that narrows bit widths must

conform to the semantics of the source language. In SA-C, the semantics of an operation such as UADD defines the result to be computed by casting the narrower input value to the width of the wider one, then performing the addition at that bit width, and finally casting the result to the output width. Thus, the addition of two 1-bit values will not produce a correct 2-bit result unless one of the two inputs is cast to a **uint2** type. Because of this, bit widths cannot be assigned during the downward pass as the maximum values are being determined. To see why, consider the dataflow graph shown in figure 4.14. When visiting the upper UADD node, a maximum output value of seven might seem to imply that the output width of the node can be set to three bits. However, that width is too narrow for the lower UADD node, for it would do its arithmetic at a 3-bit width that is incapable of producing the value ten (its maximum value). Of course, it is also not possible to know what output width the lower node requires, since it may similarly affect nodes below it. This shows that, while a downward pass can be used to tag the edges with their maximum bit widths, it must be followed by an upward pass that narrows the widths according to SA-C's semantics.

The downward pass tags each node's input and output ports with a maximum value as follows:

```
for each node N
    for each input port P
        if is-constant (P) then
            P_maxval = constval (P)
        else
            S = source (P)
            P_maxval = S_maxval

    for each output port P
        apply node-specific rules using the maxvals of the input ports
```

The upward pass assigns new bit widths to the ports, inserting CHANGE-WIDTH nodes where needed:

```
for each node N
    for each output port P
        if reads-memory (N) then
            bit width is not changed
        else
            P_width = min (P_width, max (target widths))
        for each target T from P
            if P_width ≠ T_width then
                insert a CHANGE-WIDTH node to accommodate the mismatch
```

for each input port P
apply node-specific rules using the bit widths of the output ports

The dataflow graphs that are produced after application of this algorithm may have CHANGE-WIDTH nodes that don't actually cause a width change, i.e. the input and output widths are the same. A pass through the dataflow graph is used to remove these.

Figure 4.15 shows the three loop bodies of the square root graph after bit width narrowing has been performed. The widths of the BIT-OR, CADD, CLE and SELECTOR nodes start narrow and grow with each iteration.

Another kind of narrowing opportunity may occur where a programmer has left some widths wider than necessary. For example, consider

```
uint8 a = ...
uint8 b = ...
uint6 c = a + b;
```

According to SA-C semantics, the add operation should be eight bits wide, but since the result is being assigned to a six-bit variable, the addition itself can be narrowed to six bits. The narrowing algorithm described above will reduce the input widths of the CADD node to six bits, potentially saving FPGA space.

4.1.8 Conventional optimizations

The SA-C compiler performs a variety of common optimizations [1]. Their applications to DDCF graphs are discussed here.

In **constant folding**, the compiler computes the value of a node's output if all of its inputs are constant. The node is then replaced by the value. The SA-C compiler does constant folding for all of the common arithmetic and logic operator nodes. It also computes the value of, and then eliminates, intrinsic function call nodes with constant input(s). It also folds away CAST nodes with constant inputs. The ...MANY nodes, produced when loops are fully unrolled, present opportunities for folding as well. These nodes have an arbitrary number of pairs of inputs, each pair representing a value and a boolean mask bit. During constant folding, any pair of inputs with a constant FALSE

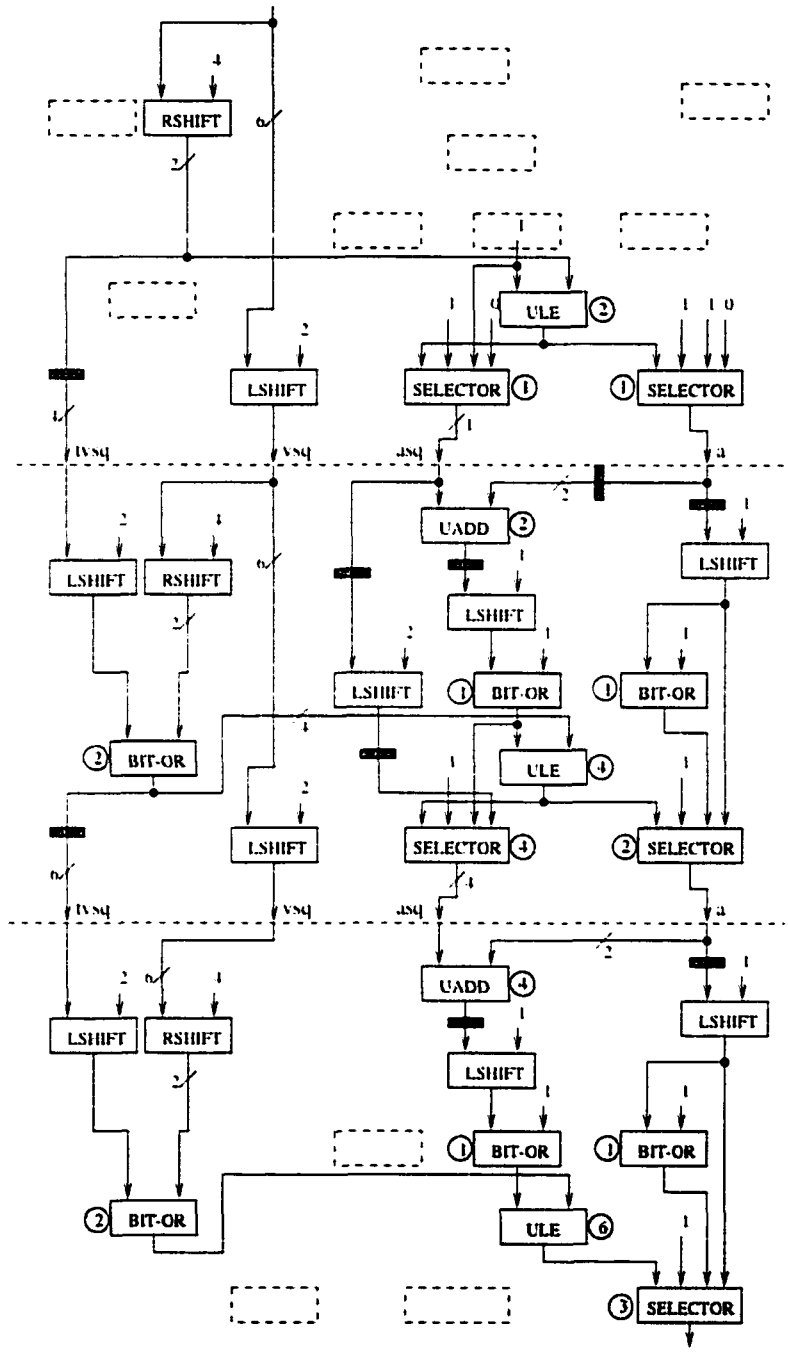
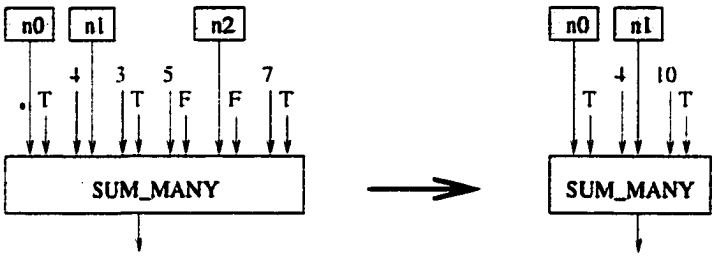


Figure 4.15: Three iterations of square root loop, after narrowing.

on its mask input can be eliminated. Also, all value inputs with constant TRUE bits on their mask inputs can be combined into one value by the compiler. Here is an example of this folding:



Algebraic **identities** and **strength reduction** are used to simplify graphs. The SA-C compiler simplifies ADD and SUB nodes with a '0' input; MUL nodes with '0', '1' or '-1' inputs; any DIV node with '1' as its right operand, BIT_AND and BIT_OR nodes with constants containing all zeros or all ones; and bit shift nodes with '0' shift distances. The SA-C compiler converts multiplies and divides by powers of two with equivalent bit-shift equivalents.

Common subexpression elimination combines nodes of identical type and operating on identical inputs. The SA-C compiler does this for all non-compound nodes that are within a given compound node. Experience has shown that this optimization often consumes significant time when graphs are large, since it looks at all pairs of nodes within a compound node. To speed this process, a graph's nodes are partitioned by hashing them with a function whose inputs are the node type and its inputs. Nodes that can be combined will always occur within the same partition, so each of the partitions can be processed separately, often speeding up the process.

Dead code elimination involves removing nodes whose output values are not used. This is easy in a DDCF graph, since it simply requires seeing whether a node has any outgoing edges.

Invariant code motion involves lifting nodes out of a loop graph if the values they compute do not change as the loop iterates. This is easy to recognize in a DDCF by following incoming edges upward. If every one of a node's inputs is either constant or comes from outside the loop, the node can be pulled out of the loop.

Constant-expression switch elimination searches for switch structures whose switch expressions are statically known. Each SWITCH graph is then replaced by the graph from the appropriate CASE node it contains.

4.2 Unimplemented optimizations

The following optimizations are not yet implemented, but nevertheless can be tested for performance since they are DDCF-to-DDCF transformations and are therefore expressible in SA-C.

4.2.1 Nextification of FORALL loops

SA-C's window generators in FORALL loops can cause redundant computation in a variety of circumstances. The simplest example is a rank-two window that is summed:

```
<type> R[:,:] =
    for window W[3,3] in A
        return (array (array_sum (W)));
```

This produces a DDCF graph in which, after application of optimizations, the window's nine outputs feed a SUM_MANY node, causing eight additions. After the first iteration of a given row traversal, the window slides to the right, and six elements of the new window are the same as those in the previous. When the window slides again (iteration three) three elements are the same as those in the first iteration. This is more clearly seen if the loop is transformed to:

```
<type> R[:,:] =
    for window W[3,3] in A {
        <type> C0[:] = W[:,0];
        <type> C1[:] = W[:,1];
        <type> C2[:] = W[:,2];
        <type> S0 = array_sum (C0);
        <type> S1 = array_sum (C1);
        <type> S2 = array_sum (C2);
        <type> S = S0 + S1 + S2;
        } return (array (S));
```

The value of S1 is the same as the value of S0 one iteration earlier, and the value of S2 is the same as the value of S0 two iterations earlier. If the partial sum S0 could be passed from one iteration to the next, two redundant sums could be eliminated. SA-C's "nextified" variables were created to express loop-carried dependencies such as these.

To understand how nextified variables can be used to eliminate some of these redundant computations, first consider the case of a loop with a single window generator. (This restriction will be eased later.) There are two steps involved in this transformation. The first step builds a loop

nest in which the *outer* loop extracts rank-two full-width horizontal stripes, of a height equal to the window height, from the source array; the *inner* loop runs the original window across the stripe and will be transformed into a nextified equivalent. The outer loop's job is to generate the stripes and produce the initial values for the inner loop's nextified variables. To create the loop nest, a hybrid window/array-slice generator is used (see section 2.4.1). After the first step of the transformation, the above example looks like this:

```
<type> R[:,:] =
    for Stripe[3,:] in A {
        <type> T[:] =
            for W[:,3] in Stripe {
                <type> C0[:] = W[:,0];
                <type> C1[:] = W[:,1];
                <type> C2[:] = W[:,2];
                <type> S0 = array_sum (C0);
                <type> S1 = array_sum (C1);
                <type> S2 = array_sum (C2);
                <type> S = S0 + S1 + S2;
                } return (array (S));
            } return (array (T));
```

Hybrid generators are used in both loops. The outer loop takes complete slices in the horizontal dimension, whereas the inner loop slices vertically. The inner loop returns vectors, and the outer loop packs them into a rank-two array.

The second step replaces redundant computations in the inner loop with nextified variables. Note that the inner loop extracts sub-arrays from its window variable W: these are AREF nodes in the DDCF graph. The general approach is to visit each AREF node, starting with the ones taking slices from the right edge of the window, and look "left" to see if there are other AREF nodes that will be reading the same sub-array in subsequent iterations. If so, and if their outputs are feeding identical computation subgraphs, then they can be replaced with nextified variables.

In the example above, the nextification transformation recognizes that the window generator is feeding three slices, that slices C0 and C1 are left shifts of slice C2, and that their outputs go to identical computations, in this case SUM_MANY nodes that were produced by unrolling the **array_sum** operators. This means that the computations of S1 and S2 can be replaced with nextified variables, whose initial values are created before entering the inner loop, yielding:

```
<type> R[:,:] =
    for Stripe[3,:] in A {
        <type> C0[:] = Stripe[:,0];
        <type> C1[:] = Stripe[:,1];
        <type> S0 = array_sum (C0);
        <type> S1 = array_sum (C1);
        <type> T[:] =
            for W[:,3] in Stripe {
                <type> C2[:] = W[:,2];
                <type> S2 = array_sum (C2);
                <type> S = S0 + S1 + S2;
                next S0 = S1;
                next S1 = S2;
                } return (array (S));
            } return (array (T));
```

In general, this transformation can deal with dotted window generators, as well as multiple loop returns as long as they are CONSTRUCT_ARRAY, CONSTRUCT_TILE or commutative/associative reductions. It also can deal with irregular occurrences of sub-slices from the generators, various loop ranks and non-unit step sizes. However, the window and step sizes of the generators, as well as the slicing parameters of the AREFs, must be statically known. To see a more general example, consider the following partial inner loop:

```
for window WA[6,9] in A dot WB[5,10] in B step (1,2) {
    <type> V0[:,:] = f (WA[0:2,4:5]);
    <type> V1[:,:] = f (WA[0:2,7:8]);
    <type> V2[:] = g (WA[2:5,1]);
    <type> V3[:] = g (WA[2:5,3]);
    <type> V4[:] = g (WA[2:5,6]);
    <type> V5[:,:] = h (WB[0:1,1:2]);
    <type> V6[:,:] = h (WB[0:1,4:5]);
    <type> V7[:,:] = h (WB[0:1,8:9]);
    <type> V8[:] = m (WB[2,2:4]);
    <type> V9[:] = m (WB[2,7:9]);
    <type> V10[:,:] = p (WB[3:4,0:2]);
    <type> V11[:,:] = p (WB[3:4,4:6]);
```

Figure 4.16 shows the two windows, and the sub-arrays that are being referenced. This example avoids sub-array overlap for clarity, but sub-arrays can overlap without affecting the transformation. Searching for nextification opportunities in the window of A starts with the sub-array WA[0:2,7:8], associated with the computation of V1, in the upper right corner. Since A's window generator has a unit step in the horizontal dimension, the sub-array is stepped leftward with unit stride until it

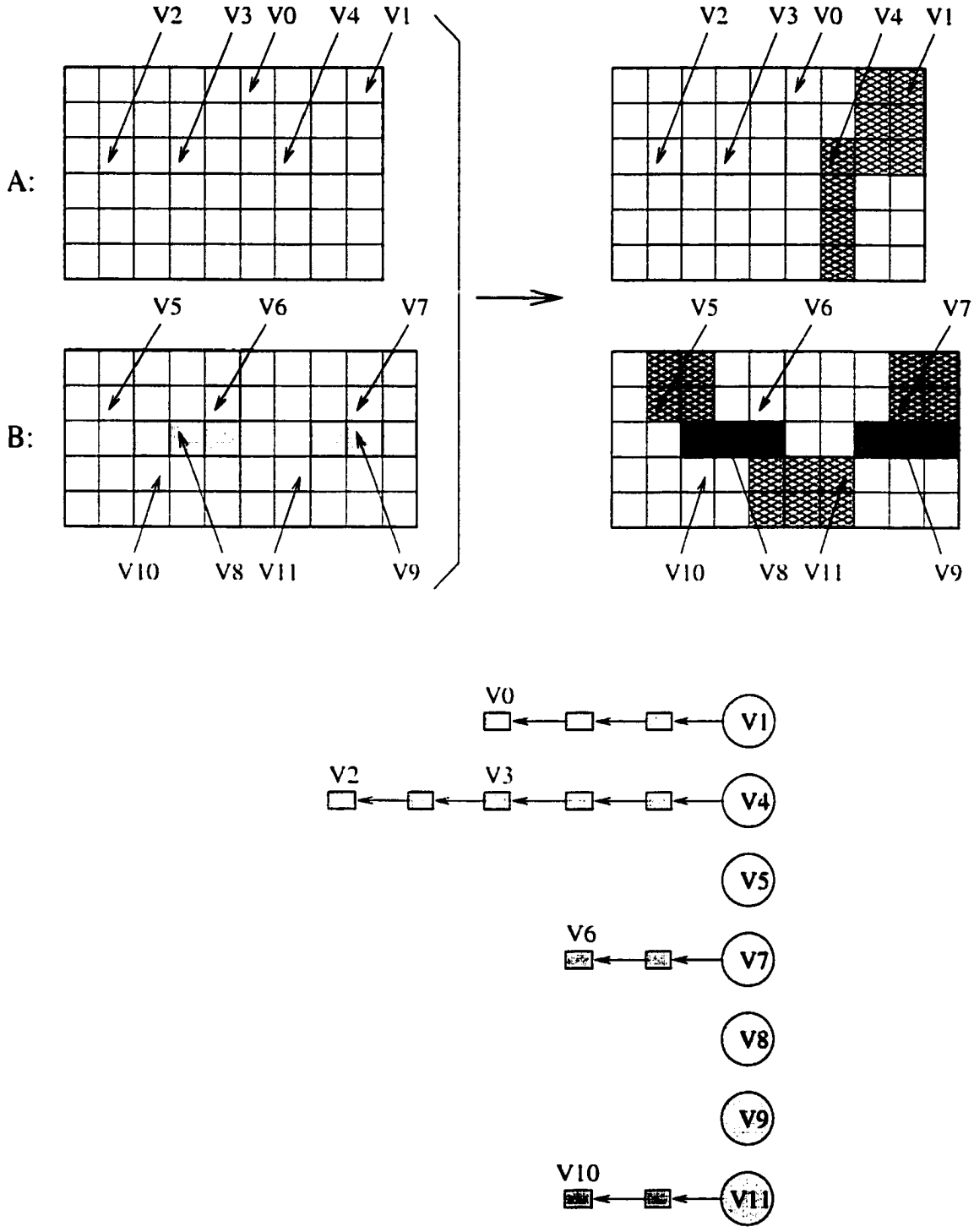


Figure 4.16: Example from text, showing the identification of values computed in previous iterations. The patterned sub-arrays in the right-hand arrays are the ones that are used by computations in the loop. The computations of the others are eliminated by bringing them in via nextified variables. The lower part of the diagram shows the shift registers and value computations derived from analyzing the upper diagram. Computations are shown as circles, and registers as small rectangles.

finds WA[0:2,4:5], the same shape sub-array, with both feeding a computation f. The computation of V0 can be replaced by a nextified variable, and since it was three steps to the left the value must be threaded through three iterations. The right half of the figure shows what happens after nextification. The patterned areas are sub-arrays used for computations, while the others have been replaced by nextified variables. Note that since the window generator traversing B has a horizontal step of two, the computation of V5 produces a value that was not computed in previous iterations, so it cannot be nextified. The same situation applies for V8. The lower part of the figure shows the actual computations and the registers that are produced for this example. Each iteration shifts values leftward.

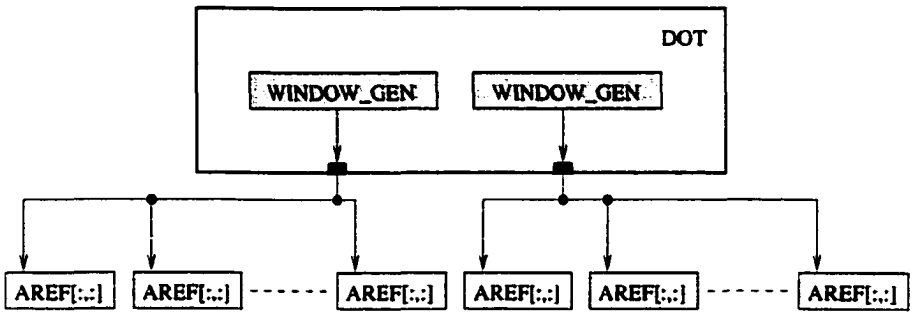
The SA-C code after nextification is shown in figure 4.17. The names have been chosen to show the computed value and the iteration distance. For example, V4_3 is value V4 from three iterations back. The initial values of the nextified variables must be computed for each of the different iteration distances. For example, since the value of V4 in the loop uses sub-array WA[2:5,6], the nextified variable for one iteration distance back, V4_1, uses WA[2:5,5]. Note that because B's window generator has a step of two, V7 and V11 have nextified variables for distances one and two in *iteration* space, which correspond to sub-arrays of distances two and four in *array-index* space.

It is useful to note that after this transformation, the inner window generators may be larger than they need to be. In the simple summing example, the inner loop generator's first two columns are never referenced. A separate optimization can by implemented to reduce window sizes in such cases (see section 4.2.3). It is also important to note that, though nextified variables in general may produce cycles in their dataflow graphs, this transformation does *not* create cycles. This is important because an acyclic graph can be pipelined for improved performance.

Perhaps the most interesting opportunity for this transformation arises from the loop fusion transformation described in section 4.1.4, in which many computations of the upper loop are repeated as the windows of the fused loop traverse their source arrays. After fusion and the subsequent unrolling of the inner loop, the windows of the upper loop become slices of the transformed loop's windows, producing DDCF subgraphs that resemble the following:

```
for StripeA[6,:] in A dot StripeB[5,:] in B {
    <type> V1_3[:,:] = f (A[0:2,4:5]);
    <type> V1_2[:,:] = f (A[0:2,5:6]);
    <type> V1_1[:,:] = f (A[0:2,6:7]);
    <type> V4_5[:] = g (A[2:5,1]);
    <type> V4_4[:] = g (A[2:5,2]);
    <type> V4_3[:] = g (A[2:5,3]);
    <type> V4_2[:] = g (A[2:5,4]);
    <type> V4_1[:] = g (A[2:5,5]);
    <type> V7_2[:,:] = h (B[0:1,4:5]);
    <type> V7_1[:,:] = h (B[0:1,6:7]);
    <type> V11_2[:,:] = p (B[3:4,0:2]);
    <type> V11_1[:,:] = p (B[3:4,2:4]);
    <type> TA[:,:], ¡type¿ TB[:,:] =
        for WA[:,9] in A dot WB[:,10] in B step (1,2) {
            <type> V0[:,:] = V1_3;
            <type> V1[:,:] = f (WA[0:2,7:8]);
            <type> V2[:] = V4_5;
            <type> V3[:] = V4_3;
            <type> V4[:] = g (WA[2:5,6]);
            <type> V5[:,:] = h (WB[0:1,1:2]);
            <type> V6[:,:] = V7_2;
            <type> V7[:,:] = h (WB[0:1,8:9]);
            <type> V8[:] = m (WB[2,2:4]);
            <type> V9[:] = m (WB[2,7:9]);
            <type> V10[:,:] = V11_2;
            <type> V11[:,:] = p (WB[3:4,4:6]);
            ...
            next V1_3 = V1_2;
            next V1_2 = V1_1;
            next V1_1 = V1;
            next V4_5 = V4_4;
            next V4_4 = V4_3;
            next V4_3 = V4_2;
            next V4_2 = V4_1;
            next V4_1 = V4;
            next V7_2 = V7_1;
            next V7_1 = V7;
            next V11_2 = V11_1;
            next V11_1 = V11;
            }
```

Figure 4.17: SA-C code after nextification of loop example in text.



Each AREF node is extracting a window that would have been produced by the original upper loop, and many of these nodes feed identical sub-graphs, meaning that redundant computation is taking place. Replacing these redundant computations with nextified values should save significant logic space in the FPGAs.

4.2.2 An alternate approach to nextification

Section 4.2.1 showed a general approach to reducing redundant computations in a **for** loop by introducing nextified variables. This section shows a variation that, while less general, can produce a more efficient loop in some circumstances. It also looks at this alternate form as applied in the most common source of nextification opportunities: loop fusion.

4.2.2.1 The transformation

The previous approach has two potential disadvantages. First, it requires the host program to compute the initial values of the nextified variables for each execution of the inner loop, i.e. for each stripe of a source array. Second, by creating a loop nest it has produced an execution pattern in which there are repeated calls by the host to the inner loop on the RCS. In the present system, each call requires an individual transfer of its stripe to the RCS and a transfer of a result row back to the host. (Of course, it is possible to improve the system so that data transfers happen just once and the calls to the RCS loop simply send pointers to source and target sub-arrays.)

An alternate approach uses the RCS loop to compute the initial nextified variable values, by padding the source array on the left with dummy values, running the loop, and discarding the leftmost junk values from the result. In other words, a few extra iterations of the loop are run at the beginning of each horizontal traversal in order to shift the correct values into the nextified variables.

Since the host does not have to compute the starting values, it is no longer necessary to create a loop nest.

The first step in this transformation is the same as in the previous nextification approach: examine the sub-arrays referencing the windows and look leftward to find values that have been computed in previous iterations. The lower part of figure 4.16 shows an example of the computations and shift registers that are created. The next step is the padding of the source arrays. Since this padding creates the extra loop iterations needed to fill the nextified registers, the number of iterations required is derived from the maximum number of registers in any of the register chains; call this d. In the figure's example, five iterations are needed to fill the registers associated with V4. Multiplying the required number of iterations by the step size gives the number of columns that must be padded onto the array to create those iterations.

SA-C requires nextified variables to be given initial values before the loop is entered. In this transformation those values do not matter, but since they must be supplied they can simply be set to zero. Note that these initial values will feed into the first horizontal sweep, and each subsequent sweep will receive the values left from the previous sweep; again, these values do not matter.

Since the source array was padded, the result array is too big, and has junk values along its left. For a CONSTRUCT_ARRAY return node, the number of extra columns will be d, since that many iterations were added. For a CONSTRUCT_TILE node, with a horizontal tile extent of e, the number of extra columns will be $d \times e$. The host code will discard these columns by taking a slice of each returned array.

Here is an example, simplified by using a rank-one array. (The horizontal dimension is the interesting one; in the other dimensions nothing is changed from the nextification discussion of section 4.2.1.) The loop before transformation looks like this:

```
<type> R[:] =
    for window W[8] in A step (2) {
        <type> v0 = f (W[5:7]);
        <type> v1 = f (W[4:6]);
        <type> v2 = f (W[3:5]);
        <type> v3 = f (W[2:4]);
        <type> v4 = f (W[1:3]);
        <type> v5 = f (W[0:2]);
```

```
    ...
    } return (array (x));
```

The values v0 and v1 cannot be nextified. The values v2 and v4 are nextified from v0, and the values v3 and v5 are nextified from v1. Each of these chains has two registers, so two new iterations are needed, and since the step size is two, this will require four values to be padded on the left of array A. The return array will have two junk elements on its left, so following the loop they are discarded. The result is:

```
<type> Pad[4] = {0,0,0,0};
<type> AA[:] = array_concat (Pad, A);
<type> v2 = 0;
<type> v3 = 0;
<type> v4 = 0;
<type> v5 = 0;
<type> RR[:] =
    for window W[8] in AA step (2) {
        <type> v0 = f (W[5:7]);
        <type> v1 = f (W[4:6]);
        ...
        next v5 = v3;
        next v4 = v2;
        next v3 = v1;
        next v2 = v0;
        } return (array (x));
<type> R[:] = RR[2:];
```

An interesting observation can be made here: the window's first four values are not being referenced, so the window can be narrowed to width four, and the first four values of AA can be discarded. But these are the padded values; they're not needed! The loop can be rewritten:

```
<type> v2 = 0;
<type> v3 = 0;
<type> v4 = 0;
<type> v5 = 0;
<type> RR[:] =
    for window W[4] in A step (2) {
        <type> v0 = f (W[1:3]);
        <type> v1 = f (W[0:2]);
        ...
        next v5 = v3;
        next v4 = v2;
        next v3 = v1;
        next v2 = v0;
        } return (array (x));
<type> R[:] = RR[2:];
```

This leads to the question: Under what circumstances does the array padding become unnecessary? Sometimes the padding cannot be eliminated; note that if the original loop had, for example, one additional statement

```
<type> w = g (W[0:3]);
```

then the window could not be narrowed and the padding would have to remain. The next section looks at array padding in the case where nextification follows loop fusion, the most common situation under which nextification occurs, to see whether the cancellation of the array pad by the narrowed window should be expected to occur often.

4.2.2.2 Nextification after fusion

Though the nextification transformations are general in nature, not caring what other transformations might have set the stage for them, they are usually enabled by SA-C's loop fusion, since fusion creates redundant computations. The previous section showed that there are times when the array padding may be canceled by window narrowing. This section asks whether this can be expected to happen often in the loops that are created by loop fusion. To keep the discussion simple, a rank-one analysis is used.

Consider two rank-one loops before fusion, with window sizes c_U and c_L for the upper and lower loops, and step sizes s_U and s_L. As shown in section 4.1.4.2.1, the fused loop will have a size and step:

$$c = (c_L - 1) \times s_U + c_U$$

$$s = s_U \times s_L$$

Figure 4.18 shows an example from fused loops with the following generators:

```
for window W[7] step (3) ...
for window W[5] step (2) ...
```

where

$$c_U = 7, s_U = 3, c_L = 5, s_L = 2$$

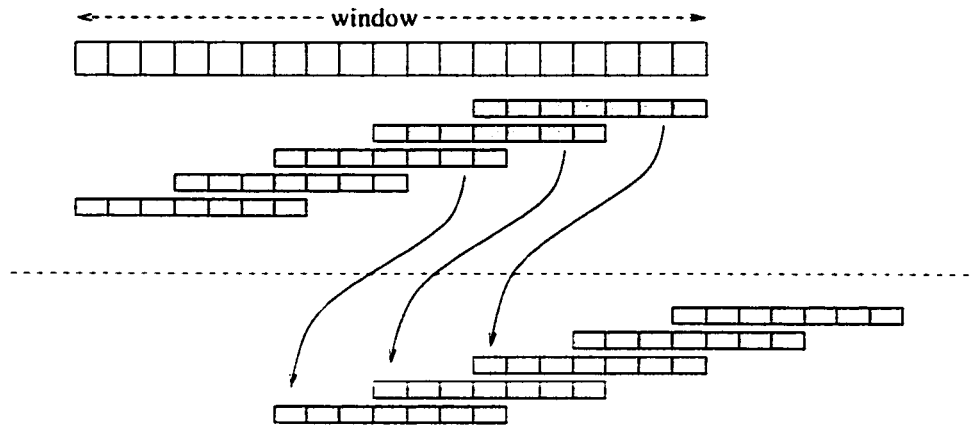


Figure 4.18: Example of a fused loop's window, with sub-arrays being taken. The sub-arrays below the dashed line show the sub-arrays for the next iteration.

The new window is 19 units wide, stepped by six. There are five width-7 sub-arrays being read from it, stepped by three. Because the sub-arrays are stepped by s_U and the new window is stepped by $s_U \times s_L$, there will be s_L sub-arrays that cannot be nextified. The figure shows when the window shifts right by six units, three of the sub-arrays are the same as those in the previous iteration; two are new.

Two adjacent slices are offset by s_U, and each slice has width c_U, so the s_L remaining slices take a total width of $s_U(s_L - 1) + c_U$. Since the fused window width is $(c_L - 1)s_U + c_U$, the number of unused window elements is

$$((c_L - 1)s_U + c_U) - (s_U(s_L - 1) + c_U) = s_U(c_L - s_L)$$

and window narrowing will eliminate them. This is the same number of elements that must be removed from the left of the source array after narrowing has occurred. In the example, there are nine unused window elements after nextification.

However, the source array must have been padded with enough elements to fill the registers with correct values. Since there are s_L slices left after nextification, each new iteration will create values in s_L nextified variables. The number of nextified variables is $c_L - s_L$, so it will take $\lceil \frac{c_L - s_L}{s_L} \rceil$ iterations to fill these variables. Since the loop's step size is $s_U s_L$, the number of padding elements is $s_U s_L \lceil \frac{c_L - s_L}{s_L} \rceil$. In the example, it will take two iterations to fill the three nextified variables, and

since the step size is six, twelve values must be padded onto the source array.

When c_L is a multiple of s_L, the number of padding elements is $s_U(c_L - s_L)$, which is the same as the number of elements removed by narrowing the window. In other words, window narrowing will exactly cancel the padding if and only if c_L is a multiple of s_L. Since unit step sizes are common (i.e. $s_L = 1$), it will often be the case that the source array can be used directly by the nextified loop without padding.

4.2.2.3 Nextification with loop reductions

Loop return node type CONSTRUCT_ARRAY and CONSTRUCT_TILE require throwing away the left edge of the result array, since it is composed of junk values. However, for reductions such as REDUCE_SUM there is no way to retract the unwanted values after the loop has run. Instead, the values have to be prevented from entering the reduction in the first place, and this can be done using the optional boolean mask that each reduction allows. Since the compiler "knows" the number of dummy iterations that will take place, it therefore knows how many values to mask at the beginning of each horizontal traversal. The window generator node in the DDCF graph produces outputs that convey the source array indices of the current window, and the compiler can connect the rightmost index output to a compare node that will produce the correct number of FALSE values.

Here is simple example:

```
int8 R0[:,:] =
    for window W[3,2] in A
        return (array (f (W)));
int8 R1 =
    for window W[2,2] in R0 {
        uint8 s = array_sum (W);
        } return (sum (s));
```

There is one dummy iteration, which caused a pad of one that was canceled by window narrowing. The following is the nextified loop:

```
int8 v00 = 0;
int8 v10 = 0;
int8 R1 =
    for window W[4,2] in A at (uint16 i, uint16 j) {
        bool b = (j>=1);
        int8 v01 = f (W[0:2,0:1]);
```

```
int8 v11 = f (W[1:3,0:1]);
int8 T[2,2] = {{v00,v01},{v10,v11}};
uint8 s = array_sum (T);
next v00 = v01;
next v10 = v11;
} return (sum (s,b));
```

The index value j is compared with 1, producing the boolean value b that is used in the **sum**.

4.2.3 Window size reduction

A window generator in a SA-C program may be larger than it needs to be. This can happen as a result of the nextification transformation discussed previously. If an outer row or column of a window generator is never referenced, the window size can be reduced to eliminate it.

This optimization is quite simple and involves two steps. First, the window size is reduced to eliminate the unused rows and/or columns. In the case of a left column or upper row, this also requires adjusting the indices of AREF nodes that reference the window to account for the change. The second step involves taking a slice of the input array and feeding its result to the generator. The need for this is obvious if a left column or upper row was eliminated, since a corresponding row or column must be removed from the source array as well. It is also necessary for right columns and bottom rows since the change in the generator size would otherwise change the number of iterations of the loop.

In this simple example, only the middle column of the window is being referenced:

```
<type> R[:,:] =
    for window W[4,3] in A {
        <type> S[:] = W[:,1];
            } return (array (array_sum (S)))
```

It can be transformed as follows:

```
_, uint32 d = extents (A);
<type> AA[:,:] = A[:,1:d-2];
<type> R[:,:] =
    for window W[4,1] in AA {
        <type> S[:] = W[:,0];
        } return (array (array_sum (S)))
```

The slice AA has trimmed the first and last columns from A. The window generator has been reduced

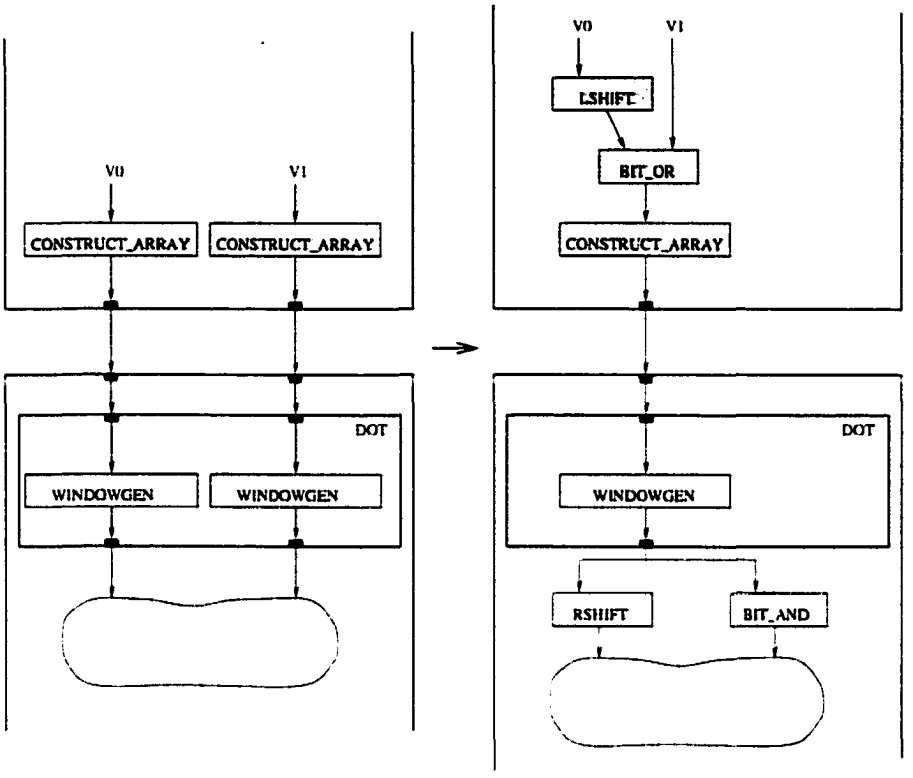


Figure 4.19: DDCF transformation that combines arrays in a producer/consumer loop pair.

to a single column, and the column index in the loop body's array reference has been adjusted by one to account for the loss of the left column.

4.2.4 Array combining using bit concatenation

Since SA-C does not have record data types, users are forced to use multiple array value returns in situations where an array of records might have been the more reasonable choice. However, sometimes it is feasible to combine the arrays created by the producer loop by bit-concatenating their values on an index-by-index basis. This one array is sent to the consumer loop, which unpacks each array value into its individual parts. This transformation is possible as long as the combined values do not exceed SA-C's scalar width limit of 32 bits.

The potential benefits are twofold. First, there may be less loop-return and generator overhead. Second, this is an easy way to set the stage for loop fusion, since the current fusion implementation is limited to handling a single array value passed between two loops. The implementation of fusion is

already the most complex transformation in the compiler, and doing array combining as a separate transformation should be easier than expanding the capability of the fusion transformation.

Figure 4.19 shows the DDCF transformation. It requires that the two window generators have the same access patterns, i.e. the same sizes and steps. Simple shifting and bit manipulations are all that are required.

4.3 Order of application and control of optimizations

Optimizations interact in complex ways, and since the interactions are program-dependent it is hard to anticipate how they will behave under the multitude of user programs they will encounter. Most of the optimizations in the SA-C compiler are formed into a sequence that is iterated until stability is reached. This cyclic sequence, described below, is used within a higher-level sequence as follows:

1. inline all functions except those designated for lookup table conversion
2. convert all designated lookup table functions
3. inline the functions that were converted to lookup tables
4. perform cyclic sequence until stable
5. perform array blocking
6. perform loop stripmining
7. perform cyclic sequence until stable

Inlining and lookup table conversion are done first since they are not dependent on any other code transformations. Inlining is performed on a function unless its definition has a **no_inline** pragma. Lookup tables are designated by **lookup** pragmas. After inlining, the cyclic optimizations are performed to do a general cleanup and to set the stage for the array blocking and loop stripmining transformations, which are controlled by user pragmas with parameters. After this, the cyclic optimizations are performed again, since these transformations create new opportunities for code improvement.

The cyclic sequence is as follows:

1. invariant code motion
2. array cast elimination
3. constant-expression switch elimination
4. array and loop size propagation
5. array value propagation
6. constant folding
7. algebraic identity simplification and strength reduction
8. dead code elimination
9. full loop unrolling
10. common subexpression elimination
11. array reference chain elimination
12. loop fusion

The order of these optimization is somewhat arbitrary, though some are placed after others because of the obvious opportunities they create. However, cycling reduces the concern about getting the order right: if a later optimization in the sequence sets up an opportunity for an earlier one, it will be picked up in the next pass. When window generator nextification and window size reduction optimizations are implemented they can be placed in this sequence after loop fusion. As optimizations take place, the compiler reports the inlining, lookup table conversion, fusion, unrolling, blocking and stripmining that takes place. It also reports the number of passes from each of the two cyclic optimization passes.

The cycle-until-stable strategy may cause concerns about termination, but non-termination is not possible in this set of transformations. Non-termination would imply either a boundlessly growing DDCF graph or else a return to a graph that had been previously encountered. Boundless growth is

not possible, since the only optimizations that can increase the number of nodes in a DDCF graph are loop fusion and loop unrolling, but both of these operate on loops and in the process reduce the number of loops in the graph. Hence growth must eventually stop for lack of new loops to transform. The rest of the optimizations either remove nodes or replace complex nodes with simpler ones. No transformation exists that could reintroduce the more complex nodes, so a cycle is not possible. As optimizations take place, the compiler reports the inlining, lookup table conversion, fusion, unrolling, blocking and stripmining that takes place. It also reports the number of passes from each of the two cyclic optimization passes.

Chapter 5

Translation of loops to DFGs

The DDCF graph of a SA-C loop that is to be executed on the RCS is transformed by the compiler to a low-level dataflow graph (DFG), which is a non-hierarchical form that is read by the system's DFG-to-RCS translator. This chapter describes the DFG form and the process by which a DDCF loop is transformed to a DFG.

5.1 Dataflow graphs (DFG)

Dataflow graphs are used internally by the SA-C compiler as a program representation that precedes the VHDL form and can be simulated using a token-driven semantics [23]. They represent a program in a low-level, non-hierarchical and asynchronous way. The graphs take into account sequencing but not timing; however, the node functions have been created in such a way as to allow reasonably straightforward translations into the synchronous circuits that finally will be mapped onto reconfigurable hardware.

The functional elements of dataflow graphs are *nodes*, each of which has a node-type, one or more input ports, and one or more output ports. The nodes of a dataflow graph are connected by directed edges, each of which connects an output port to an input port. An output port can connect to any number (including zero) of input ports, but an input port can be fed by only one output port.

When a dataflow graph executes, nodes are "fired" and data are communicated via *tokens*. Every token is created by an output port of some dataflow node. (Data are brought in from outside the dataflow graph through special INPUT nodes.) Each token represents a single k-bit untyped entity, where k is specified by the output port of its source. Each node type has a firing rule that specifies

its behavior when it fires. When a node produces a value at one of its output ports, a token of that value is sent to every input port that is fed by that output. Every input port has a queue of unbounded size. The order of the tokens at an input is the same as the order in which they were produced.

A node may fire only when its firing rule is met. If multiple nodes are ready to fire, they may fire in any order or concurrently. The behavior of most nodes is very simple: it fires only if each of its inputs sees a token. The act of firing consumes one token from each input, and the value of its output is a function of only those inputs. If an input needs a constant value, it is fed by the constant rather than by an output port. The constant input may be viewed as producing a token of that value whenever such a token is needed to fire its node. Tokens have no type, but each has a size as a number-of-bits. The bit-size of an output must match the sizes of each input it feeds. However, the size of a node's input does *not* have to match the other inputs or the outputs of its node. The exact behavior of a node whose inputs and outputs are of various bit-sizes depends on its node-type.

DFGs produced by the SA-C compiler also contain nodes with state and more complicated firing rules. All of these stateful nodes are derived from the language's generators and loop returns. The DFG also has one or more explicit addressable memories, meant to act as counterparts to the local memories on an RCS board. DFGs are nearly acyclic: only one node type, derived from a loop's nextified variables, can introduce back edges into a DFG.

5.2 Criteria for loop conversion

The SA-C compiler attempts to convert any bottom-level FORALL and FORNXT loops into dataflow graphs (DFGs) for execution on RCS hardware. (The user can prevent this by using a **no_dfg** pragma.) The following are the criteria to allow a loop to be translated into a DFG:

- The loop must be a FORALL or a FORNXT.
- No **float**, **double** or complex types are allowed, other than **double** outputs of intrinsic functions.

- Any AREF node that is fed from a generator must have constant indices.
- All AREF outputs must be scalar.
- An ARRDEF node is allowed only if it feeds only a CONSTRUCT_TILE node.
- A CONSTRUCT_TILE is allowed only if it is fed by a ARRDEF node.
- The generator graph must be DOT.
- No LOOP_INDICES nodes are allowed.
- Each WINDOWGEN node must have constant size inputs.
- Each array slice generator must have known slice dimensions.
- The following nodes may not be present within the loop: ARR_CONPERIM, ARR_CONCAT, EXTENTS, FUNC, FCALL, FORNXT, FORALL, WHILE, WHILE_PRED, DIV, MOD, CONSTRUCT_CONCAT, VAL_AT_FIRST_MAX, VAL_AT_FIRST_MIN, VAL_AT_LAST_MAX, VAL_AT_LAST_MIN, VAL_AT_MAXS, VAL_AT_MINS, PRODUCT, MEAN, ST_DEV, MODE, MEDIAN, HIST_MACRO, ACCUM_PRODUCT, ACCUM_MEAN, ACCUM_ST_DEV, ACCUM_MEDIAN

Some of these limitations exist simply because broadening the implementation has a low priority, considering the IP codes of current interest to the Cameron Project.

There are two steps in the conversion of a loop to a DFG. The first is to enclose the loop in a RC_COMPUTE node and surround the loop with host/RCS interface computations. The second is to translate the loop itself to a RC_FOR node by converting its graph to a DFG.

5.3 Interface graphs

The first step in converting a loop to a DFG is to surround it with host/RCS interface code, and to enclose this code and the loop in a RC_COMPUTE node. Within this node, but outside of the loop itself, the semantics change: memory addresses are exposed in this layer. The nodes here will

be translated into host code that handles such issues as transferring data to and from the RCS, and allocating memory on the RCS board.

There are four sets of information computed by the interface code and sent into the loop:

- Address and extents for each array that is referenced by the loop. A MALL_XFER node for each SA-C array allocates space for the array in the RCS memory, transfers the data into that memory, and outputs the address. An EXTENTS node for each array produces the extents.
- The target addresses for the loop's return nodes. These are produced by MALLOC_TARGET nodes.
- The loop's extents. These are computed with LOOP_EXTENT nodes whose input information is derived from the loop's generator nodes.
- The total number of iterations the loop will execute. These are computed by multiplying the outputs of the LOOP_EXTENT nodes.

The loop extents and iteration count are redundant, since the generator inputs have the information needed to determine these values, but they are brought in this way to keep the loop's graph as simple as possible.

There is also interface code at the output of the loop. After DFG conversion, a loop has exactly one trigger output. Each loop-return node gives rise to a TRANSFER_TO_HOST_ARRAY or TRANSFER_TO_HOST_SCALAR node, whose first input is a trigger signal from the loop's output. The second input is the result's address. Additional inputs for the TRANSFER_TO_HOST_ARRAY convey the array's extents.

As an example, figure 5.1 shows the interface code for the following SA-C loop:

```
for window W[2,3] in A
    return (array (f (W)))
```

5.4 Converting a loop to a DFG

A FORALL or FORNXT loop that meets the criteria for becoming a DFG is modified in a variety of ways during the loop's DDCF-to-DFG transformation. First, the graph's type is changed to

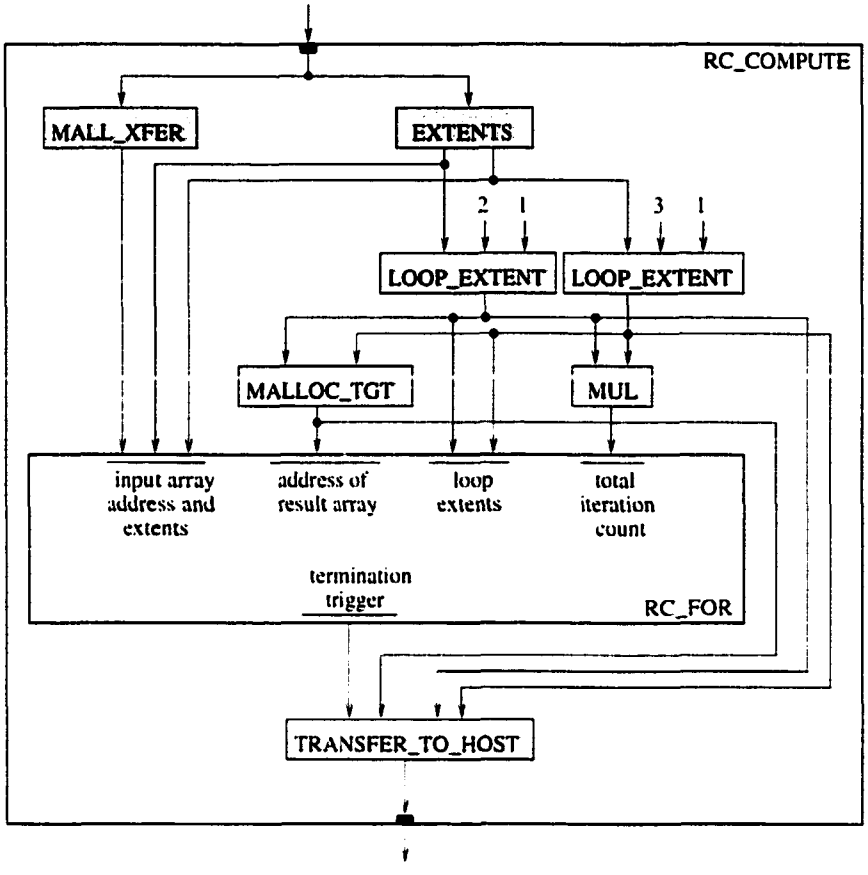


Figure 5.1: Example of DDCF interface code.

RC_FOR to indicate that it is a DFG and designated to run on the RCS. Internally the graph is flattened: the DOT graph and any SWITCH graphs are brought up to the top level, and all internal nodes are modified to a form suitable for DFGs. Various shift and mask operators are added to the graph to allow operations of various SA-C types to be performed using low level integer arithmetic operators. Any nodes that reference arrays in memory are given explicit edges that provide address and extents information. Loop-return nodes are converted to forms that explicitly provide target address and extents information. These nodes deliver termination trigger tokens when the node is done, and these triggers are ANDed to a single loop output. The edges in the DFG become typeless, having only bit-width attributes. These issues are now discussed in greater detail.

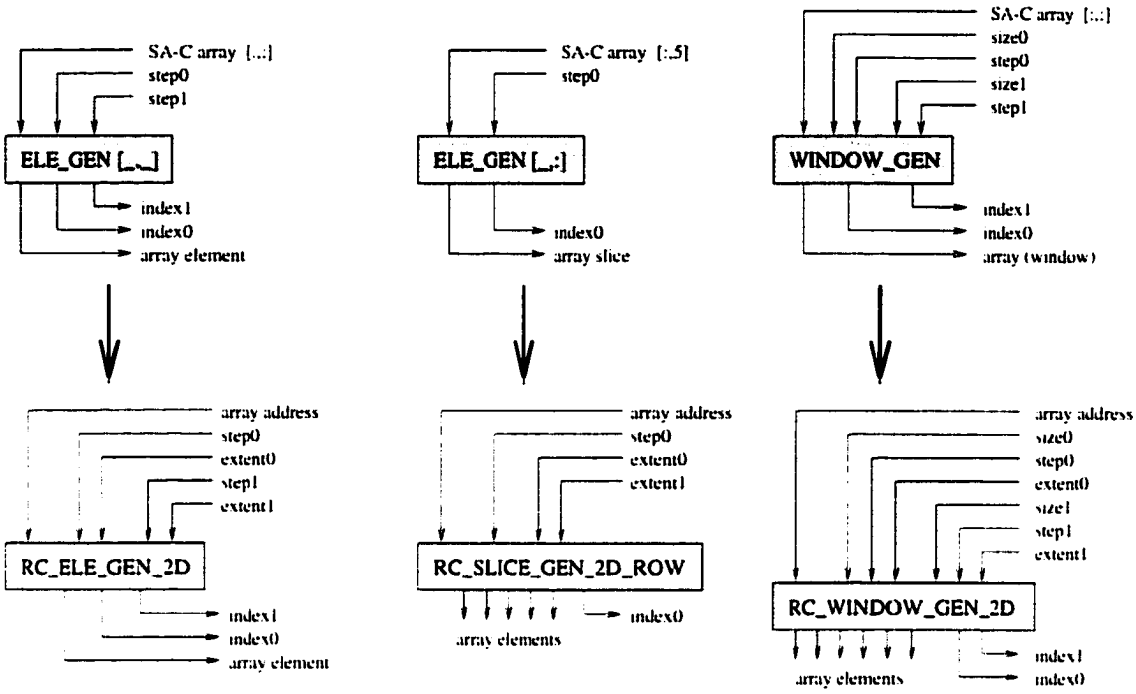


Figure 5.2: Example transformations of array-element, array-slice and window generators.

5.4.1 Generator conversion

Four kinds of generator may appear in the DOT graph of the loop before transformation: scalar, array-element, array-slice and window. The scalar generator does not require modification. Each of the other three is transformed into a node that has scalars for all of its outputs and has been given source array extent information in addition to the inputs it already had. Figure 5.2 shows an example of each.

5.4.2 Loop body conversion

Within the loop body all nodes undergo some modification. In the simplest case it is the removal of a SA-C type from its ports, leaving them untyped. However, many nodes require more changes. AREF nodes may appear in a loop body, and they require a transformation similar to that of the array-referencing generators: the array address and extents information is passed in as explicit edges.

Some nodes require more extensive work for DFG conversion: SWITCH nodes, various arithmetic operators, intrinsic function calls and NEXT nodes. These are now discussed in detail.

5.4.2.1 SWITCH node conversion

SWITCH graphs in the loop body are transformed into a flattened form in which all of the case subgraphs are brought to the top level and feed a SELECTOR node that uses the key expression to choose among the different case values. Figure 5.3 shows an example of this, corresponding to the SA-C expression

```
switch (key) {
    case -1    : return (E0)
    case 3,-7  : return (E1)
    case 2     : return (E2)
    default    : return (E3) }
```

The leftmost input of the SELECTOR node is the key value. Following it are <case-value, return-value> pairs. Finally, the rightmost input is the default return value. This approach to SWITCH conversion produces a DFG in which all cases of the SWITCH are computed, and the return value is selected according to the key expression. This somewhat unusual approach is appropriate for FPGAs, since there is nothing to be gained by preventing all of these values from being computed (the FPGA logic for all the cases must exist regardless) and there is no danger of system malfunction in doing so.

If a SWITCH graph has multiple outputs, a SELECTOR node is produced for each output value. Since SWITCHes can be nested, this SWITCH transformation routine is recursive.

5.4.2.2 Arithmetic node conversions

Since the arithmetic operators available in the DFG form are all integer operations, SA-C's fixed point operations must be converted to integer equivalents by inserting change-width (with or without sign-extension) and bit-shift (right or left) operators into the graph. For example, for the last line in the SA-C code

```
fix8.2 a = ...
ufix9.1 b = ...
...
fix16.4 c = a + b;
```

the DDCF graph and its transformation is shown in figure 5.4. In the DDCF graph the type of

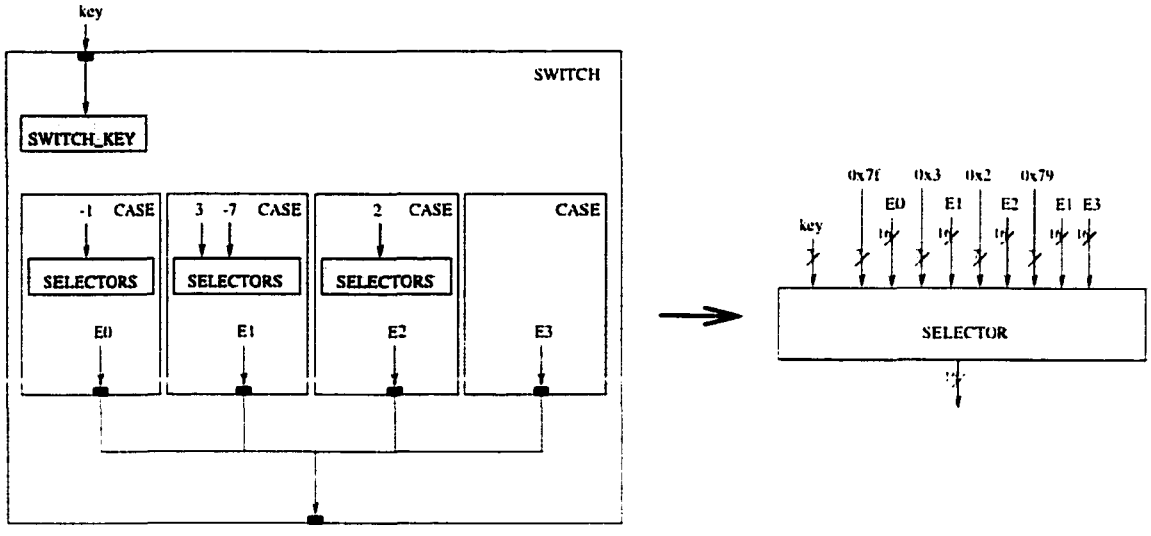


Figure 5.3: Example of SWITCH transformation.

the ADD node's output is determined by SA-C's semantics. The assignment to a **fix16.4** creates the CAST node. In the DFG form, the first CHANGE-WIDTH node adds a '0' bit on the left to convert the unsigned value to a signed value. The LEFT-SHIFT gives the value another fractional bit, to match the left operand of the ADD. (When two values are added, they must have the same number of fractional bits.) The ADD node itself is specified as IADD, i.e. a signed addition. The last two nodes are generated by the CAST: the LEFT-SHIFT gives the value four fractional bits, and the CHANGE-WIDTH-SE is a sign-extended widening of the value to sixteen bits.

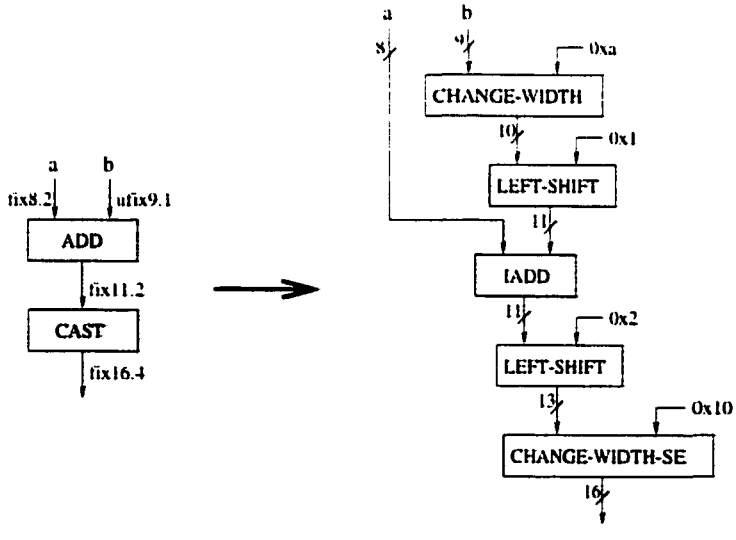


Figure 5.4: Example of arithmetic transformation.

A multiply node (MUL) in a DDCF graph is given special handling, due to the inherent inconsistency between SA-C semantics and the behavior of multiplication circuitry in hardware. To perform a multiplication at a specific bit width, the SA-C programmer often must cast one of the operands to that width. For example, if a **uint7** variable x and a **uint5** variable y are multiplied to yield a **uint12** result, the programmer might write

```
(uint12)x * y
```

However, hardware multiplication circuits typically produce outputs whose bit widths are the sums of their input widths. If the programmer's type cast stays in place after transformation to hardware, the multiplication uses more hardware than it needs.

To solve the semantic/hardware multiplication inconsistency, a new DDCF node has been introduced, called MUL-MACH, which has semantics like that of a hardware multiply, i.e. it produces a result whose width is the sum of its input widths. As a loop body is transformed into a DFG, the MUL nodes are converted to MUL-MACH nodes in three steps:

1. If one input is signed and one input is unsigned, then add a CAST node to the unsigned input to convert it to a signed value. However, if the unsigned input is already fed by a CAST node, insert the new CAST *above* the existing CAST node.

2. If one of the inputs is fed by a CAST node, and the CAST node is not a sign converter, and the CAST does not destroy bit information (i.e. it is a widening CAST), then remove the CAST node, transferring its input type to the input port of the MUL node.

3. Convert the MUL node to a MUL-MACH node, by making its output bit width the sum of its input widths, and its output fractional bit width the sum of its input fractional bit widths. Insert a CAST node below the MUL-MACH to convert the type back that which was originally on the MUL node's output.

The MUL-MACH node is then converted, in a straightforward way, to a UMUL or IMUL node in the DFG. Figure 5.5 shows four examples of MUL conversions. The left forms are the DDCF sub-graphs before transformation, for the following four SA-C expressions:

```
1:      uint7 a = ...
        uint5 b = ...
        uint12 c = (uint12)a * b;

2:      int7 a = ...
        uint5 b = ...
        int12 c = a * (int12)b;

3:      uint7 a = ...
        uint5 b = ...
        uint7 c = a * b;

4:      ufix8.5 a = ...
        ufix8.3 b = ...
        ufix16.1 c = a * (ufix16.1)b;
```

The right forms are the resulting DFG sub-graphs, and some intermediate forms are shown between the starting and ending forms.

In the first example, the user has done a cast that will cause the multiplication to take place in twelve bits. The CAST node does not lose bit information, since it is casting from seven to twelve bits, so step two of the conversion eliminates the upper CAST. Step three puts a CAST node on the output, but since it casts a **uint12** to a **uint12**, the cast is removed by a subsequent clean-up pass. The DFG at right is exactly what the user wanted.

The second example shows a mix of signed and unsigned inputs. Step one adds a CAST node above the existing CAST, as a sign transformer. Casting a **uint** to an **int** always requires adding a bit for the new sign, so the output of the new CAST has the type **int6**. Step two then removes the original CAST node, and step three puts a CAST node under the MUL-MACH. This CAST stays because its input and output types are not the same. The CAST nodes are converted to CHANGE-WIDTH nodes in the DFG; the second one is called CHANGE-WIDTH-SE for sign-extending. (A sign-extending node is always used when a signed value is an input, even though in this case the node is truncating bits and there are no bits into which the sign can be extended.)

The third example shows a MUL in which there are no casts on the inputs. This situation can arise if the user knows that the combinations of the two operands will never produce a result greater than $2^7 - 1$, i.e. 127. It is not possible to narrow one of the inputs: note that two pairs of inputs,

$< 1, 31 >$ and $< 127, 1 >$, produce results that fit in the **uint7** output type, yet all of the input bits are needed. In the conversion, step three puts a CAST node after the MUL-MACH to narrow the **uint12** result back to the correct **uint7** type.

The fourth example has fixed point types. The DDCF graph has two CAST nodes, one from the explicit cast on b and the second an implicit cast from the assignment to c. Note that the type of the MUL node's output is **ufix20.5**, determined by SA-C semantics. The upper CAST is not removed, because it loses bits, and eliminating it would allow bits to be multiplied that could change the result. Step three puts a CAST node after the MUL-MACH node, casting the type back to that of the original MUL node's output, i.e. **uint20.5**. In conversion to DFG, the CAST nodes become right shifts since they are truncating fractional bits.

5.4.2.3 Intrinsic function conversion

The only intrinsic function currently implemented with regard to DFG transformation is **sqrt**. The SQRT operator in a DFG is assumed to work on an unsigned integer input with an even number of bits, returning a value with a bit-width of half the input width. Since the intrinsic **sqrt** call returns a **double**, a DFG can only be produced if the call's output is cast to an integer or fixed point type. The transformation must then generate code that shifts the input so that the output of the SQRT produces the correct value; specifically, the fractional number of bits of the input must be twice that of the output. Figure 5.6 shows an example that comes from the last statement of

```
ufix8.1 a = ...
ufix8.4 v = sqrt (a);
```

Since the output fractional size is four bits, the input value must have eight fractional bits, requiring a left shift of seven bits. Since this yields a total bit-width that is odd, a zero is padded on the left by the CHANGE-WIDTH node.

5.4.2.4 NEXT node conversion

When a loop is converted to a DFG, each nextified variable receives its value from a previous iteration. CIRCULATE nodes manage this; they are the only source of back edges in a DFG. Nextified variables are converted to CIRCULATE nodes as shown in figure 5.7. The final value of

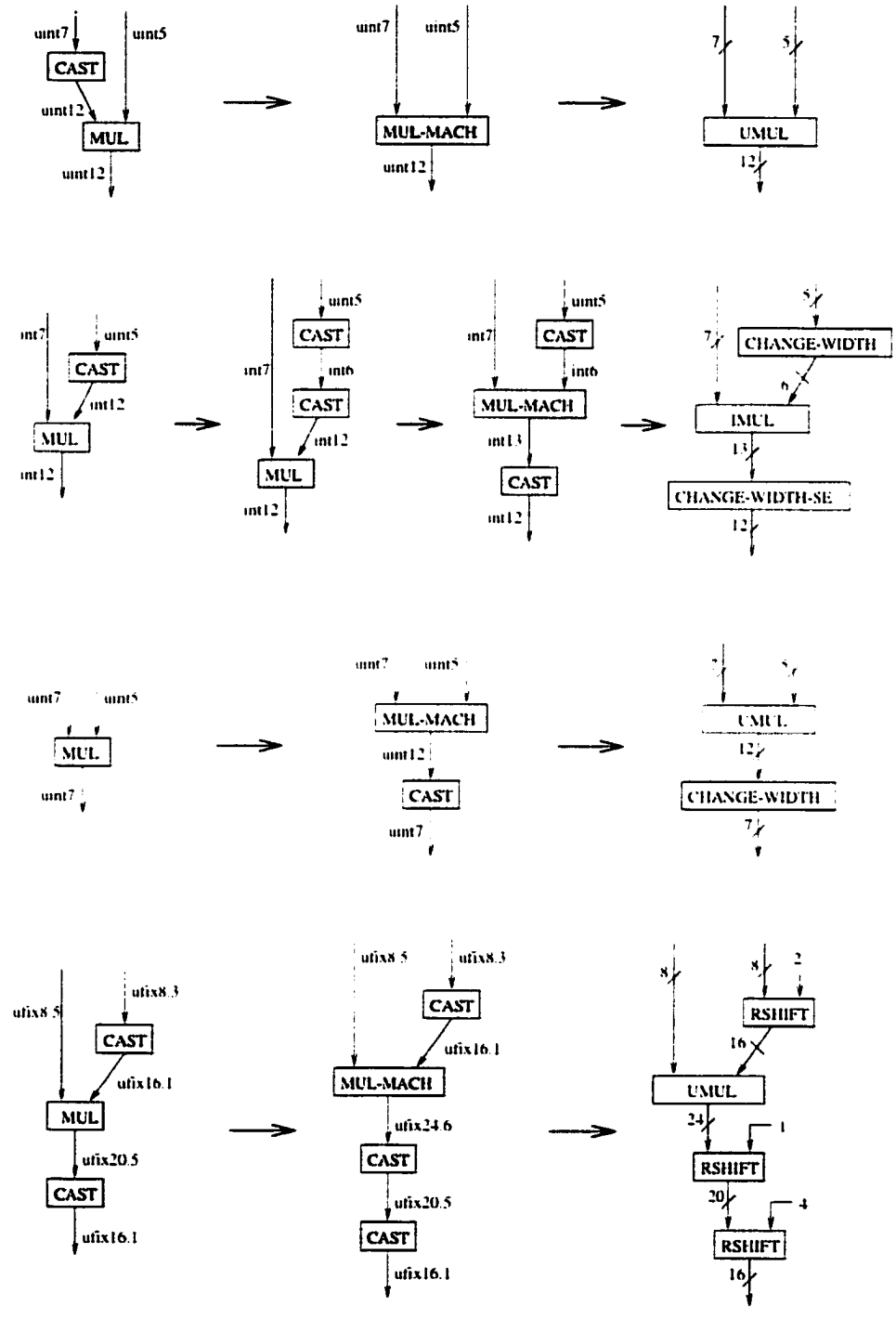


Figure 5.5: Four examples of multiplication transformations.

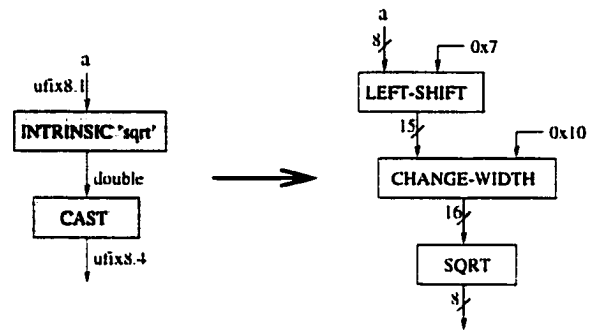


Figure 5.6: Example of sqrt transformation.

the variable emerges from a loop output of the DDCF graph if it was called for in the SA-C program. When converted to a DFG it produces a WRITE_SCALAR node that stores the final result and delivers a trigger token as described below for other loop return operators.

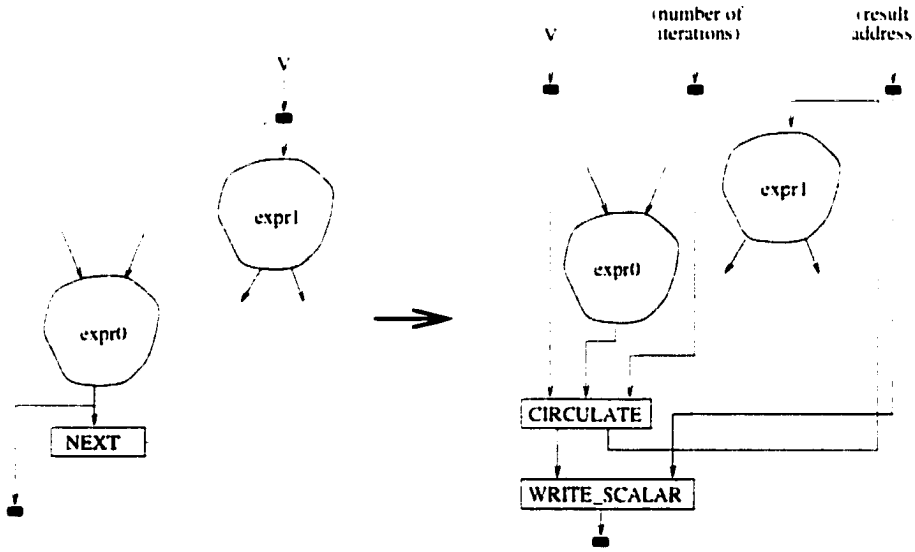


Figure 5.7: Example of nextified variable transformation.

5.4.3 Loop-return conversion

During DFG conversion, loop return nodes produce corresponding nodes that have the information needed to specify where the result is written in target memory, and how many values will be streaming into the node. CONSTRUCT_ARRAY and CONSTRUCT_TILE nodes give rise to WRITE-TILE nodes with value inputs, a base memory address, and the loop's extents. A CONSTRUCT_TILE node must be fed by an ARR_DEF node.

WRITE_TILE nodes exist is a variety of forms according to the rank of the input tile and the rank of the loop itself. (In SA-C the tile and loop ranks need not be the same.) For example, a WRITE_TILE_2D_3D node takes rank-two tiles and lives in a rank-three loop. The return node needed for a CONSTRUCT_ARRAY node is simply a special case of a WRITE_TILE in which the "tile" is a one-element array.

Figure 5.8 shows an example from the loop return

```
for ... {
    uint8 T[2,2] = {{a,b},{c,d}};
    } return (tile (T))
```

where the loop is rank-three. This kind of situation arises when a loop has been stripmined.

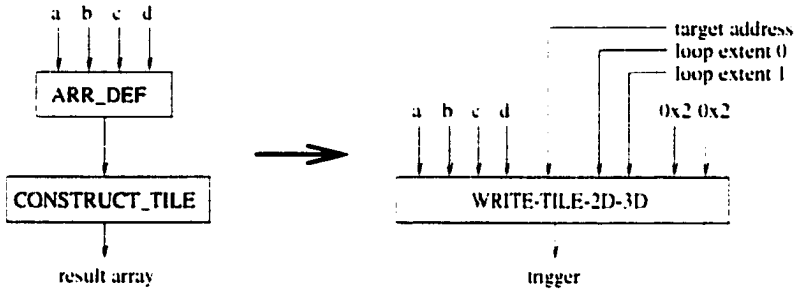


Figure 5.8: Example of CONSTRUCT_TILE transformation.

Loop reduction nodes have inputs for streams of values and their boolean masks, as well as inputs the result target address and the number of values being reduced. Figure 5.9 shows, for example, the conversion of a SUM return. Other loop return nodes require their own pieces of information. The HISTOGRAM reduction, as well as the various ACCUM-type reductions, require not only the values, masks, address and iteration count, but also information about the shapes of the arrays they write.

Since the converted loop-return nodes deliver their results by writing values to memory, their output tokens are simply triggers that indicate that the node is done and the results have been written. If there are multiple loop returns, these triggers are combined using an AND-MANY node.

5.4.4 A complete example of loop conversion

Figure 5.10 shows the result of DFG conversion for the loop in the following piece of SA-C code:

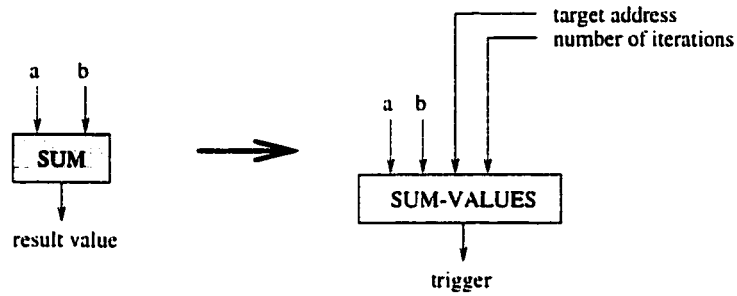


Figure 5.9: Example of SUM transformation.

```
ufix10.4[:,:], ufix20.2 main (ufix8.2 A[:,:]) {
    ufix10.4 R0[:,:], ufix20.2 R1 =
        // PRAGMA (stripmine (4,2))
        for window WA[3,2] in A {
            ufix11.2 S = array_sum ((ufix11.2[:,:])WA);
            } return (array ((ufix10.4)(sqrt (S))),
                             sum (((ufix20.2)WA[1,0]+WA[1,1])/2));
    } return (R0, R1);
```

The boolean mask values for the xxx_MANY nodes are omitted for clarity: in this example they are all '1', i.e. true. The window generator is a 4x2 with step (2,1), produced by the stripmining. The source array's address and extents feed the generator, along with the window sizes and steps. The eight CHANGE_WIDTH nodes (abbreviated "CH_WIDTH" in the figure) are the result of the cast in the **array_sum** operator. The RSHIFT nodes are the result of the division by two. The inputs of the USUM_VALUES node are the value, the mask, the target address and the number of values being reduced. The inputs for the WRITE_TILE_2D_2D node are the values (two), the target address, the loop extents (two) and the tile extents (two).

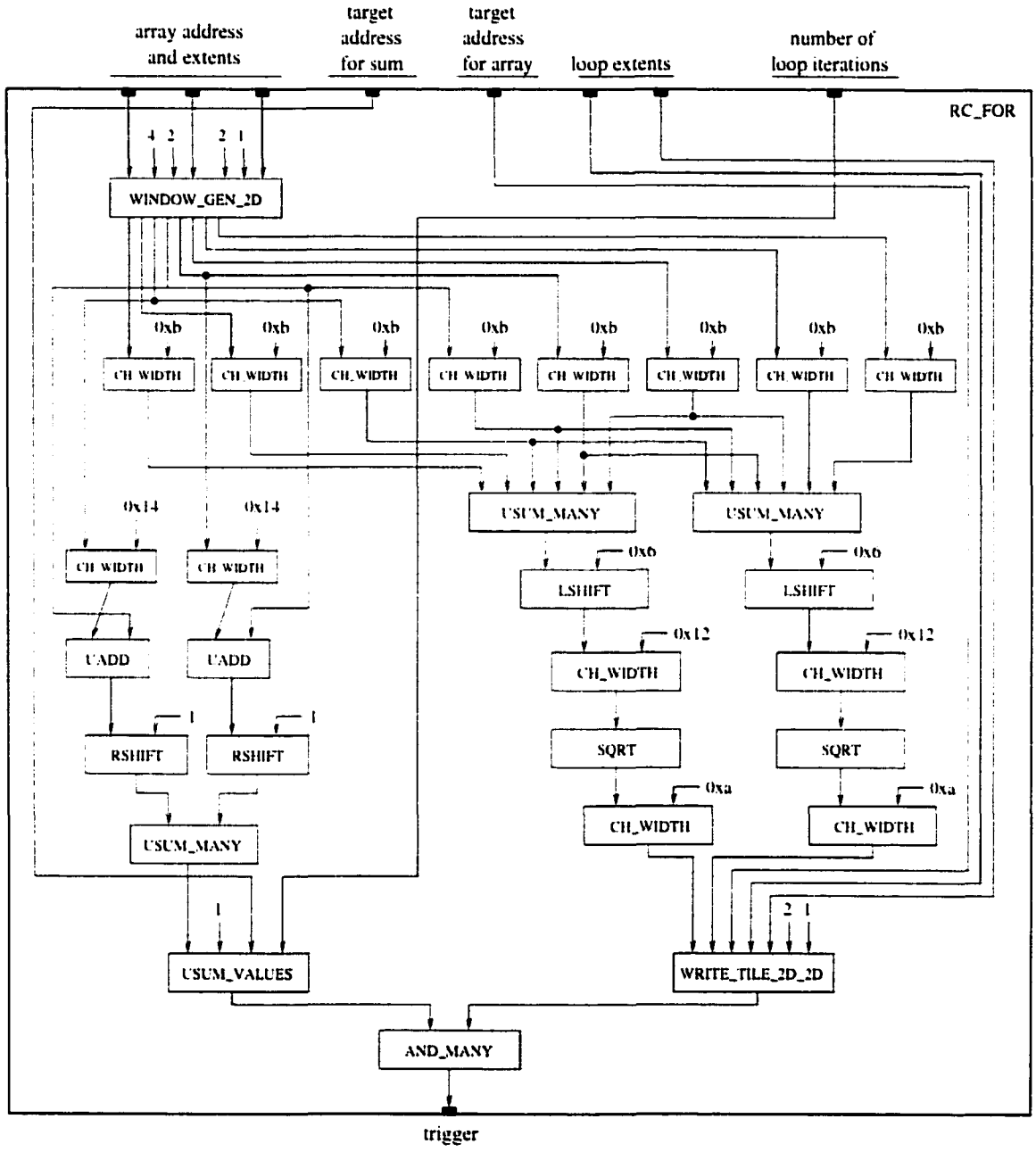


Figure 5.10: Example of FORALL loop after DFG conversion.

Chapter 6

Code generation

The code generator reads a DDCF file as input and produces a C file as host code and a DFG file for each RC_FOR node in the DDCF graph. The SA-C run-time system (RTS) is tightly linked to code generation, since the code generator produces calls to various RTS routines. In the SA-C compilation system, DFGs can be executed either by a host-based simulator or on actual RCS hardware. However, the code generator does not make a distinction between these, generating identical code in either case. Instead, a small set of function calls have dual implementations in the RTS, and the decision with regard to simulation or RCS execution is controlled by linking in one of two libraries.

6.1 Host code generation

Host code generation produces C code from the DDCF graph and the interface part of each RC_COMPUTE node.

6.1.1 Internal C data structures

The RTS has C typedefs used to declare variables that hold the various kinds of SA-C data. Scalar values from one to eight bits wide are stored as **Uint8** and **Int8** values (for unsigned and signed, respectively). Similarly, values from nine to sixteen bits are stored in **Uint16** and **Int16** values, and values larger than sixteen bits are stored as **Uint32** and **Int32** values. While SA-C does not support explicit scalar types larger than 32-bits, 64-bit entities are sometimes needed in the C code

to implement SA-C's semantics correctly, so **Uint64** and **Int64** types are also available. Finally, **Float32** and **Double64** are defined for SA-C's **float** and **double** types.

The RTS also has typedefs for SA-C's complex and array types. The complex ones are simply pairs of <real, imaginary> values, with different typedefs existing for each of the scalar data types that are allowed in SA-C's complex values. The array typedefs follow the pattern:

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

Most of these fields have obvious purposes. The mults array and the base field hold multipliers and base address for dope vector access. The mem field is the allocated memory address, used if a call to free deallocates an array. There is an array typedef for each of the scalar and complex storage types.

6.1.2 Generating host code for SA-C functions

A C function is produced for each SA-C function. Each SA-C parameter gives rise to a C function parameter of one of the above types. Since C does not give convenient support for multiple return values from a function, function returns are handled via global variables. This is possible because SA-C is non-recursive.

The C code for SA-C's arithmetic operations implements SA-C semantics, including the bit-widths of the operands and the shifting necessary to handle fixed point arithmetic properly. Signed operands are sign-extended to 32-bits before an arithmetic operation takes place. For multiplication of fixed point values, the operands are cast to 64-bits, the multiplication is performed, and the result is then shifted and masked as appropriate for the specified result type. For division, the numerator is shifted by a distance that will produce the desired result type after the division. For example, if variables a and b are SA-C type **ufix32.16**, they are stored in C language variables of type **Uint32**. The expressions $a \times b$ and a/b are defined by SA-C semantics to have a result type of **ufix32.16**,

and are computed by the C expressions

```
(Uint32)(((Uint64)a*(Uint64)b)>>16)
(Uint32)(((Uint64)a<<16)/b)
```

SA-C **for** loop simple generators give rise to loop nests in C, where the number of loops is equal to the rank of the generator. Simple generators that produce sub-arrays, i.e. window and array-slice generators, have different implementations depending on whether or not dope vectors are being used. Without dope vectors, a temporary array is allocated for the window or slice, and in each iteration the sub-array's values are copied into it. With dope vectors, the windows and slices are formed by manipulating the sub-array's base address and multipliers. Dot products are implemented by putting additional index variables in the **for** loops. Cross products give rise to a second level of loop nesting.

There are various approaches to implementing **for** loop return operators:

- The scalar reductions **sum**, **product**, **min**, **max**, **and** and **or** accumulate directly into a scalar variable.
- The **mean** reduction sums into one scalar, and counts values in a second scalar. The division takes place after the loop is done.
- The **st_dev** reduction is similar to **mean**, but maintains an additional scalar in which the sum of the squares is gathered.
- The **median** reduction allocates a temporary array in which the values are collected. After the loop, the values are sorted and the median is extracted.
- The **histogram** reduction allocates the result array, and increments its values as the loop runs.
- The **vals_at_maxs**, **vals_at_mins**, **vals_at_first_max**, **vals_at_last_max**, **vals_at_first_min** and **vals_at_last_min** reductions allocate temporary arrays in which the key values and the potential return values are collected. After the loop, the results are extracted.

- The **accum** reductions of **sum**, **product**, **min**, **max**, **and** and **or** allocate the result arrays and accumulate directly into them.
- The **accum** version of **mean** allocates two arrays for the sums and the counters. The divisions take place after the loop.
- The **accum** version of **st_dev** is similar to that of **mean**, but a third array is allocated for the sums of squares.
- The **accum** version of **median** allocates two arrays, one to collect the values and another to collect the labels. After the loop is complete, the values are partitioned by label, and each group is sorted so that the median can be selected and returned.
- The **accum** version of **histogram** allocates the rank-two result array, and the appropriate values are incremented as the loop runs.
- The **array** return allocates the result array, and collects values into it as the loop runs. The values are generated in order, so an incremented pointer is used to store the values.
- The **tile** and **concat** operators collect a temporary array of arrays. After the loop finishes, the result array is allocated and the values are filled in. This approach is necessary in the general case since the sizes of the array components being gathered are not always known when the loop head is reached during execution.

The return operators for **while** loops are similar, but for the **array**, **tile** and **concat** operators the size of the array, or array-of-arrays, is not known when the loop head is reached, so dynamically growing arrays are implemented.

As an example, figure 6.1 shows the C code generated for the following SA-C file:

```
int16 g (ufix16.4);
uint8, int16[:] f (ufix16.4 A[:], bool M[:]) {
    uint8 R0, int16 R1[:] =
        for a in A dot m in M {
            int16 v = g (a);
            uint8 w = a;
            } return (max (w, m), array (v));
    } return (R0, R1);
```

Lines one and two are function prototypes for functions g and f. Lines three through five give types for the return variables of all functions. Lines six and seven declare the return variables for the function f that is defined in this file. The function f, named "SC_f" in the C code, begins on line nine and has two arrays as its parameters. The code corresponding to the SA-C loop begins on line 17, first creating size and extent information. Lines 17 through 23 come from the first generator, and lines 24 through 30 come from the second generator. To keep the code generation process straightforward, each simple generator in a DOT product produces all possible extent and iteration variables, even though some may not be needed. The C compiler is relied upon to eliminate unused variables and dead code.

Lines 34 through 40 create the loop's return variables. The create_mults macro initializes the multipliers in the dope vector. The tests on lines 42 and 43 check that the two generators have the same extents. Lines 47 and 48 extract array elements from A and M. Function g, which in the C code is called "SC_g", is called on line 49 and its return value is captured on line 50. The max value is updated on line 52 and the array value is written on line 53. Finally, the function's two return values are written to its global variables on lines 55 and 56.

6.2 Run time system (RTS) and input/output (I/O)

The run time system for SA-C consists of three parts: general purpose macros that implement various useful operations for the host code, input/output routines, and interface functions that communicate between host and RCS (or DFG simulator).

Host code-related helper macros and functions exist for such things as array references, bounds checks, array copying and array concatenating. Many routines are related to the various loop return operators. Also, the dynamically-resizing arrays used to implement some `while` loop returns are implemented in the RTS.

For a SA-C program, the parameters of the function "main" represent the program's inputs, and its return values represent the program's outputs. Input/output for SA-C happens in one of two

```
 1:    void SC_g (Uint16);
 2:    void SC_f (ArrayUint16, ArrayUint8);
 3:    extern Int16 SC_g_ret0;
 4:    extern Uint8 SC_f_ret0;
 5:    extern ArrayInt16 SC_f_ret1;
 6:    Uint8 SC_f_ret0;
 7:    ArrayInt16 SC_f_ret1;
 8:
 9:    void SC_f (ArrayUint16 T0, ArrayUint8 T1) {
10:      Uint16 T2;
11:      Uint32 T3, T4;
12:      Uint8 T5;
13:      Uint32 T6, T7;
14:      Int16 T8;
15:      Uint8 T9, T10;
16:      ArrayInt16 T11;
17:      {{ Int32 i0;
18:         Uint32 li0;
19:         Uint32 sz0 = T0.extents[0]<=0?0:1+(T0.extents[0]-1)/(1);
20:         Int32 i1;
21:         Uint32 li1;
22:         Uint32 sz1 = T0.extents[1]<=0?0:1+(T0.extents[1]-1)/(1);
23:         Uint32 sz_gen0 = sz0*sz1;
24:         Int32 i2;
25:         Uint32 li2;
26:         Uint32 sz2 = T1.extents[0]<=0?0:1+(T1.extents[0]-1)/(1);
27:         Int32 i3;
28:         Uint32 li3;
29:         Uint32 sz3 = T1.extents[1]<=0?0:1+(T1.extents[1]-1)/(1);
30:         Uint32 sz_gen1 = sz2*sz3;
31:
32:         Uint32 sz = sz_gen0;
33:
34:         Int16 *p1;
35:         T10 = 0;
36:         SC_malloc (2, Int16, sz, p1)
37:         T11.base = T11.mem = p1;
38:         T11.size = sz;            T11.rank = 2;
39:         T11.extents[0] = sz0;     T11.extents[1] = sz1;
40:         create_mults (T11, 2)
41:
42:         dot_gen_size_test (sz0, sz2)
43:         dot_gen_size_test (sz1, sz3)
44:
45:         for (i0=0,i2=0,li0=0; i0<T0.extents[0]; i0+=(1),i2+=(1),li0++)
46:           for (i1=0,i3=0,li1=0; i1<T0.extents[1]; i1+=(1),i3+=(1),li1++) {
47:             get_array_ele2(T2, T0,i0,i1,5,"tst731.sc","f");
48:             get_array_ele2(T5, T1,i2,i3,5,"tst731.sc","f");
49:             SC_g (T2);
50:             T8 = SC_g_ret0;
51:             T9 = ((T2>>4)&255);
52:             if (T5 && T10<T9) T10 = T9;
53:             *(p1++) = T8; }}}
54:
55:      SC_f_ret0 = T10;
56:      SC_f_ret1 = T11;
57:      }
```

Figure 6.1: Generated C code for function shown in text.

modes: via ASCII stdin/stdout for easy use during program testing, and via command line-specified arguments for other environments. For the latter, there are two file formats: ASCII and binary.

For ASCII inputs in either mode, a full parser exists to read values and fill in specified data structures. The C function "main," created by the code generator, allocates a data structure for each of the program's inputs. The data structure is filled in with information specifying the SA-C type of the input (including its total and fractional bit-widths), its rank, and a pointer to the location to which the data should be stored. The input values are interpreted according to the SA-C type. For example, an input string "1" is a valid value for every `uint`, `int`, `ufix`, `fix`, `float` and `double` SA-C type, but the bit pattern that is stored will vary based on the type.

For binary I/O, a file format has been defined that is a superset of the PGM/PPM image format. SA-C type information is placed in lines that are interpreted as comments by the image format. In the absence of such type information, the SA-C input routine will assume the input values to be of type `uint8`, as a 2D array (for PPM) or 3D array (for PGM). This allows SA-C to use and generate image files with minimal manual intervention.

There are seven Host/RCS interface functions that set up the RCS hardware and transfer data between host and RCS. These functions have two implementations, one for the real RCS and one for DFG simulation. The functions are described:

- The **initialize** function is called once at the beginning of the program, to put the hardware in a reset mode. In the simulation implementation, this function forks the simulator process and connects it with the running program with a pair of pipes.

- The **download_configurations** function transfers a specified set of FPGA configurations to the RCS. In the simulation implementation, this function tells the simulator to load a specified DFG file.

- The **write_arg** function writes a single 32-bit DFG argument to a specified location in RCS source memory.

- The **mem_write_input** function is used to transfer a block of data from host memory to the

RCS source memory.

- The **execute_dfg** function starts the RCS computation and waits for a termination signal.
- The **mem_read_results** function transfers a block of data from RCS memory back to the host.
- The **end_dfg_process** releases the RCS. In the simulation implementation, this function tells the simulator process to terminate.

The implementations of most of these routines are simple. For example, the **write_arg**, **mem_write_input** and **mem_read_results** functions make calls to routines supplied by Annapolis, the maker of the RCS board.

The **download_configurations** function is designed to minimize the time spent in programming the FPGAs. This is important since the FPGAs in the current RCS are programmed in a bit-serial way. This function keeps track of each configuration that is used by the executing program, and also keeps track of which configuration is currently loaded in the FPGAs. Each time a **download_configurations** call is made, the configuration is downloaded unless it is already in the FPGAs. If it is the first call for the particular configuration, then the file is read and loaded into a memory buffer before being downloaded to the FPGAs.

6.2.1 Host code generation for RCS calls

When an executing host program is about to call a DFG to be run on the RCS, it must execute code that has been generated from the RC_COMPUTE node described in section 5.3. This includes doing memory allocation for the RCS memory.

RCS memory is 32-bit word addressed. To conserve RCS memory space and reduce the number of FPGA reads and writes, the arrays written to RCS memory are packed into 1-, 2-, 4-, 8-, 16- or 32-bit chunks. Also, to make array element extraction easier for the FPGAs, multidimensional arrays are word aligned, i.e. each row is padded so that the next row begins on a word boundary. Since the host data are not stored in this way, the interface host code must pack and word align when data are sent down, and perform the reverse operations when data are read back. This incurs

no cost, since the DMA transfers require the use of separately allocated memory, hence copying, anyway.

An example will illustrate the issues involved in code generation. Consider the following SA-C loop, with its loop compiled to run on the RCS:

```
uint8[:,:] main (uint8 A[:,:]) {
    uint8 R[:,:] =
        for window W[2,3] in A
            return (array (array_sum (W)));
    } return (R);
```

This is the same example whose interface code is shown in figure 5.1. Figure 6.2 shows the C code generated to call this loop. The variable `src_alloc` holds the address of the start of free RCS memory. The inputs of the DFG are stored as 32-bit values in the bottom memory addresses. Since this loop has seven input ports, the `src_alloc` is set on line 1 to start just after the first seven words. Lines 2 through 8 allocate space for the array A and transfer the data to the RCS memory after writing it to DMA memory with word alignment. The free memory address is set to the free word following the array. Lines 9 through 13 capture and/or compute the array's extents, the loop's extents and the loop iteration count. Lines 14 through 21 compute the size of the return array and allocate space in RCS memory for it to be written.

Line 22 downloads the FPGA configurations. Lines 23 through 29 write the seven DFG arguments to the first seven memory addresses on the RCS. Line 30 tells the RCS to execute; here the host waits for a termination signal. Lines 31 through 51 allocate host space for the result array, as well as DMA memory for the data transfer from RCS to host. Line 49 does the data transfer, and line 50 copies the data to the allocated space, removing word alignment and packing.

```
 1:    src_alloc = 7;
 2:    { ArrayUint8 Tmp;
 3:      Uint32 tfr_sz;
 4:      copy_array_to_word_align2 (Tmp, T0, Uint8, SC_DMA_malloc, tfr_sz)
 5:      mem_write_input (src_alloc, (void*)(Tmp.mem), tfr_sz);
 6:      SC_DMA_free (Tmp.mem);
 7:      T1 = src_alloc;
 8:      src_alloc += tfr_sz; }
 9:    T2 = T0.extents[0];
10:    T3 = T0.extents[1];
11:    T4 = ((Int32)T3-(3))<0?0:1+(T3-(3))/(1);
12:    T5 = ((Int32)T2-(2))<0?0:1+(T2-(2))/(1);
13:    T6 = ((T5*T4));
14:    { Uint32 sz, sz0, sz1;
15:      sz0 = T5;
16:      sz1 = T4 / 4;
17:      if (T4 & 0x3)
18:        sz1++;
19:      sz = sz0 * sz1;
20:      T7 = src_alloc;
21:      src_alloc += sz; }
22:    download_configurations (0);
23:    write_arg (0, (Uint32)T1);
24:    write_arg (1, (Uint32)T2);
25:    write_arg (2, (Uint32)T3);
26:    write_arg (3, (Uint32)T7);
27:    write_arg (4, (Uint32)T5);
28:    write_arg (5, (Uint32)T4);
29:    write_arg (6, (Uint32)T6);
30:    execute_dfg (use_viewer, 0);
31:    { Uint32 adr = T7;
32:      Uint32 sz = 1;
33:      ArrayUint8 Tmp;
34:      Uint32 sz0, sz1, tfrsz;
35:      sz0 = T5;
36:      sz *= T5;
37:      sz1 = T4;
38:      sz *= T4;
39:      sz1 = (sz1 & 0x3) ? 1 + sz1/4 : sz1/4;
40:      Tmp.mults[0] = sz1 * 4;
41:      Tmp.mults[1] = 1;
42:      tfrsz = sz0 * sz1;
43:      SC_DMA_malloc (42, Uint8, 4*tfrsz, Tmp.mem)
44:      Tmp.base = Tmp.mem;
45:      Tmp.size = sz;
46:      Tmp.rank = 2;
47:      Tmp.extents[0] = T5;
48:      Tmp.extents[1] = T4;
49:      mem_read_results (adr, (void*)(Tmp.mem), tfrsz, 1);
50:      copy_array2 (T9, Tmp, Uint8, SC_malloc)
51:      SC_DMA_free (Tmp.mem) }
```

Figure 6.2: C code generated for RCS call of loop described in text.

Chapter 7

Performance measurements

This chapter discusses the performance implications of the SA-C compiler's optimizations. These optimizations fall into two broad categories: graph simplifying and loop restructuring. Graph simplifying optimizations are geared toward lowering the count and/or complexity of the nodes in a graph, therefore saving both space (in FPGA logic) and compute time by reducing the length of the graph's critical path. Loop restructuring optimizations are oriented toward changing host/RCS code partitioning (to move more of the computation onto the RCS) and altering data communication patterns, both between the host and RCS, and between the RCS local memories and the FPGAs. It should be noted that the simplifying optimizations sometimes set the stage for the restructuring optimizations: a simplified graph often is more amenable to structuring transformations. Also, the simplifying optimizations often are needed to enable a graph's transformation to a DFG. Thus in many cases it is not possible to test RCS performance without doing at least some of the simplifying optimizations.

First the hardware test platform and the DFG-to-RCS compilers are described. Then the effects of graph simplifying optimizations are presented, followed by the effects of the restructuring transformations.

7.1 Test platform and its limitations

There are two major system components that are external to those developed in the compiler described in this dissertation. First, there is hardware consisting of a conventional Pentium-based PC supplemented with a peripheral card containing multiple FPGAs. Second, there is a software com-

piler that takes the DFGs generated by the SA-C compiler and translates them into VHDL codes, which are then compiled and mapped onto the FPGAs using commercial software packages.

7.1.1 Hardware description

All tests described in this chapter have been run on a dual processor Pentium II machine, clocked at 266 MHz. The operating system is the Red Hat 6.0 version of Linux [38]. The PCI bus is 32 bits wide and runs at 33 MHz.

The RCS hardware is a Wildforce Board [2] from Annapolis Micro Systems. A simplified block diagram of the board is shown in figure 7.1. Its five user-programmable FPGAs are Xilinx XC4036 chips, each of which has a 36x36 matrix of Configurable Logic Blocks (CLBs). Four of the FPGAs, called "PE1" through "PE4" are nearly identical in their board interconnections. The fifth, called "CPE0", is seen as a control device, though it can be used simply as a coprocessor alongside the other four. CPE0 has a local memory of 256K 32-bit words; each of the other four FPGAs has a local memory of 128K 32-bit words. These memories are also directly accessible by the host via the PCI bus, though each is switched between host and FPGA access so that in general it is not practical for an FPGA to compute while the host is accessing its memory. Not shown in the diagram are three 512-word FIFO buffers connected to CPE0, PE1 and PE4. They allow concurrent access by the host and FPGA, but experiments using them showed miserable performance and they have since been ignored [39].

The board has a user-programmable clock that can be set in a range from 2 to 50 MHz. Each FPGA can access its local memory once per clock cycle. The board frequency is typically determined by the results of the place-and-route software that creates the FPGA configuration files, and must be the minimum frequency of the configurations.

Each CLB contains a small cluster of resources that are used and referred to by the VHDL-to-netlist translation software. A CLB consists of two 4-input lookup tables (FMAPs), one 3-input lookup table (HMAPs), and two flip-flops. Experience has shown that FMAP usage is usually the limiting factor when DFGs are mapped to the Wildforce FPGAs, though some register-intensive situations (e.g. very large loop windows) may find the number of flip-flops to be the relevant

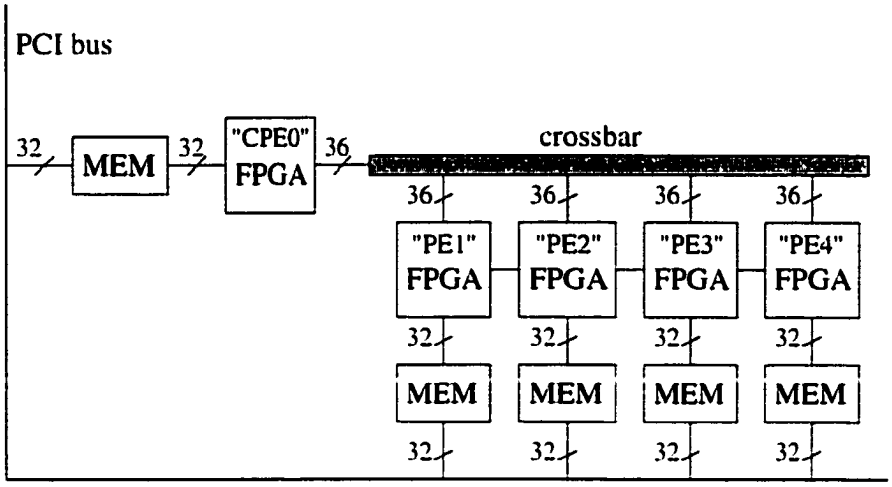


Figure 7.1: Block diagram of Wildforce board.

constraint. HMAP usage has been found in all situations to be negligible. In the tables presented later, the FMAP numbers are those that come from the VHDL-to-netlist compilation, and they provide an easy way of making approximate space comparisons.

The execution times for some parts of this system are consistent from run to run: both the configuration download time as well as the DMA transfers are predictable (assuming an otherwise quiet machine.) The FPGAs on the Wildforce board must be configured by streaming the data in a bit-serial way, leading to disappointingly large configuration times. It is not possible to partially-configure any of these FPGAs, so the time to configure the board is a consistent 106 msec. Since the configuration initially comes from a separate file, there is also an unpredictable file access time the first time a file is referenced. The SA-C run-time system caches configurations in host memory so subsequent downloads do not re-read the file.

DMA transfer rates have also been found to be very consistent, though they occasionally rise due to other bus activity on the host. A simple approximation of a DMA transfer assumes a fixed start-up time followed by a continuous transfer at a fixed rate. Measurements on the host machine used in these performance experiments have determined download start-up time and transfer rate parameters of approximately 50 μsec and 120 Mbytes per second, and corresponding upload parameters of 40 μsec and 66 Mbytes per second. In most of the performance data that follow, the configuration

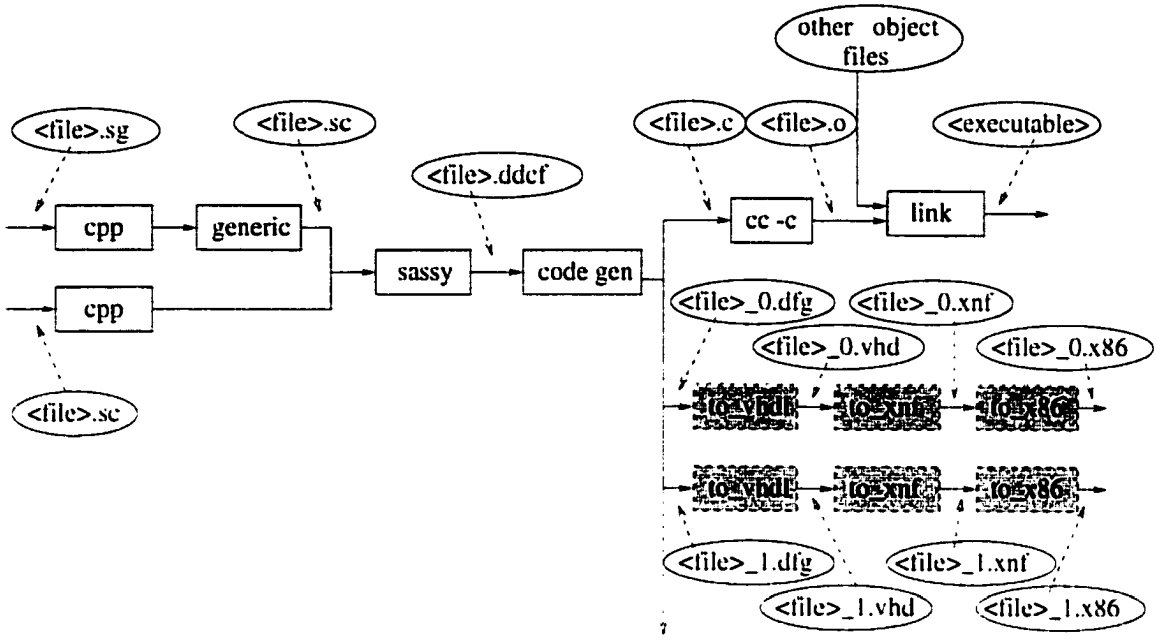


Figure 7.2: Compile paths of the SA-C compiler.

download times and the DMA transfers are not shown since they have been found to be consistent and to conform closely to the values implied by these parameters.

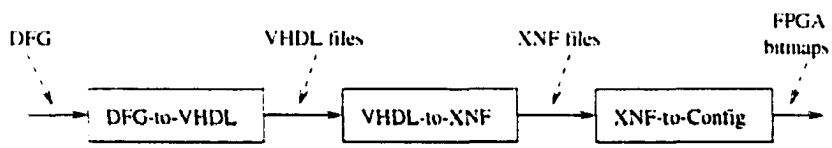
7.1.2 The SA-C compiler

The upper-level portion of the Cameron Project's software system is the SA-C compiler that is the focus of this dissertation [19]. A SA-C program compiles to a host machine executable that has calls to the Wildforce board. The system can also compile the entire program to a host executable for efficient program debugging. The host code includes interface code that automatically downloads FPGA configurations and source data, and uploads the results for further computation on the host. Figure 7.2 shows the compile paths of the system, including the dashed boxes that represent the low-level translation of DFGs to netlist (*.x86) files. A SA-C program may produce zero or more DFGs. Coordination of the various compiler components is done via a Perl script called *scc*. It handles a variety of options, as well as allowing the compile process to start and stop at various points in the diagram.

7.1.3 Abstract architecture and DFG-to-RCS compilation

When a SA-C program is compiled for execution on host and RCS, bottom-level **for** loops that meet certain criteria are compiled to DFGs for translation to FPGA configurations (see section 5.4). The current system follows a simple protocol for executing a loop on the RCS: First the FPGA configuration is downloaded into the chip. (The run-time system keeps track of the currently loaded configuration, so this download does not occur if it is unnecessary.) Next, any required source arrays are written to RCS memory, using DMA transfer routines supplied by Annapolis. Then various loop parameters are written to designated locations in RCS memory, using individual non-DMA word transfers. After these transfers, the host releases the RCS to compute, and waits for an interrupt indicating completion of the loop. Once interrupted, the host uploads the results using DMA transfers. The host then continues with SA-C program execution. While there are numerous opportunities for optimizing this process, no such optimization has yet been undertaken.

The SA-C compiler is able to produce DFGs for many bottom-level loops in a DDCF graph. Each of these DFGs is individually compiled in the following process:



A piece of Cameron-developed software transforms a DFG into multiple VHDL files [8]. These are then compiled by *Synplicity* [41], a commercial piece of software that produces a "netlist" file containing a low-level logic description of the FPGA circuit. There is one netlist for each FPGA target. Part of the feedback from *Synplicity* is a count of FMAPs, the small lookup tables that are paired in each Configuration Logic Block (CLB). An FMAP count is a useful way to quantify the FPGA logic space requirement of a netlist. Finally, each netlist goes through a "place-and-route" process, using software provided by Xilinx (the manufacturer of the FPGAs), producing the actual bitmap files that are loaded onto the FPGAs when a program is run.

The place-and-route step also returns a maximum clock frequency for the configuration, which is a function of the critical path through the FPGA logic. Since the place-and-route process is heuristic,

clock frequencies sometimes vary in unexpected ways. Experience has shown that as the FPGA internal logic resources approach 70% FMAP utilization, the clock frequency drops significantly, probably due to the place-and-route software's inability to find a good routing. The time spent in place-and-route also seems to grow significantly as logic utilization grows, increasing from a typical twenty minutes per FPGA to many hours. Of course, it is possible for the place-and-route process to fail: there may be no possible routing, or the software may not be able to find one. Some DFG-to-RCS translations map to one FPGA, whereas others target multiple FPGAs. Since the Wildforce board has one clock, a system that targets multiple FPGAs must clock the board at the minimum of the individual frequencies returned by the place-and-route calls.

The Cameron project has two separately developed DFG-to-RCS compilers, designated in this document as "One-PE" and "Two-PE" systems. In the One-PE system the entire DFG is compiled onto the Wildforce board's CPE0 device, and all local memory references are to its memory. In the Two-PE system the CPE0 device is used to implement the DFG's generator nodes, which read array values from its local memory, and PE1 is used to implement the DFG loop body and loop-return nodes that write the results to PE1's local memory. CPE0 sends the generator outputs to PE1 via the crossbar. One job of the host/RCS interface code (see section 5.3) is the allocation of source and target memory in the RCS local memories, so the SA-C compiler must be told whether it is compiling for the One-PE or the Two-PE system. For the One-PE system it allocates space for both the source and target arrays in CPE0's memory, whereas in the Two-PE system it allocates source arrays in CPE0's memory and target arrays in PE1's memory. While compilation targeting all five Wildforce FPGAs should not be particularly difficult to implement, the Cameron project has not yet progressed to that point.

The DFG-to-RCS translation currently does no pipelining and does not handle look-up tables, so the entire body of a loop is translated into a purely combinational, asynchronous circuit: this means that a loop body always executes in one RCS machine cycle. Local memory accesses on the Wildforce board occur using a clock derived from the same clock that executes loop bodies, so a loop

that gives rise to a slow clock (derived from the place-and-route step) forces all memory accesses to be slowed down as well. Reads and writes to each memory can be issued one-per-cycle.

The one-cycle-per-memory-access behavior of the system makes it easy to establish lower bounds on the number of cycles a given loop execution must take on the RCS. For example, since any loop must read the data it needs and write the data it produces from/to local memory, a simple lower bound can be derived. For the One-PE system it is the sum of the words read and the words written (because one memory is used for both source and target data). For the Two-PE system, the lower bound is the max of the read and write counts. This lower bound is independent of any loop transformations that may be applied, and will be referred to as the "I/O LB".

A second kind of lower bound can be inferred based on the behavior of a loop's window generator(s). Overlapping windows read some rows of source arrays more than once, and this behavior is a function of the window size and step values, making the memory read pattern completely predictable. Thus, for a given set of window generators another lower bound can be derived since the exact number of words read and written can be predicted. Also, the one-cycle-per-iteration behavior of the system means that the number of loop iterations gives rise to a lower bound as well. Combining these, the lower bound for a given execution is the max of the bounds derived from the memory accesses and from the iterations. This will be referred to as the "Generator-based LB".

7.2 Dataflow simulation

The DFG form that is produced when a SA-C loop is compiled for RCS execution has been designed to allow not only translation to VHDL but also token-driven simulation. A DFG simulator/viewer has been developed as part of the SA-C compilation system. One of the original motivations was to achieve the capability of deriving useful performance data without having to go through the long DFG-to-RCS compilation step. For example, it was hoped that statistics gathered during a DFG simulation might allow inference of performance behavior on the RCS.

However, the token-driven aspect of a DFG simulation does not relate well to the RCS execution. Token flow implies synchronization between DFG nodes, but when translated to FPGAs the

DFG loop body becomes an asynchronous circuit, with no synchronization between the low-level operations in the loop body. Thus token timing behavior, token queues and parallelism profiles that have been used in other dataflow projects to approximate the behavior of program execution are not relevant here. Nevertheless, the DFG simulator and viewer have been useful for debugging, DFG validation and demonstration of SA-C language semantics.

7.3 Graph simplifying optimizations

Many of the optimizations described in chapter 4 are designed to simplify a DFG, thus improving both space and time performance. These optimizations are:

1. invariant code motion (hoists code out of the DFG)
2. constant-expression switch elimination
3. array value propagation
4. constant folding
5. algebraic identity simplification and strength reduction
6. dead code elimination
7. common subexpression elimination
8. array reference chain elimination
9. bit width narrowing

Two SA-C codes will be used to demonstrate the effects of these optimizations: a square root computation and a Prewitt edge detector.

7.3.1 Square root

Section 4.1.7 discussed the Bit-width Narrowing optimization, using as an example a nextified loop that computes a square root. When the bit-width narrowing optimization is enabled, the compiler reports the before-and-after total bit widths of various categories of dataflow nodes, such as

arithmetic and conditional operators. This optimization has been applied to SA-C codes written by members of the Cameron Project, and nearly all showed little or no narrowing of operators, a result that is explained by the fact that SA-C programmers tend to do such narrowing themselves by selecting variable types that are just wide enough to hold their data values. The only SA-C codes that have shown significant narrowing of operator bit-widths are those containing square root operations; these opportunities arise due to the unrolling, by the compiler, of the square root loop.

The loop that computes the square root of a **uint32** is used here to measure the effects of bit-width narrowing. In order for this optimization to take place, the loop is unrolled and enclosed in an outer loop so that the compiler will create a DFG. The SA-C code is shown in the appendix, section A.1. The following table shows the effects of bit-width narrowing for this code:

	with narrowing	without narrowing
logic ops	542 bits	60 bits
compare ops	512 bits	272 bits
arithmetic ops	480 bits	240 bits
selector ops	736 bits	375 bits
FPGA space	1010 FMAPs	1002 FMAPs

In spite of significant bit-width reductions in this code, the improvement in FPGA space use is very slight. This surprisingly small difference indicates that the VHDL compiler and/or the place-and-route software is eliminating most of the unnecessary bits and connections, making bit-width narrowing by the SA-C compiler largely irrelevant.

7.3.2 Prewitt edge detector

A common image processing task is the detection of edges in an image, and the *Prewitt* algorithm [37] is one of a number of standard edge detection methods. A SA-C implementation is shown in the appendix, section A.2.

If functions are allowed to be inlined and this code is compiled for RCS execution without other optimizations, the inner loop

```
for h in H dot w in W dot v in V
    return (sum ((int11)w*h), sum ((int11)w*v));
```

is the only part of the code that compiles to a DFG (since only bottom-level loops can be transformed

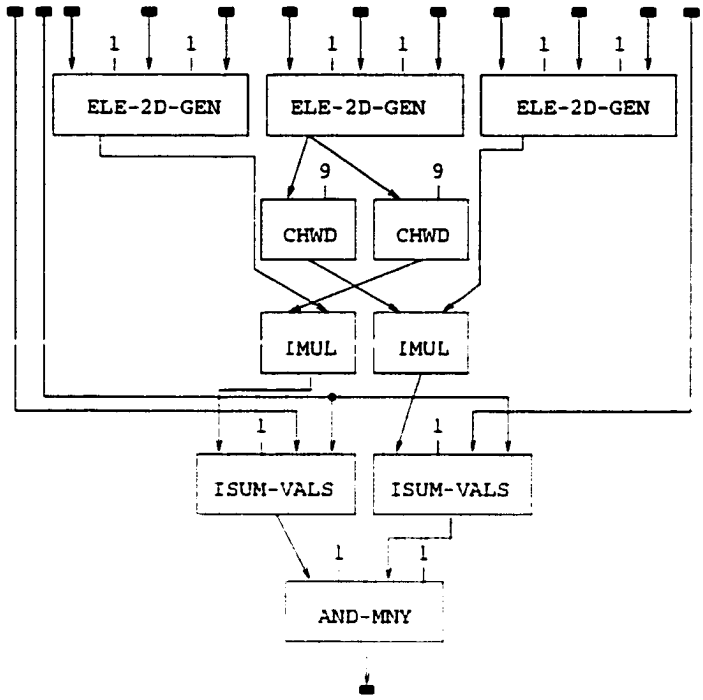


Figure 7.3: DFG produced by unoptimized Prewitt edge detector.

to DFGs.) Figure 7.3 shows this DFG. The CHANGE-WIDTH nodes (abbreviated CHWD in DFG figures) convert the output of the middle generator from an unsigned value (**uint8**) to a signed value (**int9**) by concatenating a zero on the left. The cast to an **int11** that appears in the source code has been removed in the DFG translation process, as described in section 5.4.2.2. Each IMUL node has inputs of nine and two bits, and an output of eleven bits. Each of the three generator nodes has three inputs with external sources, one for the address of its source array and two for the extents of that array. Each of the two loop-return ISUM-VALS nodes has two inputs from external sources. One is the iteration count; the other is the target address to which it will write its final result. Since this loop performs only nine iterations, its execution on the RCS produces no benefit.

If all optimizations are allowed to take place, the DFG of figure 7.4 is produced. The multiplication nodes are removed, and the middle value from the window generator is now unused. The width-changing nodes have been reduced by common subexpression elimination.

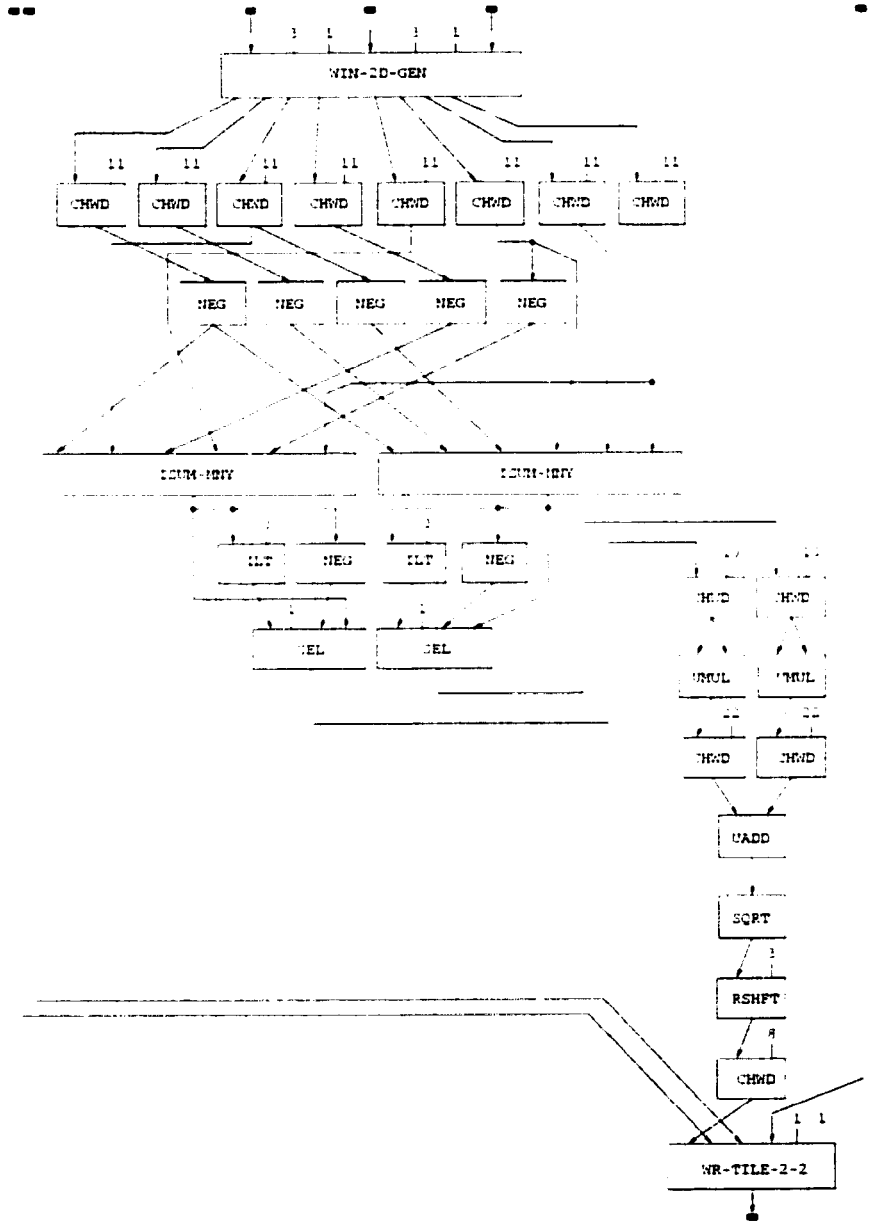


Figure 7.4: DFG produced by fully optimized Prewitt edge detector.

7.4 Loop restructuring optimizations

Loop restructuring optimizations consist of the following:

1. stripmining
2. full loop unrolling
3. loop fusion
4. loop nextification
5. window narrowing
6. array unification

In the discussion that follows, they are studied both individually and, when appropriate, in combinations. Because the Wildforce board's FPGAs are quite small, it is difficult to study the effects of some of these optimizations on real IP codes. For example, stripmining is designed to reduce FPGA-local memory traffic by running larger windows that feed multiple instances of the loop body. This increases the logic space requirements of both the generators and loop bodies, and since just one instance of an IP routine often barely fits or fails to fit, it is impossible to measure the effects of stripmining on interesting IP routines. Therefore, for most of these tests a tiny loop body is used, which allows the optimizations to be applied while still keeping the configurations within a feasible amount of space.

7.4.1 Stripmining

Section 4.1.3 describes Stripmining, which in combination with inner loop unrolling produces the effect of partial loop unrolling in multiple dimensions. The transformation has three goals:

1. increase parallelism by executing multiple loop bodies concurrently
2. reduce the number of loop iterations, thereby reducing loop overhead
3. reduce the amount of data read by a generator on the FPGA from the local memory.

This comes at the expense of FPGA logic space, due to multiple loop bodies and larger window generators. The first two goals should be achieved for loops of any rank, whereas the third goal applies only to loops with a rank greater than one, since those loops have vertical window overlap and therefore read some rows of data from local memory more than once.

As discussed earlier, lower bounds on the number of cycles required to execute a given loop on the RCS can be inferred for a given loop and input data. This leads to inherent limits on the amount of useful parallelism that can be exploited, since the local memory-FPGA communication quickly becomes a bottleneck. For example, consider a loop reading a **uint8** array with a 4xM window generator. Each loop iteration requires four new array values (ignoring beginning and end of row effects). This means that, on average, every iteration will require a memory access, and stripmining for parallelism (i.e. reducing the number of iterations by replicating loop bodies) should not be useful: the 4xM window generator is already large enough to be memory-bound. But in spite of the memory limits, stripmining can still be useful by reducing loop overhead and reducing the number of rows that are read multiple times.

Horizontal stripmining does not change the number of words read by the FPGA from its local memory, because the low-level implementation of a window generator uses shift registers to implement the sliding behavior of the window. Making a window wider adds registers but does not change the number of words read from memory. Thus, if horizontal stripmining is to be effective at all, it must happen only by the reduction of loop overhead due to fewer iterations. Vertical stripmining, on the other hand, will reduce the number of times that rows are read from local memory. For example, a 3x3 window generator such as that used in the Prewitt edge detector will read each row of the source array three times (neglecting the special cases of the boundaries.) Stripmining to a 3x4 will not change this, but a 4x3 stripmine (which will have a step of two) will read each row twice rather than three times, a 33% reduction in traffic from the local memory. Carrying this further, a 5x3 stripmine (with a step size of three) will produce a regular pattern of two rows read twice and a third row read once, a 44% reduction in traffic.

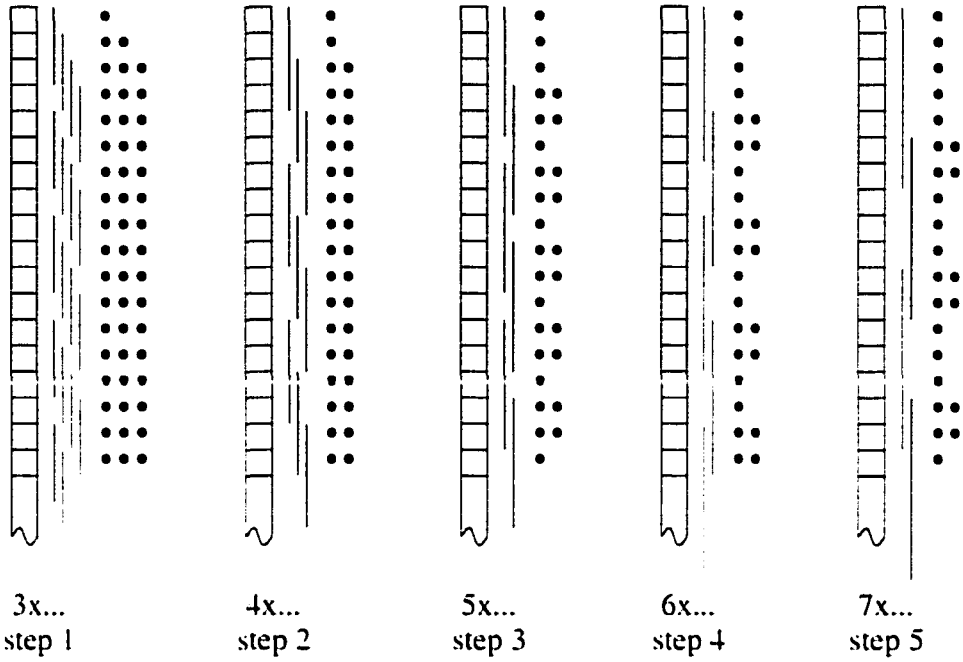


Figure 7.5: The number of times rows are read varies with different stripmining parameters. The dots show the number of times a row is read.

There is an invariant in vertical stripmining: the overlap of the windows of two successive horizontal traversals is constant, regardless of the degree of stripmining, because the window and step sizes both increment. Repeating the variable definitions as described in section 4.1.3.

- v is the original window's extent
- s is the original window's step
- a is the stripmine window's extent
- t is the stripmine window's step

the overlap is always $v - s$. In the 3x3 case and its vertically stripmined variants, this overlap is two. Figure 7.5 shows the effects in this example as the vertical stripmining parameter grows. In general, as the stripmine window extent grows to the region where $t > a/2$, there are clusters of rows that are read twice, interspersed with clusters that are read once. The number of rows in the read-twice clusters are the overlap size, $v - s$.

Two simple codes, called "Strip-1x3" and "Strip-2x3" are used to test the performance effects of stripmining:

```
// Strip-1x3:
uint<m> R[:,:] =
    for window W[1,3] in A
        return (array (array_max (W)));
```

```
// Strip-2x3:
uint<m> R[:,:] =
    for window W[2,3] in A
        return (array (array_max (W)));
```

Strip-1x3 uses a SA-C loop that runs a 1x3 window over a matrix and returns an array of the max of each window. Because the vertical window dimension is one, there is no vertical window overlap and vertical stripmining will not affect the number of words read from memory. In Strip-2x3 the windows overlap, allowing vertical stripmining to reduce the number of memory accesses. Because loop stripmining does not affect configuration download or the memory traffic between host and RCS, the reported measurements are only of the loop executions on the RCS.

7.4.1.1 Vertically stripmining a 1x3 window

Since a 1x3 window does not read any memory location more than once, any gains in performance will be due to increased parallelism and reduced iteration overhead. Three vertical stripmining tests have been performed with the Strip-1x3 code, using arrays of 2-bit, 8-bit and 32-bit data.

Table 7.1 shows the results of the 2-bit stripmining tests run on the One-PE system. The first three columns are FMAPs (FPGA internal logic space), clock frequency resulting from place-and-route, and execution time (not including configuration download time or the data transfers between host and RCS.) The remaining columns are inferred values: the number of cycles is computed by multiplying the clock frequency by the execution time, and the number of iterations and memory accesses are calculated knowing the stripmine dimensions and the array size. Note that the 2-bit data elements are packed in the RCS's 32-bit words, so one 300-element row of the source array takes 19 words of memory, with the entire image consuming 3,762 words. Because rows are word-aligned, the 198x298 result array also takes 3,762 words. The last column of the table represents both reading and writing accesses. Also note that, since stripmining can leave a "fringe" that is not touched by the larger window, the number of memory accesses varies somewhat for different stripmine depths.

stripmine dimensions	FMAPs	clock freq (MHz)	exec time (msec)	cycles (X 1000)	iterations (X 1000)	mem word accesses (X 1000)
1x3	726	9.18	8.12	74.54	59.004	7.524
2x3	802	8.93	4.42	39.47	29.502	7.524
3x3	877	8.05	3.41	27.45	19.668	7.524
4x3	903	8.65	2.47	21.37	14.602	7.448
5x3	986	8.18	2.17	17.75	11.622	7.410
6x3	1113	8.41	1.87	15.73	9.834	7.524
7x3	1093	7.44	1.86	13.84	8.344	7.448
8x3	1096	9.41	1.32	12.42	7.152	7.296
9x3	1194	5.47	2.14	11.71	6.556	7.524
10x3	1240	8.06	1.31	10.56	5.662	7.220
11x3	1318	7.08	1.46	10.34	5.364	7.524
12x3	1373	4.93	1.92	9.47	4.768	7.296
13x3	1453	7.42	1.24	9.20	4.470	7.410
14x3	1509	4.35	2.03	8.83	4.172	7.448

Table 7.1: Vertical stripmining performance on 198x300 2-bit elements using Strip-1x3 code. The lower bound on number of cycles is the max of the iterations and word accesses: the dominant value is underlined in each row.

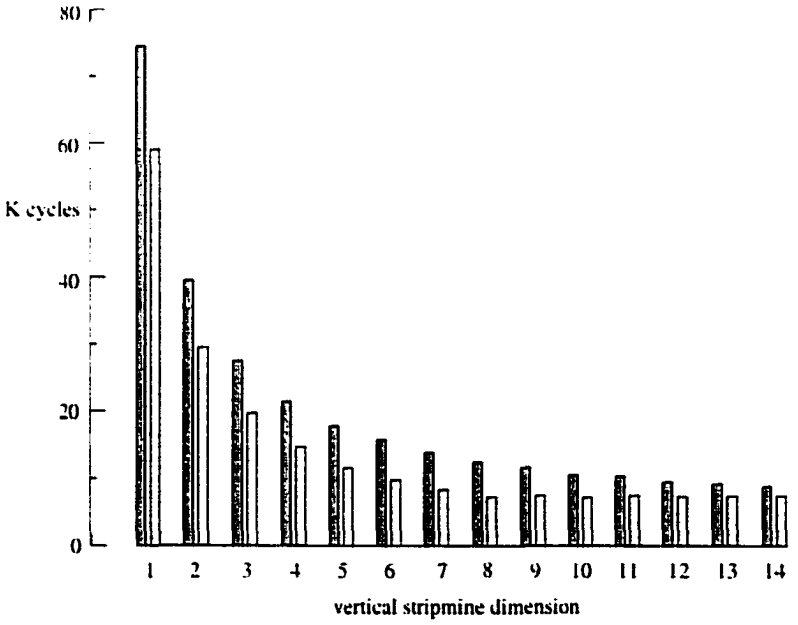


Figure 7.6: Comparing cycles used (left bar) and minimum cycles required (right bar) for the data of table 7.1.

The FMAPs show a growing consumption of FPGA logic space as vertical stripmining deepens. The 14x3 code uses 1509 CLBs, about 59% of the chip. The clock frequency goes generally downward as the FMAPs increase, although the heuristic place-and-route process causes significant random variation in frequency. The execution time drops significantly as stripmining deepens, though it is not monotonic due to the variation in clock frequency.

Performance measured in number-of-cycles perhaps is more interesting than the actual execution time. The clock cycles drop monotonically as stripmining deepens. The number of iterations goes down with deeper stripmining because the window step size grows. There is a Generator-based LB on the number of cycles required for each window size, found by taking the max of the iteration count and the number of memory accesses. For the shallower window sizes the iteration count determines this lower bound, but starting with an 8x3 window the memory accesses set the bound. At 14x3, the 8.83 Kcycles are approximately 18% over the lower bound of 7.45 Kcycles. In other words, there are relatively few cycles in which memory is not being accessed. Figure 7.6 shows a bar graph that compares the measured number of cycles with the Generator-based LB. It is useful to note that, since a 1x3 window does not read data more than once, the last column of the table (the number of memory accesses) is also the I/O LB for this computation, since all source elements must be read and all result elements must be written. Thus, at a 14x3 stripmining dimension, the number of cycles used is not far from the ideal, assuming that only one local memory is used.

The same experiment was performed on the Two-PE system, and yielded the results shown in table 7.2. In this system the loop generator is in one FPGA and the remainder of the loop is in another, so there are two FMAP entries for each code. Since this system reads and writes to two separate memories, the Generator-based LB is the max of the number of iterations, reads and writes. In all cases the iteration count dominates, so that stripmining is effective here by reducing the number of iterations. In all cases the number of cycles is only a few percent above the lower bound. (For the last two it is slightly below the lower bound, which must be attributed to various sources of measurement error in the system.)

stripmine dimensions	FMAPs	clock freq (MHz)	exec time (msec)	cycles (X 1000)	iterations (X 1000)	mem read accesses (X 1000)	mem write accesses (X 1000)
1x3	560 769	8.03	7.52	60.39	59.004	3.762	3.762
2x3	637 799	7.89	3.84	30.30	29.502	3.762	3.762
3x3	705 882	7.91	2.56	20.25	19.668	3.762	3.762
4x3	772 951	8.30	1.81	15.02	14.602	3.724	3.724
5x3	773 1012	8.01	1.50	12.02	11.622	3.705	3.705
6x3	873 1033	7.40	1.38	10.21	9.834	3.762	3.762
7x3	933 1099	7.78	1.11	8.64	8.344	3.724	3.724
8x3	980 1211	7.86	0.95	7.47	7.152	3.648	3.648
9x3	988 1260	7.95	0.93	7.39	6.556	3.762	3.762
10x3	1070 1322	7.67	0.77	5.91	5.662	3.610	3.610
11x3	1142 1469	8.00	0.74	5.92	5.364	3.762	3.762
12x3	1232 1547	5.86	0.83	4.86	4.768	3.648	3.648
13x3	1278 1557	4.59	0.92	4.22	4.470	3.705	3.705
14x3	1358 1525	4.45	0.93	4.14	4.172	3.724	3.724

Table 7.2: Vertical stripmining performance on 198x300 2-bit elements using Strip-1x3 code and the Two-PE system. The lower bound on number of cycles is the max of the iterations, read accesses, and write accesses. The dominant value is underlined in each row.

stripmine dimensions	FMAPs	clock freq (MHz)	exec time (msec)	cycles (X 1000)	iterations (X 1000)	mem word accesses (X 1000)
1x3	680	9.17	12.95	118.75	59.004	29.700
2x3	744	9.31	7.19	66.94	29.502	29.700
3x3	860	8.41	5.90	49.62	19.668	29.700
4x3	864	8.54	4.75	40.57	14.602	29.400
5x3	978	9.01	3.91	35.23	11.622	29.250
6x3	1023	7.02	4.60	32.29	9.834	29.700
7x3	1101	7.61	4.16	31.66	8.344	29.400
8x3	1131	8.57	3.59	30.77	7.152	28.800
9x3	1233	7.58	4.16	31.53	6.556	29.700
10x3	1276	7.30	4.12	30.08	5.662	28.500
11x3	1361	4.90	6.36	31.16	5.364	29.700
12x3	1445	5.33	5.65	30.11	4.768	28.800
13x3	1511	4.27	7.14	30.49	4.470	29.250
14x3	1579	2.96	10.31	30.52	4.172	29.400

Table 7.3: Vertical stripmining performance on 198x300 8-bit elements using Strip-1x3 code. The lower bound on number of cycles is the max of the iterations and word accesses; the dominant value is underlined in each row.

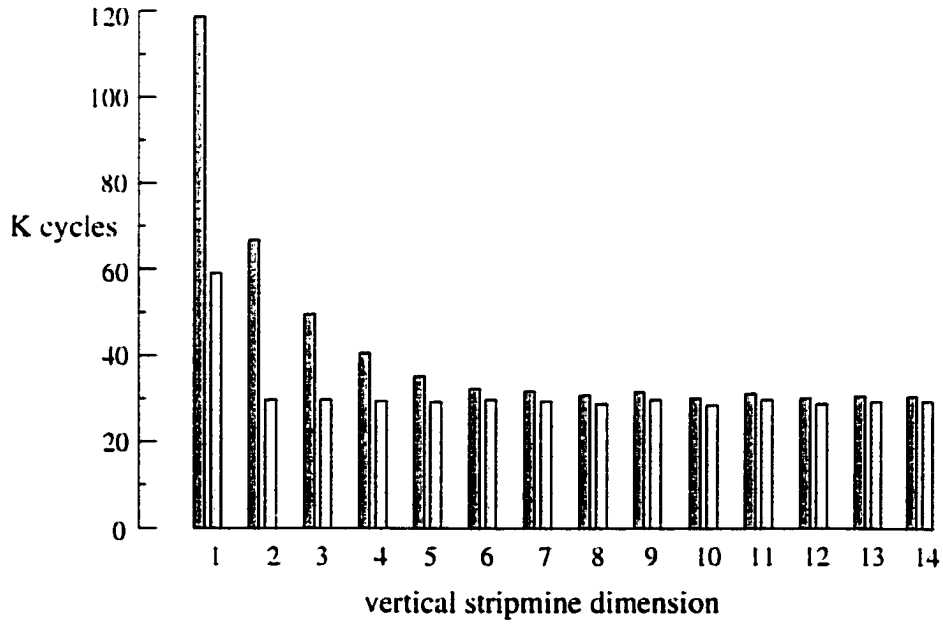


Figure 7.7: Comparing cycles used (left bar) and minimum cycles required (right bar) for data of table 7.3.

A similar experiment, performed on a 198x300 array of 8-bit elements and using the One-PE system, yields table 7.3. This time the iterations determine a lower bound for only the 1x3 case; all others are memory constrained. The iterations are identical to the 2-bit table, but the number of memory accesses is nearly four times larger because of the 8-bit array elements. (Because of word alignment, the memory access ratio compared to the 2-bit tests is slightly less than four.) At an 8x3 stripmine, the number of cycles has reached a plateau. For 14x3, the actual number of cycles is about 4% greater than the lower bound. Figure 7.7 shows the bar graphs for these data.

Finally, the same experiment performed on a 198x300 array of 32-bit elements yields table 7.4. The memory accesses establish the lower bound of required cycles in all cases. At 11x3 the cycles used are about $4\frac{1}{2}\%$ above the lower bound. Note that the 10x3 case could not be routed during the place-and-route phase. Also note the unusual case of 3x3, where the number of cycles used is only 0.4% above the required minimum. There is currently no known reason for this anomalous data point. Figure 7.8 shows the bar graphs for these data.

7.4.1.2 Vertically stripmining a 2x3 window

The 2x3 window in Strip-2x3 has vertical overlap, so vertical stripmining of such a window affects the amount of data read from the RCS local memory. This means that the Generator-based LB and the I/O LB will not be the same. As stripmining deepens, fewer rows are redundantly read and the Generator-based LB moves closer to the I/O LB.

Table 7.5 shows the data for this experiment. The number of memory accesses is computed using three values: the number of rows read per horizontal sweep, the number of rows written per sweep, and the number of sweeps performed (i.e. the vertical loop extent). The number of rows read is the vertical window size, and the number of rows written is one less than the vertical window size. The number of sweeps is derived directly from the computation described in section 4.1.3.1 that computes the start index of the fringe array. For example, consider the 7x3 case. The number of rows read in a sweep is seven, and the number of rows written is six. There are 75 words in a row, so there are 13 × 75, or 975 words accessed per sweep. The fringe index equation says that the fringe starts at vertical index 192, and since the step size is six, there are 192/6, or 32 sweeps.

stripmine dimensions	FMAPs	clock freq (MHz)	exec time (msec)	cycles (X 1000)	iterations (X 1000)	mem word accesses (X 1000)
1x3	632	10.04	23.67	237.65	59.004	118.404
2x3	734	9.66	15.39	148.67	29.502	118.404
3x3	860	8.56	13.89	118.90	19.668	118.404
4x3	887	8.84	14.93	131.98	14.602	117.208
5x3	1023	9.33	13.77	128.47	11.622	116.610
6x3	1085	8.36	15.38	128.58	9.834	118.404
7x3	1194	8.83	14.23	125.65	8.344	117.208
8x3	1274	7.45	16.40	122.18	7.152	114.816
9x3	1348	4.32	28.93	124.98	6.556	118.404
10x3						
11x3	1543	3.25	38.11	123.86	5.364	118.404

Table 7.4: Vertical stripmining performance on 198x300 32-bit elements using Strip-1x3 code. The lower bound on number of cycles is the max of the iterations and word accesses; the dominant value is underlined in each row.

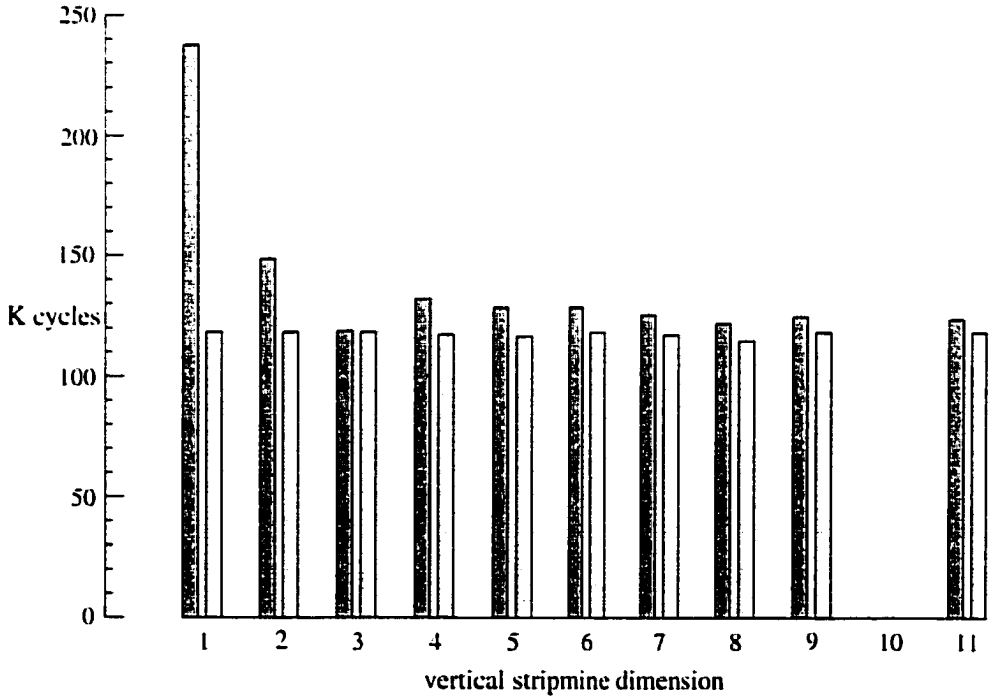


Figure 7.8: Comparing cycles used (left bar) and minimum cycles required (right bar) for data of table 7.4.

Thus there are 975 × 32. or 31,200 memory accesses performed. The Generator-based LB in the 2x3 case is determined by the iteration count, while the others are determined by the number of memory accesses. The last column of the table represents the I/O LB and varies slightly because the amounts of data read and written change due to varying fringe effects. As stripmining deepens, the Generator-based LB gets close to the I/O LB (29.325 versus 28,125 for the deepest case).

The number of cycles executed decreases monotonically in this experiment. At the deepest stripmine. 12x3. the number of cycles is about 5% larger than its Generator-based LB. and 9% larger than the I/O LB. Figure 7.9 shows a bar graph of the cycle times.

7.4.1.3 Conclusions about stripmining

Vertical stripmining is a highly effective optimization. reducing the number of local memory reads as well as reducing total loop overhead. Stripmining can bring the number of cycles impressively close to the lower bound for a given set of loop parameters. On the Wildforce board. stripmining's effectiveness has been limited by FPGA space constraints. With larger FPGAs. stripmining should be effective on actual IP loops.

7.4.2 Array blocking

Array blocking is closely related to stripmining and was described in section 4.1.3. To test the effectiveness of array blocking. the 2x3 window code of section 7.4.1.2 and its stripmined variants were used. but with an added `block (500,500)` pragma to move data to the RCS in 250,000-byte chunks. A 3000x4000-byte image was used for the test. so the image is broken into 48 blocks. The FPGA configurations were the same as those in section 4.1.3; only the host code changed.

As expected. the data transfer times did not vary for the different levels of stripmining. The download times were 2.0 msec per block. for a total of 100 msec and a transfer rate 120 MBytes per second. The upload times were 3.7 msec per block. for a total of 178 msec and a transfer rate of 67.5 MBytes per second. The 500x500 block is 4.21 times the size of the 198x300 image used in the stripmining experiments. The execution times for each block varied within a narrow range of 4.21 to 4.40 times the corresponding time in the stripmining test. These results demonstrate that array blocking on this system is an effective way to deal with the limited memory sizes on the RCS.

strip dimensions	FMAPs	clock freq (MHz)	exec time (msec)	cycles (X 1000)	vert loop extent	iterations (X 1000)	mem word accesses (X 1000)	I/O LB (X 1000)
2x3	724	9.18	14.48	132.93	197	58.706	44.325	29.625
3x3	827	8.20	8.97	73.55	98	29.204	36.750	29.475
4x3	948	9.20	5.84	53.73	65	19.370	34.125	29.325
5x3	1023	8.97	4.94	44.31	49	14.602	33.075	29.475
6x3	1121	7.50	5.08	38.10	39	11.622	32.175	29.325
7x3	1195	7.92	4.26	33.74	32	9.536	31.200	28.875
8x3	1300	7.94	4.25	33.75	28	8.344	31.500	29.475
9x3	1444	6.70	4.86	32.56	24	7.152	30.600	28.875
10x3	1461	6.19	5.11	31.63	21	6.258	29.925	28.425
11x3	1582	4.84	6.49	31.41	19	5.662	29.925	28.575
12x3	1664	4.36	7.05	30.74	17	5.066	29.325	28.125

Table 7.5: Vertical stripmining performance on 198x300 8-bit elements using Strip-2x3 code. The lower bound on number of cycles is the max of the iterations and word accesses: the dominant value is underlined in each row.

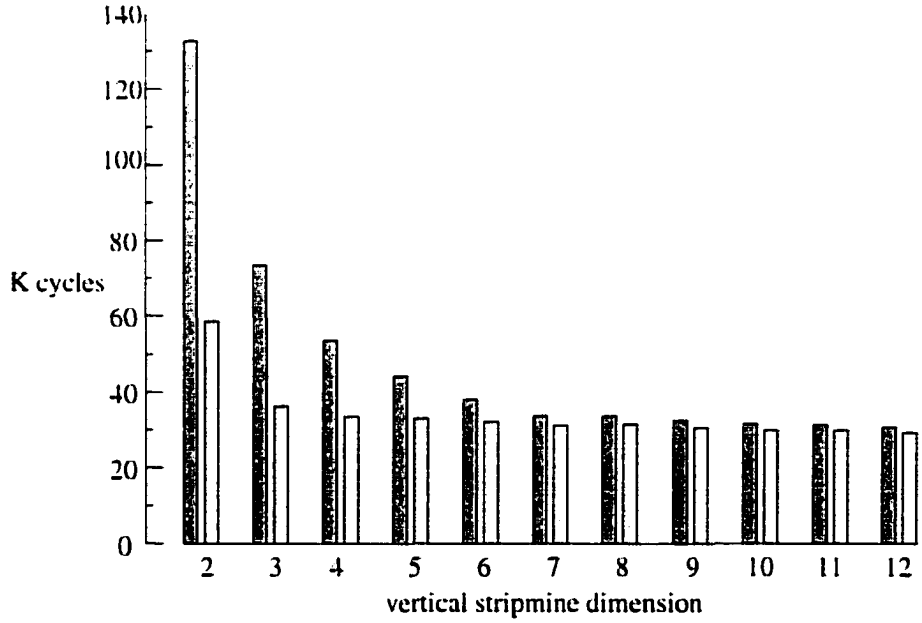


Figure 7.9: Comparing cycles used (left bar) and minimum cycles required (right bar) for data of table 7.5.

7.4.3 Loop fusion and nextification

The loop fusion approach described in section 4.1.4 fuses a producer/consumer loop pair. It creates a new loop with a window large enough to read from the source array all values that are needed to compute one value of the result array. After fusion, the loop body produces the required intermediate array values that are needed to compute one element of the result, and often these values are computed redundantly in multiple iterations. Thus, by itself this transformation reduces host-RCS memory traffic, but at the cost of a significantly larger loop body. The memory traffic between the FPGA and its local memory is reduced somewhat as well, and loop overhead may be reduced since fewer total loop iterations are run.

To measure the effects of loop fusion, the code shown in the appendix, section A.3, is used with an input array of 198x300. The first loop runs a 2x3 window over the image, producing a 197x298 result. To each window it applies a function that has enough work to make the example interesting but still fit the space limits of the Wildforce's FPGA after fusion. The second loop also uses a 2x3 window, producing a 196x296 result. Without loop fusion, the present system requires two host-to-RCS array downloads and two RCS-to-host array uploads, for a cost of 2.86 msec. It also requires a configuration download for the second loop, costing 106 msec. Fusion produces one loop with a 3x5 window; one array download and one array upload are required, costing 1.43 msec. There is no need for a new configuration download. Figure 7.10 shows a time-line that contrasts the unfused and fused timings. Here the configuration download time has been left out of the unfused time-line; if it had been included, the time-line would stretch across four pages.

The first two entries of table 7.6 show the performance data for the unfused and fused loops. Measured in number of execution cycles, the fused loop is nearly twice as fast as the unfused loop pair, 146.71 Kcycles versus 264.78 Kcycles.

Though the current system is not able to allow multiple loops to reside in the FPGA simultaneously, a reasonable system should be able to accommodate both loops, and the unfused time-line without the configuration download should be possible in the future. Also, a future compiler modification should allow the upload and download of the intermediate array to be eliminated, keeping

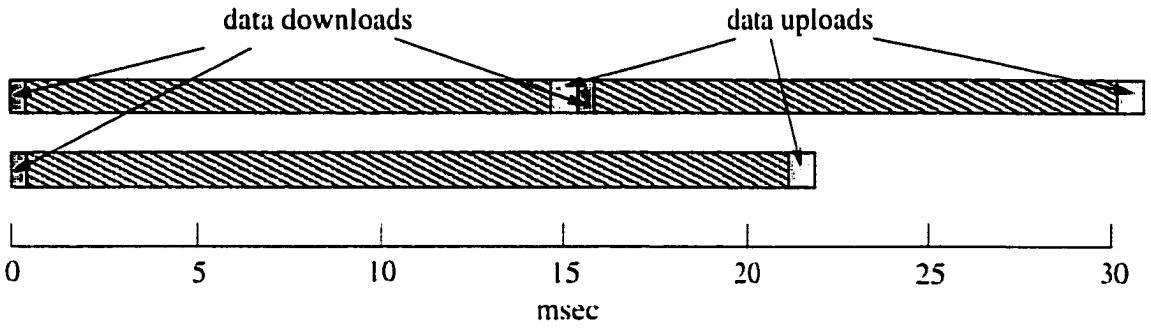


Figure 7.10: Time-lines of unfused and fused loops. The current system requires a reconfiguration between the loops of the unfused example. If the configuration download time had been shown, the first time-line would stretch across four pages.

code	FMAPs	clock freq (MHz)	time per sweep (msec)	exec time (msec)	cycles (x1000)	number of stripes	iters (x1000)	mem word accesses (x1000)
unfused	847	9.33	14.25	14.25	132.95		58.706	44.325
	724	9.18	14.36	14.36	131.83		58.016	43.904
fused	1778	7.06	20.78	20.78	146.71		58.016	58.604
fused/next	1146	8.61	0.102	20.00	172.20	196	58.016	58.604
strip(4x3)	1447	7.81	0.125	12.25	95.67	98	29.008	43.904
strip(5x3)	1734	8.10	0.130	8.45	68.45	65	19.240	38.805
fused/next(alt)	1154	8.27	17.77	17.77	146.98		58.408	58.800
strip(4x3)	1463	8.37	9.66	9.66	80.88		29.204	44.100
strip(5x3)	1770	6.82	8.56	8.56	58.34		19.370	39.000

Table 7.6: Performance of loop fusion with stripmining and nextification on a 198x300 8-bit array. The bottom three entries use the alternate approach to nextification.

the data on the RCS. Even after such system improvements, fusion will yield significant benefits, reducing compute time by more than 25% in this example.

The main disadvantage of SA-C's loop fusion transformation is its FPGA logic space consumption. In this example, six identical subgraphs of function f's code body are in the fused loop, contributing to its 1778 FMAPs. (Even so, note that the unfused loops together used 1571 FMAPs. Fusion trades an increase in code body for a reduction in window generators. Unfused, there are two generators. After fusion, there is one.) Loop nextification (and window narrowing) can reduce the number of duplicated code bodies, in this case eliminating four of the six since they are computing values that were already computed in previous iterations.

The general form of nextification, described in section 4.2.1, comes at the cost of breaking the loop into a loop pair. The outer loop runs on the host and produces horizontal stripes of the source array. Each is sent to the RCS, which uses it to compute a row of the result. If the fused loop is nextified, its window can be narrowed from 3x5 to 3x3, since the left two columns are not used. The main code is shown in the appendix, section A.4.

FPGA space is reduced from 1778 FMAPs to 1146 FMAPs, but nextification leads to the time-line at the top of figure 7.11. There are 196 horizontal stripes of data: each is sent to the FPGA, and the result row is read back. Clearly the compute/communicate ratio is very poor here. The third entry in figure 7.6 shows the performance data. Each data download and upload takes 0.053 msec, and each execution takes 0.102 msec. Thus this nextified loop takes 40.77 msec to run, double the time of the fused loop before nextification. The number of FPGA cycles grows after nextification due to the increased overhead of starting and stopping each of the stripe computations.

However, the space saving from nextification can be used to stripmine the loop. There are fewer and deeper stripes of data after stripmining, and the communicate time reduces because the number of transfers is reduced. The fourth and fifth entries in table 7.6 show the performance data for 4x3 and 5x3, and the second time-line in figure 7.11 shows the 5x3 case. The total time to execute the 4x3 case is 22.83 msec, while the 5x3 case runs in 15.99 msec, an improvement over the simple fused case in spite of the inefficient movement of data.

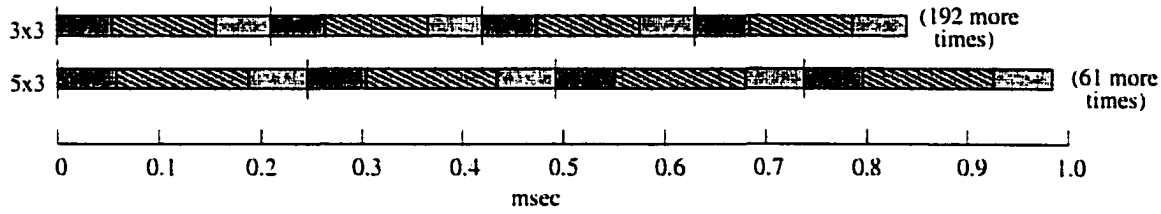


Figure 7.11: Time-lines of fused and nextified loops, before and after stripmining. The timeline is the concatenation of outer loop iterations, each with three parts: download, execute and upload. The first timeline completes after 40.77 msec, compared with 15.99 msec for the second one.

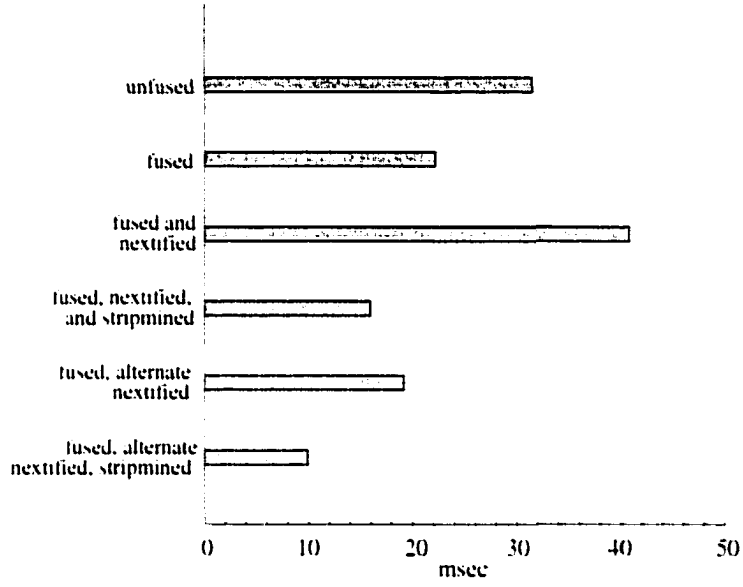


Figure 7.12: Performance of fusion and related optimizations.

Section 4.2.2 described an improved nextification approach that can be applied to this example. The resulting code is shown in the appendix, section A.5. The last three entries of table 7.6 show the performance of this loop, along with 4x3 and 5x3 stripmined versions. The number of cycles is reduced by roughly 10% compared to the previous nextification approach. Note that the number of iterations and memory accesses is slightly larger because of the added iterations that initialize the nextified values.

Figure 7.12 summarizes the loop fusion performance with the various other optimizations. Fusion, nextification and stripmining have reduced the total time from 31.5 msec to 10.0 msec. FPGA space limitations prevented deeper stripmining, which probably would have yielded further improvements.

	FMAPs	clock freq (MHz)	exec time (msec)	cycles (X 1000)	LB (X 1000)
before	862	7.56	17.59	132.98	58.10
	866	8.96	14.83	132.88	58.51(*)
after	762	9.13	16.16	147.54	59.10
	643	8.65	16.98	146.88	58.51(*)
fused	892	9.27	12.76	118.29	58.51(*)
strip 6x4	1921	5.18	6.80	35.22	32.18

Table 7.7: Array combining, with a 198x300 **uint8** input array. The (*) lower bounds come from the number of loop iterations; the others are from the number of memory accesses.

7.4.4 Array combining

Section 4.2.4 discussed the idea of combining arrays in situations where multiple arrays are passed between a producer/consumer loop pair. By combining the arrays into one, the upper loop has just one loop return and the lower loop has just one generator. To test the effects of this transformation, the codes shown in the appendix, section A.6, were used. The arrays are unified by bit-concatenating the values inside the loop and storing them in a single array. The lower loop reads values from the array and breaks them into two values for its loop body. Table 7.7 shows the performance results. After combining the arrays, the execution time rose by a very small amount, probably not significant after one considers the somewhat arbitrary nature of the clock frequencies. However, the number of cycles rises from 265.86 to 294.42 Kcycles after arrays have been combined. It is not clear why this should be.

While combining arrays did not by itself improve performance, it did enable loop fusion. (The present fusion implementation does not handle multiple edges between a loop pair.) After fusion, the performance is significantly improved. Both real execution time and number of cycles is reduced by more than half. Stripmining to a 6x4 window improves performance even more, reducing execution time from 32.42 msec originally to 6.80 msec, and the number of cycles from 265.86 to 35.22 Kcycles. This compares with a Generator-based LB for the 6x4 case of 32.18 Kcycles; stripmining has gotten to within 10% of this bound. This loop is not a good candidate for nextification, since there is little work in the upper loop body, and nextification trades upper loop bodies for registers.

7.4.5 Performance conclusions

Graph simplifying optimizations are important, both for the sake of reducing the size and complexity of a graph and for setting the stage for other optimizations. The Prewitt DFG, for example, was simplified significantly. Bit-width narrowing, however, was found to make little difference in FPGA space consumption for an unrolled square root, probably because the VHDL compiler is also able to remove connections that are not needed.

Array and loop size propagation is vitally important in setting the stage for full loop unrolling, which itself is important since it can turn a higher-level loop into a bottom-level loop. Loop restructuring optimizations were highly effective, especially when applied together. Producer/consumer loop fusion speeds up computations considerably, primarily by reducing the data traffic between the FPGA and its local memory. Stripmining adds further improvement by reducing still further the local data traffic as well as reducing the number of loop iterations. Finally, loop nextification reduces FPGA space consumption by eliminating redundant computations across loop iterations. This in turn may leave space for deeper stripmining.

Combining arrays between a producer/consumer loop pair did not improve performance, though it is still useful in that it sets the stage for fusing the loops. Array blocking, on the other hand, is effective in circumventing the size limits of the FPGA local memories. There is no significant penalty paid for breaking a large array into relatively few coarse-grain chunks and processing them one by one. However, in a larger context the need for array blocking could be detrimental since other parts of the computation might benefit from keeping an entire array in local memory rather than in host memory.

Chapter 8

Future work

The research described in this document can be continued in a variety of directions. What follows is a non-comprehensive list of some possible directions for further research.

8.1 Language enhancements

The SA-C language was designed very conservatively, in order to allow quick progress without complicated side issues. As the system matures it is worth considering ways to broaden the language. A variety of items may be considered:

- *Record* data structures were omitted in SA-C, simply to streamline the initial system development. (A SA-C programmer can circumvent this somewhat by concatenating values together in bit vectors, as long as they don't exceed the 32-bit limit.) But records are important for software engineering reasons, and in some cases they can be useful since they store data in an order that may improve locality. They should be added to the language.

- Recursion was not allowed in SA-C, in order to simplify program analysis and because it makes little sense in an FPGA-based environment. (There is no function-calling mechanism on FPGAs.) However, recursion can be useful in code intended for execution on the host machine, and probably should be available to a SA-C programmer.

- In the image processing and computer vision worlds (the application area of primary interest to the Cameron Project), it is often the case that a program's inputs and outputs are streams of data, e.g. frames received in real-time from a video camera. A ***stream*** data structure in

SA-C would accommodate programming for these and introduce a non-strict data structure into the language. (This would be similar to Sisal, where the stream is the lone non-strict data structure in the language.) While non-strictness tends to complicate program analysis for the compiler, it can be highly useful to programmers.

- The SA-C language, because of its intended target architecture, has emphasized regular data structures and access patterns, hence the language has very poor support for expressing irregular program behavior. For example, while SA-C has borrowed Fortran90's ability to read sub-arrays, it does not have a corresponding array-targeting capability. For example, a programmer cannot easily express the idea of "A[lo:hi] = <expr>". Of course, this is a problem in general for pure functional languages, since updatable memory does not exist in the semantics of these languages. However, many functional languages have array "updating" functions that take existing arrays and some kind of target specification, and return a new array that looks like an updated version of the source array. Under some circumstances compiler analysis can do update-in-place in the generated code, yielding execution efficiency similar to that of imperative programs [7]. Some effort should be devoted to looking at ways to allow SA-C arrays to be described in terms of existing arrays, and with syntax that is as consistent as possible with that of the existing array-reading operators.

8.2 More complete implementation of SA-C capability

The development of the SA-C compiler was driven by the requirements of the early SA-C programs, particularly the VSIPL routines. Some low-priority items were bypassed when producing DFGs, including the handling of **complex int** and **complex fix** values, and the handling of the **cross** operator in loop generators. Complex arithmetic probably should be handled during the DDCF-to-DFG translation, breaking the complex values into explicit individual edges and using low level integer operations that already exist in the DFGs. The **cross** operator of a loop generator creates fundamental problems in translation to DFGs: There is no way, with current DFG nodes and semantics, to express the generator interaction that takes place in a **cross** operation. Either the

DFG form must be altered to accommodate the operator, or the SA-C compiler could break a **cross** operation into an explicit loop nest and then translate the inner loop to a DFG.

8.3 SA-C in a mixed-language environment

Since a SA-C source file is compiled to an object file, it is possible for that file to be linked with object files that may have originated from other languages, e.g. C, Fortran, assembler, etc, making it possible to use a mix of languages at the function call level to express an application. It is possible both to call SA-C routines from other languages as well as to call a foreign language routine from a SA-C program. The former mode is safer and probably more useful. While calling a foreign language routine from a SA-C program is possible, the routine would have to be written carefully so that it did not violate SA-C's single-assignment semantics.

To call SA-C from C, the primary challenge for the programmer is the exposure of the SA-C run-time system internals. For example, a user wishing to pass a C array to a SA-C routine must first put the array into the appropriate data structure, and a C program receiving an array returned from a SA-C routine must either convert it back to C (since SA-C's use of dope vectors may leave the elements in the wrong order) or use array-referencing macros that already exist in the run-time system. A serious effort toward supporting mixed languages would include the development of an interface layer that would shield the user from the internal details of the SA-C run-time system. This would also allow SA-C compiler developers to make changes to the run-time system structures without breaking users' existing codes.

8.4 Host code efficiency

The emphasis of this research has been on compiling SA-C code to RCS hardware, and the host C code generation in the compiler has been done with little regard to efficiency. Rather, simplicity and correctness were the guiding principles. There are many opportunities for improvement in the host code, including array update-in-place, more efficient reduction implementations, elimination of unnecessary value masking, and freeing of heap-allocated objects.

8.5 Handling results of unknown sizes

In the current SA-C system implementation, the host code (generated from the interface portion of the RC_COMPUTE graph in the DDCF form) allocates target space for the values to be written by the loop executing on the RCS. During execution, the host is suspended until a "done" signal is received. Thus the current implementation is limited to loops where the target array sizes are known when the loop is about to be executed. This approach does not allow for the execution of **while** loops (since the iteration count is not known at allocation time) or loop-return operators that can produce varying sized results. An important of example of the latter is the **concat** operator: the sub-arrays that are concatted are allowed to vary in their rightmost dimensions, and the compiler cannot in general predict the sizes of the sub-arrays.

To solve this problem in a general way, the host could allocate on the RCS an accumulation buffer of some specified arbitrary size. The RCS can write to the buffer and signal the host when it is full. The host can upload the partial result, and signal the RCS that it can continue writing to the now-empty buffer. The work involved to include this capability would require small changes to the DDCF graphs and DFGs, and changes to the code generator and the protocols used to communicate between the host and the RCS.

8.6 More loop fusion

The beginning of section 4.1.4 showed three kinds of loop fusion, the first two of which have not yet been implemented. If these fusion transformations were included, they could interact favorably with the producer/consumer fusion that already exists in the compiler, making it possible to fuse a variety of loops in various configurations.

8.7 Multiple independent loops

The current system allows only one loop to exist on the RCS at any one time, and the run-time system downloads a new FPGA configuration each time a loop is to be run, unless it is already loaded. In the case of repeated execution of the same loop this approach works fine: The loop is

loaded once for the first execution and no reloads are necessary. But in the case of executions that alternate between two loops, the slow configuration speeds can kill performance. In cases such as this, it would be highly beneficial if both loops could exist in the FPGA and the host could select which loop to activate at any given time. In the current hardware, this is usually not practical because the small FPGAs typically have insufficient space for multiple loops, but this will become increasingly important as FPGA logic capacities grow.

To add this capability to the system, the protocols between the host and RCS would have to be expanded to allow the host to specify which loop to run; this should not be difficult. A more interesting issue is the matter of determining in the general case which loops to pack together, assuming that a program has too many loops to fit simultaneously. It's likely that programmer control here is needed, since the compiler cannot know in general what the sequence of loop calls will be. User control in turn would benefit greatly by a quick FPGA space estimation function to aid in selecting feasible loop packings.

8.8 Loop pipelines

If multiple loops can fit on an FPGA, then a logical next step is the placement of a producer/consumer loop pair on the FPGA. While the SA-C compiler can fuse such loops under some circumstances, loops whose production and consumption orders differ will not fuse. In this case it would be useful to put both loops on the FPGA and have them connected by an appropriate data transfer mechanism. There are significant challenges here with regard to the compiler's ability to infer useful information about the patterns of data production and use, as well as bounds on the required transfer memory size and methods of synchronizing the two loops.

8.9 Nested loops

Since this project has demonstrated that a complete loop, including generators and return collectors, can be placed on an FPGA, it is possible in principle to place a loop nest on an FPGA. This would be useful in a situation where a large loop, i.e. with enough work to be worth running on RCS, contained a loop that is too small to be worth running independently on the RCS. It would also be

useful in a setting where the SA-C compilation system might be used to produce complete stand-alone FPGA codes, i.e. in a system that has no host machine. In that case, feasibility rather than efficiency is important.

8.10 Reducing host-RCS traffic

There are various opportunities for making host/RCS data communication more efficient. For example, consider the case of a producer/consumer loop pair that could not be fused. In the current system the host moves the results of the first loop from FPGA local memory to host memory, then sends it back to the RCS to be read by the second loop. This useless round trip could be eliminated. More complex situations can also arise. For example, one of the nextification transformations breaks the source array into overlapping stripes, each of which is sent individually to the RCS. It should be possible instead to send the complete source array down once and then, for each RCS call, send a pointer to the stripe to be read.

Chapter 9

Conclusions

9.1 Conclusion

Field Programmable Gate Arrays (FPGAs) have been available for approximately fifteen years and have experienced gains in gate count and speed similar to those of conventional microprocessors. Currently-available FPGAs can be reprogrammed in a matter of milliseconds, making them interesting candidates for reconfigurable computing, in which specialized circuits can be produced for specific programs to execute more efficiently than a sequence of instructions on a CPU. Algorithms that exhibit parallelism and are highly regular may benefit from the use of FPGAs.

This research has presented an alternative to the conventional ways of programming FPGAs. SA-C (derived from "Single-Assignment C"), is a pure functional algorithmic language intended for the expression of image processing (IP) applications. SA-C's functional nature makes the compiler's job easier, as compared with imperative languages: parallelism is easy to detect, and analysis and transformations are fairly straightforward. Perhaps the most important part of the language is its loop *window generators*. They were designed to make many IP algorithms easy to express, but this research has shown them to be useful in expressing various graph restructuring transformations. A Data Dependence and Control Flow (DDCF) hierarchical graph form has been presented, as an intermediate form in which the SA-C compiler performs all of its optimizations.

Many conventional optimizations have been implemented in the SA-C compiler, primarily to set the stage for other analysis and optimizations that are particularly relevant when compiling to FPGAs. Array and loop size propagation have been found to be tremendously important in the

compiler, for they infer loop extents that can often allow a loop to be fully unrolled. This in turn may allow a higher level loop to be put on the RCS. Lookup tables may be an important optimization when mapping codes to FPGAs, but the capabilities of the system did not allow their effects to be measured for this dissertation.

Three graph structuring optimizations have been shown to be highly effective, especially when combined, and all three are performed using SA-C's window generators. Stripmining, especially in the vertical dimension, is a method of reducing loop iterations as well as reducing the number of source array rows that are read redundantly. Loop fusion reduces host/RCS memory traffic as well as reducing loop iterations and local memory/FPGA traffic. Nextification reduces redundant computations that are often caused by fusion, reducing FPGA space use and thereby allowing deeper stripmining to be applied.

Appendix A

Test codes

A.1 Square root code

The following shows a square root function embedded in a loop so that it can be converted to a DFG. This loop is described in section 7.3.1.

```
uint16[:] main (uint32 A[:]) {
    uint16 R[:] =
        for a in A
            return (array (sq_root (a)));
    } return (R);

uint16 sq_root (uint32 vsqn) {
    bits32 vsq = vsqn;
    bits32 asq = 0;
    bits16 a = 0;
    bits32 tvsq = 0;

    uint16 v =
        for uint4 i in [16] {
            bits32 nasq = ((bits32)((uint32)asq+(uint16)a)<<2) | 0b1;
            bits16 sa = a<<1;
            next tvsq = (tvsq<<2) | ((bits2)(vsq>>30));
            next vsq = vsq<<2;
            next a, next asq =
                if (nasq <= tvsq) return (sa|0b1, nasq)
                else return ( sa, asq<<2);
            } return (final (a));
    } return (v);
```

A.2 Prewitt edge detector code

This is the edge detector code described in section 7.3.2:

```
uint8[:,:] prewitt (uint8 Image[:,:]) {
    int2 H[3,3] = {{-1,-1,-1},{ 0,0,0},{ 1,1,1}};
    int2 V[3,3] = {{-1, 0, 1},{-1,0,1},{-1,0,1}};
    uint8 res[:,:] =
        for window W[3,3] in Image {
            int11 sh, int11 sv =
                for h in H dot w in W dot v in V
                    return (sum ((int11)w*h), sum ((int11)w*v));
            uint10 shp, uint10 svp = absval (sh), absval (sv);
            } return (array ((uint8)(((bits11)magnitude (shp, svp))>>3)));
    } return (res);

uint10 absval (int11 v) return (v<0?(-v):v);

uint11 magnitude (uint10 a, uint10 b)
    return ((uint11)sqrt ((uint22)a*a + (uint22)b*b));
```

A 3x3 window traverses the Image array. Each 3x3 window is multiplied point-wise by a horizontal mask (H) and a vertical mask (V), and the values are accumulated into two sums. The magnitude of those two sums is the return value for that window. Various type casts are used to cause arithmetic to be done with sufficient bit widths. Since the magnitude values are eleven bits wide, a right-shift of three bits normalizes the result back to eight bits so that it is viewable by tools such as *xv*. (The right shift could be replaced by a divide-by-eight; the compiler strength reduction pass will convert such a divide to a right-shift.)

A.3 Example of loop fusion opportunity

The following code is used to demonstrate loop fusion in section 7.4.3:

```
uint8[:,:] main (uint8 A[:,:]) {
    uint8 R0[:,:] =
        for window W[2,3] in A
            return (array (f (W)));
    uint8 R1[:,:] =
        for window W[2,3] in R0
            return (array (array_max (W)));
    } return (R1);

uint8 f (uint8 W[2,3]) {
    uint8 v0 = W[0,0] + W[1,1];
    uint8 v1 = W[0,1] + W[1,0];
    uint16 v2 = (uint16)v0 * v1;
    uint8 v3 = W[1,2] + W[0,2];
    uint16 v4 = (uint16)v3 * 5;
    uint16 v5 = v2 - v4;
    uint8 v7 = if (v5 > 3562) return (v5-1)
                    else return (v5+1);
    } return (v7);
```

A.4 Example of loop fusion followed by nextification

The following code shows the effects of nextification after the loops of section A.3 have been fused:

```
uint8[:,:] main (uint8 A[:,:]) {
    _, uint32 d= extents (A);
    uint8 R[:,:] =
        for window Stripe[3,d] in A {
            uint8 S00 = f (Stripe[0:1,0:2]);
            uint8 S01 = f (Stripe[0:1,1:3]);
            uint8 S10 = f (Stripe[1:2,0:2]);
            uint8 S11 = f (Stripe[1:2,1:3]);
            uint8 X[:,:] =
                for window W[3,3] in Stripe[:,2:] {
                    uint8 S02 = f (W[0:1,0:2]);
                    uint8 S12 = f (W[1:2,0:2]);

                    uint8 T[2,3] = {{S00, S01, S02},
                                    {S10, S11, S12}};

                    next S00 = S01;
                    next S01 = S02;
                    next S10 = S11;
                    next S11 = S12;

                    uint8 V = array_max (T);
                    uint8 RT[1,1] = {{V}};
                    } return (tile (RT));
            } return (tile (X));
        } return (R);
```

A.5 Example of loop fusion followed by alternate form of nextification

The following code shows the effects of the alternate form of nextification after the loops of section A.3 have been fused:

```
uint8[:,:] main (uint8 A[:,:]) {
    uint8 S00 = 0;
    uint8 S01 = 0;
    uint8 S10 = 0;
    uint8 S11 = 0;

    uint8 R[:,:] =
        for window W[3,3] in A {
            uint8 S02 = f (W[0:1,0:2]);
            uint8 S12 = f (W[1:2,0:2]);

            uint8 T[2,3] = {{S00, S01, S02},
                            {S10, S11, S12}};

            next S00 = S01;
            next S01 = S02;
            next S10 = S11;
            next S11 = S12;

            uint8 V = array_max (T);
            uint8 RT[1,1] = {{V}};
            } return (tile (RT));
    } return (R[:,2:]);
```

A.6 Example of array combining

The following codes show an array-combining example, used in section 7.4.4, before and after:

```
ufix8.2[:,:] main (uint8 A[:,:]) {
    uint8 R0[:,:], uint8 R1[:,:] =
        for window W[2,3] in A
            return (array (array_max (W)), array (array_min (W)));
    ufix8.2 Res[:,:] =
        for window W0[1,2] in R0 dot window W1[1,2] in R1
            return (array (((ufix8.2)W0[0,0]+W0[0,1]+W1[0,0]+W1[0,1])/4));
    } return (Res);
```

```
ufix8.2[:,:] main (uint8 A[:,:]) {
    bits16 R0[:,:] =
        for window W[2,3] in A {
            uint8 mx = array_max (W);
            uint8 mn = array_min (W);
            bits16 v = ((bits16)mx<<8) | (bits8)mn;
            } return (array (v));
    ufix8.2 Res[:,:] =
        for window W[1,2] in R0 {
            uint8 mx0 = W[0,0]>>8;
            uint8 mn0 = W[0,0];
            uint8 mx1 = W[0,1]>>8;
            uint8 mn1 = W[0,1];
            } return (array (((ufix8.2)mx0+mn0+mx1+mn1)/4));
    } return (Res);
```

REFERENCES

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers Principles, Techniques, and Tools.* Addison-Wesley Publishing Company, 1986.

[2] Annapolis Micro Systems, Inc., Annapolis, MD. *WILDFORCE Reference Manual.* 1997. `www.annapmicro.com`.

[3] A. P. W. Böhm and J. Hammes. On the memory performance of pure and impure, strict and non-strict functional programs. In *PARCO99.* Delft, August 1999.

[4] A. P. W. Böhm and J Sargeant. Code optimization for tagged-token dataflow machines. *IEEE Transactions on Computers*, pages pages 4–14, January 1989.

[5] S. Brown and J. Rose. Architecture of FPGAs and CPLDs: A tutorial. *IEEE Design and Test of Computers*, 12(2):42–57, 1996.

[6] T. J. Callahan and J. Wawrzynek. Instruction-level parallelism for reconfigurable computing. In *Field-Programmable Logic and Applications*, volume 8th International Workshop, September 1998.

[7] D. Cann. *Compilation Techniques for High Performance Applicative Computation.* PhD thesis, Colorado State University, 1989.

[8] M. Chawathe, M. Carter, C. Ross, R. Rinker, A. Patel, and W. Najjar. Dataflow graph to VHDL translation. Technical report, Colorado State University, Dept. of Computer Science, 2000.

[9] K. Compton and S. Hauck. Configurable computing: A survey of systems and software. Technical report, Northwestern University, Dept. of ECE, 1999.

[10] VSIPL Consortium. Vector Signal Image Processing Library Forum, October 1997. `www.vsipl.org`.

[11] A DeHon. Dynamically programmable gate arrays: A step toward increased computational density. In *Proceedings of the Fourth Canadian Workshop of Field-Programmable Devices*, May 1996.

[12] J. B. Dennis. The evolution of 'static' dataflow architecture. In J. L. Gaudiot and L. Bic, editors, *Advanced Topics in Data-Flow Computing.* Prentice-Hall, 1991.

[13] T. Ellis, I. Philips, and T. Lahey. *Fortran 90 Programming.* Addison-Wesley, 1994.

[14] A. Field and P. Harrison. *Functional Programming.* Addison-Wesley, 1988.

[15] S. C. Goldstein and M. Budiu. *The DIL Language and Compiler Manual.* Carnegie Mellon University, 1999. `www.ece.cmu.edu/research/piperench/dil.ps`.

[16] OXFORD Hardware Compiler Group. The Handel language. Technical report, Oxford University, 1997.

[17] J. R. Gurd, C. C. Kirkham, and I. Watson. The Manchester prototype dataflow computer. *Communications of the ACM*, 28(1):34–52, January 1985.

[18] J. Hammes and Böhm. *The SA-C Compiler DDCF Graph Description*, 1999. Document available from `www.cs.colostate.edu/cameron`.

[19] J. Hammes and Böhm. *The SA-C Compiler - Version 1.0.242*. 2000. Document available from `www.cs.colostate.edu/cameron`.

[20] J. Hammes and W. Böhm. *The SA-C Language - Version 1.0*. 1999. Document available from `www.cs.colostate.edu/cameron`.

[21] J. Hammes, L. Lubeck, and A. P. W. Böhm. Comparing Id and Haskell in a Monte Carlo photon transport code. *em Journal of Functional Programming*, 5(3):283–316, July 1995.

[22] J. Hammes, R. Rinker, W. Böhm, and W. Najjar. Cameron: High level language compilation for reconfigurable systems. In *PACT'99*. Oct. 1999.

[23] J. Hammes, R. Rinker, D. McClure, Böhm, and W. Najjar. *The SA-C Compiler Dataflow Description*. 1999. Document available from `www.cs.colostate.edu/cameron`.

[24] J. Hammes, S. Sur, and A. P. W. Böhm. On the effectiveness of functional language features: NAS benchmark FT. *Journal of Functional Programming*, 7(1):103–123, January 1997.

[25] S. Hauck. The roles of FPGAs in reprogrammable systems. *Proceedings of the IEEE*, 84(4):615–639, April 1998.

[26] J. Hicks and et al. Performance studies of Id on the Monsoon dataflow system. *Journal of Parallel and Distributed Computing*, 18:273–300, 1993.

[27] P. Hudak, S. Peyton Jones, and P. Wadler. Report on the programming language Haskell, a non-strict purely functional language (version 1.2). *ACM SIGPLAN Notices*, 27(5), 1992.

[28] Xilinx Inc. Xilinx delivers two million gate Virtex solution for gigabit per second applications. Press release, September 1999. Available from `www.xilinx.com/prs_rls/vtxe.htm`.

[29] Xilinx Inc. Sales of Xilinx Virtex FPGAs surpass $100 million. Press release, February 2000. Available from `www.xilinx.com/prs_rls/virtex100m.htm`.

[30] B. Kernighan and D. Ritchie. *The C Programming Language*. Prentice Hall, second edition, 1988.

[31] K. Konstantinides and J. Rasure. The Khoros software development environment for image and signal processing. In *IEEE Transactions on Image Processing*, volume 3, pages 243–252, May 1994.

[32] J. McCarthy and et al. *LISP 1.5 programmers manual*. 1962.

[33] J. McGraw and et al. *SISAL: Streams and Iteration in a Single Assignment Language: Reference Manual Version 1.2*. Lawrence Livermore National Laboratory, memo m-146 rev. 1 edition, 1985.

[34] W. Najjar. The Cameron Project. Information about the Cameron Project, including several publications, is available at the project's web site, `www.cs.colostate.edu/cameron`.

[35] R. Nikhil. *Id Version 90.0 Reference Manual*. Computational Structures Group Memo 284-1, Massachusetts Institute of Technology, 1990.

[36] D. Perry. *VHDL*. McGraw-Hill, 1993.

[37] J. M. S. Prewitt. Object enhancement and extraction. In B. S. Lipkin and A. Rosenfeld, editors, *Picture Processing and Psychopictorics*. Academic Press, New York, 1970.

[38] Red hat linux 6.0. Information is available at the web site, `www.redhat.com`.

[39] R. Rinker, G. Yarbrough, and W. Najjar. Implementation of the Prewitt Algorithm on the Wildforce-XL reconfigurable computing board. Technical report, Colorado State University. Computer Science Department. 1999.

[40] S. K. Skedzielewski and J. R. W. Glauert. *IF1. an Intermediate Form for Applicative Languages.* Lawrence Livermore National Laboratory, reference manual m-170 edition. July 1985.

[41] Synplicity. Inc. *Synplify User's Guide. Release 5.2.2.* 1999. `www.synplicity.com`.

[42] M. Wolfe. *High Performance Compilers for Parallel Computing.* Addison-Wesley Publishing Company. 1996.

[43] Xilinx. Inc., San Jose. CA. *The Programmable Logic Databook.* 1998. `www.xilinx.com`.