The SA-C Compiler – Version June, 2001

J. P. Hammes, Monica Chawathe and A. P. W. Böhm
Colorado State University

June 21, 2001

This document describes the use of the SA-C compiler and its related software. The language, SA-C, is described separately.

1 Overview of SA-C compilation system

A pure SA-C compiler lives at the heart of the compilation system. It is supplemented by a generic transformer, macro preprocessor, dataflow simulator and run-time system. All of these are coordinated by a Perl script called **scc** that handles the various user-controlled options, calls the various components in the proper order, and manipulates temporary intermediate files.

2 Using the SA-C compilation system

To use the SA-C compiler and dataflow simulator, the environment variable SASSYHOME must be set with the path to the root directory of the SA-C compiler. To run the compiler, execute

$SASSYHOME/scc

Most users will set the environment variable in a `.cshrc` file, and create an alias for **scc**. The compiler flags can be viewed by executing **scc** with no arguments, and consist of the following:

-G	Compile from generic *.sg files to *.sc files (Not present at all)
-C	Compile to *.c files.
-c	Compile to *.o files.
-ddcf	Compile to *.ddcf files.
-dfg	Where possible, compile loops to *.dfg files. The rest of the code compiles to *.c files. This option also invokes optimizations. If any of the hardware flags (-aha, -vhdl, -edf, -x86, -b86, -bo or -bout) are also present, it will override the -dfg flag and produce an executable for running on FPGAs or AHA simulator. Otherwise, dataflow graphs will be generated and the executable will invoke the dataflow graph simulator.

Table 1: Compiler Goal Flags

-aha	Compile the dataflow graphs into abstract hardware architecture (AHA) graphs. If a -vhdl, -edf, -x86, -b86, -bo, -bout flag is also present, it will override the -aha flag and produce an executable for running on FPGAs. Otherwise, AHA graphs will be generated and the executable will invoke the AHA simulator.
-vhdl	Produce *.vhd files for AHA graphs that were generated. The host code that is produced will be compiled so as to invoke FPGA execution of converted loops.
-edf	Produce *.xnf files from the VHDL files that were generated. The host code that is produced will be compiled so as to invoke FPGA execution of converted loops.
-x86	Produce *.x86 files from the *.edf files that were generated. The host code that is produced will be compiled so as to invoke FPGA execution of converted loops.
-b86	Produce *.b86 files (*.x86 files bundled with tag information)
-bo	Produce *.bo file (*.o file bundled with corresponding *.b86 files)
-bout	Produce *.bout file (a.out file bundled with corresponding *.b86 files)
-ws	Compile hardware-related files for the WildStar board (The default is to compile for the StarFire board).
-vsim	Generate files used in ModelSim simulation.
-LAD <freq>	Set the LAD frequency to <freq>. (Only 33MHz and 66MHz allowed, with the default being 33MHz).
-mf	Generate and use a makefile for generating *.x86 files instead of the 'scc' script.

Table 2: Hardware Related Flags

-O	Turn on DDCF graph level optimizations.
-fs	Turn on final-srunching.
-pipe <num>	Pipeline the AHA graph with a maximum of <num> pipeline stages.
-ndv	Turn off use of dope vectors.
-nbw	Turn off bitwidth narrowing.
-extv	Print generator and loop information after each loop related optimization.

Table 3: Optimization Flags

-nbc	Turn off array bounds checking.
-nchk	Turn off DDCF consistency checks.
-dfg_report	Print why loops did not get converted into DFGs.

Table 4: Check Flags

-D <xx>[=<yy>]	Pass a #define to the 'cpp' preprocessor.
-U <xx>	Pass a #undef to the 'cpp' preprocessor.
-I <path>	Add <path> to the list of directories searched by 'cpp' for #include files.
-L <path>	Add <path> to the list of directories searched by 'cpp' for library files.
-l <lib>	Add <lib> when linking.
-gmp	Turn on support of ¿ 32 bits and link the gmp library.
-heap	Produce code to generate a heap trace.
-ggdb	Produce code to generate a 'gdb' debug trace.

Table 5: C, cpp, Linker Flags

-o <filename>	Use the specified <filename> to name the target file.
-host	Do not compile any hardware related files (.aha, .edf, .x86, etc.), only recompile the C code generated.
-buf <sz>	Set the buffer size for 'buffered concats' to <sz>.
-m	The main function name is equal to the SA-C file name, without the .sc extension. This is useful in the Khoros workspace compiler environment.
-q	Suppress error and optimization reports from the compiler. This is useful for automated test scripts. The compiler's system return value indicates whether the compile produced any errors.
-k	Keep the intermediate files produced by 'scc' script.
-v	Print the commands called by the scc script before executing them.
-n	Print the commands called by the scc script but do not execute them. (You just print some stuff, not the actual commands)

Table 6: Miscellaneous Flags

The compiler, see figure 1, takes one or more source files as arguments; each can be generic SA-C (.sg), SA-C (.sc), DDCF graph (.ddcf), C code (.c), or object code (.o). The file suffix determines the entrance point in the compile process. The `-c`, `-G`, `-ddcf`, `-C`, `-dfg`, `-aha`, `-vhdl`, `-edf`, `-x86`, `-b86`, `-bo` and `-bout` options, described below, determine the stopping points of the compile process. Some examples show how the process works:

- "`scc -ddcf file1.sg file2.sc`" compiles the files to DDCF graphs.
- "`scc -C file1.sc file2.ddcf`" compiles a SA-C file and a DDCF file to C files.
- "`scc file1.c file2.c file3.ddcf file4.o -o run`" compiles the four files to an executable called `run`.
- "`scc -dfg file.sc`" compiles the SA-C file to an executable with dataflow simulation of the *.dfg files that are produced.
- "`scc -x86 file.sc`" compiles the SA-C file to an executable with reconfigurable hardware execution of the loops that were translated to *.x86 files.

Note that all arguments following the `-cflags:` argument are passed to the C compiler, so these must be specified at the end of the command line.

3 Running a SA-C Program

When a SA-C program is compiled to an executable and run, its input data become the arguments to the "main" function, and that function's return values become the program's output. This I/O can be handled in two ways: via `stdin` and `stdout`, or by specified data files. Both approaches can use ASCII formats but the data file specification can also use binary file formats. The ASCII file format is intended to be used for test purposes only, and the input will be read sequentially and completely before program execution starts.

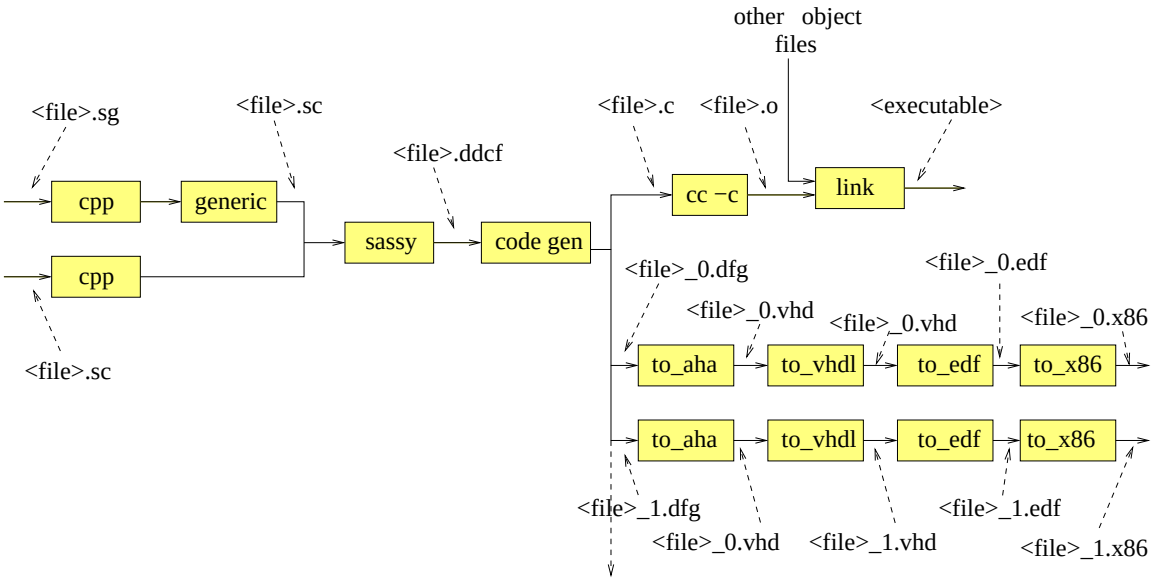

Figure 1: Compilation routes of "scc".

The ASCII (text) representations of program input are similar, but not identical, to the representations of values within SA-C source code. The differences occur with boolean values and array specifications. First, boolean values in input files may use the letters 'T', 't', 'F', and 'f' to signify "true" and "false" values. Second, array expressions in program input must be preceded by their extents. For example:

```
[2,5] {{3, 7, 9, 8, 4},{1, 3, 2, 4, 4}}
```

3.1 I/O using stdin and stdout

The default approach to I/O is to use `stdin` and `stdout` for input and output. Of course, redirection of either or both of these allows the use of source and target files through the user's shell. For example, let the file containing a matrix multiply executable be called "mm", and let the file mm.dt contain:

```
[3,2] { {2,1}, {3,4}, {6,2}}
[2,5] { {2,5,1,9,3}, {5,3,7,4,2} }
```

Then, `mm <mm.dt` will produce

```
[3,5]
{ { 9, 13, 9, 22, 8}
  , { 26, 27, 31, 43, 17}
  , { 22, 36, 20, 62, 22}
  }
```

on standard output.

3.2 I/O via specified files

A second approach to I/O specifies source and target information as arguments to the executable; each input parameter and each output value gets its own specification, typically a file name.

Input parameters on the command line have the following format:

```
-i<specifier><number>  operand
```

The number (zero-based as in C) specifies a main function input parameter. If the specifier is 'f', the operand specifies a file name. If the specifier is 'v', the operand is a value, denoted as it would be in an input file. (More complex input values may need to be surrounded by quotes in certain shells or operating systems.) Each input file can be either ASCII or binary.

Output parameters on the command line have the following format:

```
-of<kind><number>  filename
```

The "kind" specifier is either 'a' for ASCII or 'b' for binary.

Continuing the matrix multiply example above:

```
mm -if0 mmA.dt -if1 mmB.dt -ofa0 mm.res
```

would take its inputs from the files `mmA.dt` and `mmB.dt` and put its result, in ASCII format, in the file `mm.res`.

3.3 Binary file format

The SA-C binary file format is designed to be compatible with *pgm* and *ppm* image files. The additional information needed by the SA-C run-time system is put in a comment line. Figure 2 shows an example of a binary file with its text preamble. The SA-C system pays attention only to its special commented line and the block of binary data. The necessary comment line contains:

- the string "SA-C"
- a format indicator, currently "format1"
- a SA-C type, with all extents specified; it will be one or two tokens, depending on whether or not the type is complex
- an endian indicator, 'E' for big-endian and 'e' for little-endian

This comment line can appear anywhere among the other comment lines of the file. It is important to note that SA-C's extents occur in reverse order as compared to the two dimensions that are specified to *pgm*.

In the case of a P5 or P6 file, the SA-C information line may be omitted; data type `uint8` will be assumed. This makes it possible to read and write *pgm* and *ppm* files without modification.

```
P5
# this is a comment line
# SA-C format1 uint8[8,15] E
# this is another comment line
15 8
255
< binary data >
```

Figure 2: An example showing binary file format. Note that the SA-C dimensions are in opposite order to those that occur outside of the comment area.

3.4 Run-Time Options

The run-time options consist of the following flags:

-time	To print number of clock-cycles required for execution.
-v	To print the function name at start and end of each host-interface functions.
-view	To run the DFG simulator with a viewer.
-debug	To print host to simulator/ hardware communication in 'hosttrace' file.
-trace	To print data generated in the simulation of graphs at every clock cycle (useful in hardware debugging).
-vsim	To print data-files used in ModelSim Simulation.
-oc	To overclock the board (at 25 MHz) in case the frequency from place and route is under 25MHz.

Table 7: Run-Time Options

4 Tools

The tag information can be grabbed from the bundled files (*.b86, *.bo, *.bout) using the utility tool $SASSYHOME/bundler/bndl_info. The tag cantains hardware occupancy and frequency given by both Synplify and Place and Route tools along with time required by each phase during compilation. It also conatins the scc command line used to generate the bundled file.

5 Generic SA-C Code

Given SA-C's size- and precision-parameterized type system and the need to create many versions of programs with related types (e.g. an Image Processing routine manipulating matrices of **uint4**, **uint8**, **uint16**, or **uint24** data) it is useful to be able to generate these various SA-C programs from one source. The *cpp* macro facility could be used for this, but it has drawbacks: A C macro is one line, perhaps with backslashes used for layout. Error messages refer to invocations of the one line macro, not to the macro's source text. These problems make the use of the C preprocessor unattractive for producing multiple versions of a SA-C function, so a generic text expansion mechanism has been provided in the SA-C compilation system. Generic SA-C code has the following form:

```
$generic
  $param1='text11'  ,  $param2'text12'  ...$paramN='text1N'  ;
  $param1='text21'  ,  $param2'text22'  ...$paramN='text2N'  ;
    ....
  $param1='textM1'  ,  $param2'textM2'  ...$paramN='textMN'  ;

$in

  sassy code with $param1 ... $paramN

$end_generic
```

The keyword **$generic** starts a generic function definition, **$end_generic** ends it. Each line between **$generic** and **$in** declares the values of a set of macro parameters. For each line the SA-C text between **$in** and **$end_generic** will be expanded. These generic constructs cannot be nested. Error messages refer to the file containing the generic SA-C code, the function that causes the error, and the line number in the generic SA-C file where the error occurs. Generic SA-C files have a suffix ".sg".

Here are some examples of generic SA-C code. The generic code appears on the left, and its SA-C equivalent on the right. First, a straightforward definition of one generic function:

```
/* Generic Code */

$generic
  $saxpy='saxpy8'  , $ti='uint4',$tr='uint8';
  $saxpy='saxpy12' , $ti='uint6',$tr='uint12';
  $saxpy='saxpy16' , $ti='uint8',$tr='uint16';
$in

$tr $saxpy ($ti x, $ti Y[:]) {
 $tr r = for y in Y return(sum(x*y));
} return(r);

$end_generic
```

```
/* SA-C Code */

uint8 saxpy8 (uint4 x, uint4 Y[:]) {
  uint8 r = for y in Y return(sum(x*y));
} return(r);

uint12 saxpy12 (uint6 x, uint6 Y[:]) {
  uint12 r = for y in Y return(sum(x*y));
} return(r);

uint16 saxpy16 (uint8 x, uint8 Y[:]) {
  uint16 r = for y in Y return(sum(x*y));
} return(r);
```

Not only types can be generic, any piece of SA-C text can be. For example,

```
/* Generic Code */

$generic
  $or= 'boolor'  , $t='bool'    , $op='||'  ;
  $or= 'uint1or' , $t='uint1'   , $op='|'   ;
$in

$t $or ($t x, $t y) return(x $op y );

$end_generic
```

```
/* SA-C Code */

bool boolor (bool x, bool y)
   return(x || y );

uint1 uint1or (uint1 x, uint1 y)
   return(x | y );
```

6 Optimizations, Pragmas and Internal graphs

In the SA-C compiler, optimizations are triggered by any of the following compiler options: **-O**, **-dfg**, **-vhdl**, **-edf**, **-x86**, **-b86**, **-bo** and **-bout**. The optimizations can be further controlled by pragmas in the SA-C source code. Certain loops can be compiled to dataflow graphs (DFGs) for execution by a simulator or for compilation to VHDL (via AHA graphs), as described below.

6.1 Optimizations

A variety of optimizations are performed at DDCF graph level. (Compilation with **-O -ddcf** options will write DDCF files of the optimized graphs.) Optimization passes consist of the following:

- *Invariant Code Motion* hoists invariant code out of loop bodies.
- *Push Array Casts* can, under certain conditions, push array casts to the places in which the array is being referenced, removing the need to do an array copy.
- *Constant Switch* takes a switch graph with a constant key value and replaces the graph with the case graph corresponding to the key.
- *Size Propagation* analyzes loops and arrays, propagating extents information through the graph in order to enable full loop unrolling.
- *Array Value Propagation* replaces an array reference with its corresponding expression when possible.
- *Constant Folding* replaces an operator with a value if the operator's inputs are constants.
- *Identities* applies various algebraic identities and other rules. For example, $a * 1$ is replaced with a, and $a * 0$ is replaced by zero.
- *Strength Reduction* replaces certain operators with simpler equivalents. For example, multiplying an unsigned int by a power of two is replaced by a left shift.
- *Dead Code Elimination* removes code if its values are not used in producing the output results.
- *Full Loop Unrolling* replaces a loop with multiple explicit loop bodies if the number of loop iterations is statically known.
- *Canonify Macros* orders the input edges in increasing order of the source nodes and source ports (to allow CSE of part of the macro).
- *Common Subexpression Elimination (CSE)* eliminates redundant nodes if they compute the same value. This is done by looking for DDCF nodes with the same operation and the same inputs.
- *Extent Elision* replaces extents node (being fed by an array reference) by extent calculation logic, if possible.
- *Window To Element Gen* converts window generators that take single element windows to element generators. These situations arise as a result of the application of other optimizations.
- *Duped Inputs* removes the inputs being fed into a loop but not used in it.

- *Concat Masked* converts a concat or concat-masked node being fed by array definitions with a concat-masked-many node and removes the array definitions.
- *Window Narrow* decreases size of the window (and also image) in the innermost dimension if the pixels are not being used and increases the step accordingly. This lessens the FPGA space needed to hold the window values.
- *Function Inlining* replaces a function call with the function's body.
- *Lookup Tables* replaces function calls with references to a lookup table that holds precomputed values.
- *Nextified Array Size Propagation* analyzes nextified arrays, propagating extents information through the graph in order to enable converting the array into nextified scalars.
- *Temporal CSE* looks for values computed in one loop iteration that were computed in earlier iterations (only in the innermost dimension) and replaces the redundant computation by a chain of registers (of compile-time fixed length). This leaves some window elements (in the innermost dimension) unreferenced thus allowing window narrowing.
- *Scrunch* pushes computation of expressions fed by window elements into earlier iterations by moving the window references (in the innermost dimension) and using a register chain to move the result. This also creates a window that can be narrowed.
- *Array Reference Elision* replaces two array references, if the first takes a slice and it feeds only the second, with a single reference.
- *Convert Nextified Array* convert a nextified array of known size into nextified scalars.
- *Loop Fusion* can, under certain circumstances, fuse two consecutive loops into one loop.
- *N-Dimensional Blocking* stripmines a loop by enclosing it in another loop that reads chunks of data with which to feed the inner loop.
- *N-Dimensional Stripmining* stripmines a loop by enclosing it in another loop that reads chunks of data with which to feed the inner loop. The inner loop will subsequently be unrolled, yielding the effect of partial unrolling in multiple dimensions. It is parameterized by the outer-loop window size.
- *N-Dimensional Partial Unroll* stripmines a loop by enclosing it in another loop that reads chunks of data with which to feed the inner loop. The inner loop will subsequently be unrolled, yielding the effect of partial unrolling in multiple dimensions. It is parameterized by the inner-loop iterations.
- *CSE Macro* identifies macro nodes being fed by some identical inputs and replaces eliminates partial redundant computations.
- *Final Scrunch* is similar to scrunch except that it scrunches all computations into a window of size one in the innermost dimension.
- *Pipeline* inserts registers across AHA graphs to minimize the critical path delay.

Optimizations interact with each other. A sequence of a subset of optimizations forms *cyclic opts*, with the sequence repeated until stability is reached (i.e. a pass occurs in which the DDCF is not changed). The cyclic sequence is

1. Code Motion
2. Push Array Casts
3. Constant Switch
4. Size Propagation
5. Array Value Propagation
6. Constant Folding
7. Identities
8. Strength Reduction
9. Dead Code Elimination
10. Canonify Macros
11. CSE
12. Extent Elision
13. Window To Element Gen
14. Duped Inputs
15. Concat Mask
16. Window Narrow

The full optimization path is as follows:

1. Function Inlining (skipping lookup functions)
2. Lookup Tables
3. Function Inlining (inlining loopup functions)
4. Nextified Array Size Propogation
5. Temporal CSE
6. Scrunch
7. Array Reference Elision
8. *cyclic opts*
9. Convert Nextified Array
10. *cyclic opts*
11. Loop Fusion
12. Temporal CSE
13. Scrunch
14. *cyclic opts*

15. Array Reference Elision
16. *cyclic opts*
17. N-Dimensional Blocking
18. N-Dimensional Stripmining
19. N-Dimensional Partial Unroll
20. Array Reference Elision
21. *cyclic opts*
22. CSE Macro
23. Final Scrunch
24. *cyclic opts*

This path is arranged as it is because of certain interactions among optimizations. The first of the Function Inlining passes (step 1) skips the inlining of functions designated as lookup tables, but it does inline function calls that occur in the function body of the lookup table. The next pass (step 2) converts the designated functions into lookup tables, followed by inlining of these functions (step 3) to their call locations. The loop fusion (step 11), temporal CSE (step 12, scrunch (step 13), array reference elsion (step 15) and the cyclic optimizations (steps 14 and 16) are performed repeatedly until no more loops can fuse.

6.2 Pragmas

The compiler flags **-O**, **-dfg**, **-aha**,**-vhdl**, **-edf**, **-x86**, **-b86**, **-bo** and **-bout**enable optimizations; pragmas in SA-C source code allow control of individual optimizations (except that final scrunch is controlled by **-fs** flag). Some pragmas are associated with loops, others with functions. In either case the pragma appears immediately before the loop or function it is controlling. For example,

```
// PRAGMA (no_unroll, no_dfg)
for ...
```

will prevent the **for** loop from being unrolled or converted to a dataflow graph. As this example shows, multiple pragmas may be listed, and their order does not matter.

The pragmas are as follows:

1. **no_inline** indicates that a function should not be inlined. The default action is to inline all functions.
2. **no_unroll** indicates that a loop should not be unrolled. The default action is to fully unroll a loop where possible.
3. **nextify_cse** indicates that temporal CSE optimization must be done over the loop
4. **scrunch** indicates that scrunch optimization must be done over the loop;
5. **no_dfg** indicates that a loop should not be turned into a dataflow graph (DFG) for execution by the simulator. The default action is to turn a loop into a DFG if possible.

6. **lookup** indicates that a function should be converted to a lookup table which uses the physical memory of the hardware for implementation.

7. **rom** indicates that a function should be converted to a lookup table which uses the ROM on the hardware for its implementation.

8. **no_fuse** indicates that the loop should not be fused with another loop that is being fed by its output. The default action is to fuse loops wherever possible.

9. **stripmine** is a parameterized pragma that causes a loop to undergo N-Dimensional Stripmining. The inner loop subsequently will be fully unrolled. The parameters indicate the size of window used in the new outer loop.

10. **block** is a parameterized pragma that causes a loop to undergo N-Dimensional Stripmining for array blocking. The inner loop will not be unrolled. The parameters indicate the size of window used in the new outer loop.

11. **part_unroll** is a parameterized pragma that causes a loop to undergo N-Dimensional Stripmining. The inner loop subsequently will be fully unrolled. The parameters indicate the iterations of the inner-loop.

12. **vhdl** is attached to a SA-C function prototype, and indicates that the function call will be passed through the compiler and into the dataflow graph, to be filled in externally with an outside VHDL routine.

The optimization order described previously will determine exactly what is done. For example, if a loop can be fully unrolled and it is given a **stripmine** pragma, the loop will be unrolled (since unrolling occurs earlier than stripmining) and the pragma will be lost since the loop no longer exists.

The Loop Fusion optimization fuses loops that have a producer-consumer relationship, i.e. the output of the "upper loop" feeds the input of the "lower loop". In the transformation, a new outer loop is created, enclosing the two loops. The lower loop is then melted away, leaving only its body. The new outer loop is given the pragmas of the upper loop, and the upper loop loses all of its pragmas. The lower loop's pragmas are lost, since it no longer exists as a loop.

When N-Dimensional Stripmining takes place, a new outer loop is created, enclosing the original loop. Loops are also created to handle the "fringes" that may be left over since the window generator of the new outer loop has a non-unit step. The fringe loops are given **no_dfg** pragmas to assure that their computations, probably small, do not produce DFGs. The new outer loop is given copies of the original loop's pragmas, but with the **stripmine** pragma removed. (The **stripmine** pragma is also removed from the original loop.)

N-Dimensional Blocking is similar to stripmining, but the inner loop is given a **no_unroll** pragma by the compiler. Thus the outer loop will run on the host, and the inner loop will become a dataflow graph that computes with chunks of the source array.

6.3 Producing Dataflow Graphs (DFGs)

The **-dfg** compiler option causes the compiler to convert inner loops to dataflow graphs wherever it can. (The option also causes optimizations to occur, the same as if **-dfg -O** had been specified.) These DFGs will go into files with a **.dfg** suffix. When executing the SA-C program, these DFGs will be executed by a simulator whose behavior somewhat matches the execution on reconfigurable hardware.

6.4 Producing Abstract Hardware Architecture Graphs (AHAs)

The **-aha** compiler option causes the compiler to convert inner loops to dataflow graphs wherever it can and then convert the DFGs into abstract hardware architecture graphs. (The option also causes optimizations to occur, the same as if **-aha -O** had been specified.) These AHAs will go into files with a **.aha** suffix. When executing the SA-C program, these AHAs will be executed by a simulator whose behavior matches the execution on reconfigurable hardware (except for memory contection).

7 Installation

The SA-C compiler sources are contained in a tarred and compressed file called `sassy.june_xx_01.tar.Z`. When uncompressed and untarred, this will produce a SA-C home directory with subdirectories for various things such as the compiler, the run-time system, and test codes. The first installation step is to create the SA-C home directory and move the `sassy.june_xx_01.tar.Z` file into it.

Once the source file is in place, the following command can be used to build the directories of sources:

```
> zcat sassy.june_xx_01.tar.Z | tar xvf -
```

The next step is to compile the sources. While in the SA-C home directory, type:

```
> make
```

This will descend into the various subdirectories and compile their contents.

Since the SA-C compilation system is controlled by Perl scripts, two items must be dealt with. First, the `scc` script in the SA-C home directory must be told the location of the `cpp` macro preprocessor on the host system. This is done by editing the line:

$cpp = "/usr/local/cpp";

Second, the location of `perl` can vary from system to system. The first line of all the Perl scripts must hold the correct path to `perl`, and will have to be altered if `perl` does not live at `/usr/local/bin/perl`. The Perl scripts in the SA-C compiler are:

- `<SA-C home>`/scc
- `<SA-C home>`/test/sassy_test
- `<SA-C home>`/test/aha_test/sassy_test
- `<SA-C home>`/test/examples/sassy_test
- `<SA-C home>`/test/lib_test/sassy_test
- `<SA-C home>`/test/mix_test/sassy_test
- `<SA-C home>`/test/separate_comp/sassy_test
- `<SA-C home>`/test/sim_test/sassy_test

- `<SA-C home>`/test/test_IO/sassy_test

Once the system has been compiled and the Perl scripts altered, the compiler can be tested. First, while still in the SA-C home directory, execute:

> setenv SASSYHOME $PWD

Then move into the `test` directory and run the test script:

```
> cd test
> ./sassy_test
```

The script will descend into the subdirectories and run the compiler tests they contain.

Once the compiler has been shown to work correctly, refer to the material in section 2 to see how a user should use the system.