ROCCC 2.0 Developer's Manual - Revision 0.5.1

July 23, 2010

Contents

1 Introduction **3**

2 Installation **4**

3 Overall Compilation **5**
3.1 Specification Files 5
3.1.1 Compiler Optimization File 5
3.1.2 Timing Information File 6
3.2 Compiling Procedure 6
3.2.1 compile_suiftohicirrf 7
3.2.2 compile_llvmtovhdl 7

4 Gcc2SUIF **8**

5 Hi-CIRRF Compilation **9**
5.1 Hi-CIRRF Passes 9

6 Hi-CIRRF File Format **17**
6.1 Fakes 17
6.2 Functions 17

7 Lo-CIRRF Passes **21**
7.1 LLVM Changes 21
7.2 Passes 21

8 Database **26**

1 Introduction

This document describes the current implementation of the Riverside Optimizing Compiler for Configurable Circuits (ROCCC) and describes all of the transformations and processes that take a C file into VHDL. The information in this document is not needed in order to run and compile code with ROCCC, but is provided if you wish to improve or alter ROCCC to support some new constructs.

ROCCC is currently split up into several phases. The GUI is written as an Eclipse plugin and controls overall compilation and file management. The compiler proper contains several intermediate representations, which we refer to as Hi-CIRRF and Lo-CIRRF (CIRRF stands for Compiler Intermediate Representation for Reconfigurable Fabrics).

The Hi-CIRRF passes are responsible for high level compiler transformations and are implemented using the SUIF (Stanford University Intermediate Format) toolset. SUIF is an older toolset and we have made some modifications to update it, including changing some of the data structures into standard library calls. The Hi-CIRRF passes that transform the compiled code are described in Section 4.

The Lo-CIRRF passes are responsible for hardware generation and pipeline as well as low level optimizations to reduce hardware and increase throughput. The Lo-CIRRF passes are implemented using the Low Level Virtual Machine (LLVM) toolset. The Lo-CIRRF passes are described in Section 6.

The Hi-CIRRF passes communicate to the Lo-CIRRF passes through the creation of a hi_cirrf.c file. This file is a C file with extra macros and is described in Section 5.

If you wish to add a pass, it is important to place it in the correct location. Passes that transform the code in a traditional compiler sense or something similar should always be located on the Hi-CIRRF side. Any pass that effects the hardware generated in a direct capacity should take place at the Lo-CIRRF side.

Section 3 describes the overall compilation of a C module and system.

ROCCC has several implementation languages, including C, C++, python, bash scripting, SQL, and Java. Both the Hi-CIRRF and Lo-CIRRF sections are mainly written in C++, and as such any changes made to the compiler require a working knowledge of C++. The GUI is written in Java under the Eclipse plugin framework. The running and installing of the compiler is performed using bash scripts. A few python scripts are still available, but have been deprecated and will be removed in future releases.

2 Installation

Installation of ROCCC is performed by running the bash script file "rocccInstall.sh." This script performs the following tasks:

- Check Requirements

  ROCCC needs to find installed versions of flex, bison, gcc, g++, make, autoconf, and patch in order to fully install. If any of these components are not found then installation halts.

- Install Modified gcc 4.0.2

  The Hi-CIRRF portion of ROCCC requires a file in the SUIF format. In order to generate a SUIF script, we created a program that translates gcc's GENERIC abstract syntax tree into SUIF, which is described in Section 4. The Modified gcc-4.0.2 installed during this step has some changes to the abstract syntax tree output to pass more information to the Hi-CIRRF passes (localized in the file "tree-dump.c") as well as some changes to help compilation on multiple machines.

- Unpacks a version of gcc for Lo-CIRRF

  The Lo-CIRRF portion of ROCCC requires a file in the LLVM binary format. in order to generate this binary, we use the prepackaged binaries available on the llvm website. We determine the type of the host system and unpack the appropriate binary while removing all of the others.

- Compile the Hi-CIRRF section

  This step involves compiling both the SUIF distribution as well as all of the passes we have written.

- Compile Lo-CIRRF section

  This step involves compiling both the sqlite3 database we use as well as the LLVM code base.

- Initialize the Database

  At this point in the install script, a program to insert default floating point cores into the database is compiled and run. This program is located under roccc-compiler/src/tools.

- Create Scripts For Compilation

  Several scripts are created by the install script, including a file that will set up all the environment variables necessary to compile (setupRocccVariables.sh), a script to perform the Lo-CIRRF compilation (compile_llvmtovhdl.sh), a script to reset the compiler database to its install state (reset-compiler.sh), a script that removes a single module from the roccc-library.h file and corresponding repository (remove_module.sh), a script that adds a single module from the roccc-library.h file and corresponding repository (add_module.sh), and the script that controls overall compilation (compile_hardware.sh).

3 Overall Compilation

3.1 Specification Files

Before compilation can begin, two files must be specified. The GUI is the main interface to the compiler and is responsible for creating these files necessary for compilation. The creation of these files by the user is only necessary if the GUI is not being used.

Hi-CIRRF compilation requires an optimization file that lists the order and number of optimizations to perform. Lo-CIRRF compilation requires a file that describes the weights of every operation so retiming can be performed. The files are described in greater detail below.

3.1.1 Compiler Optimization File

This file is passed to the main compile script and specifies the number and order of the optimizations. As the compiler evolves, more optimizations will be added. The general layout of the compiler optimization file takes the form of:

```
Optimization [argument1][ [argument2]
Optimization [argument1][ [argument2]
...
```

All tokens are separated by white space and it is assumed that there is only one optimization per line. Some optimizations require no arguments while others might need one or two. The list of available optimizations with their arguments is:

- MultiplyByConstElimination
- DivisionByConstElimination
- LoopUnrolling [CLabel]
- LoopInterchange [CLabel] [CLabel]
- FullyUnroll
- SystolicArrayGeneration [CLabel]
- TemporalCommonSubExpressionElmiination
- LoopFusion
- Export

The value of CLabel should be a label of a for loop in the original C code that we are compiling. The value for Amount should either be a positive integer or the string ”FULLY”.

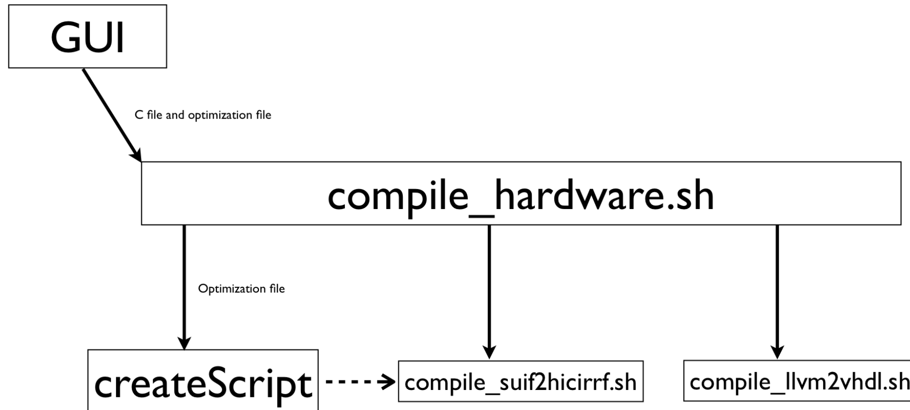

Figure 1: Overall Compilation Flow

3.1.2 Timing Information File

The Lo-CIRRF passes expect a specific file called "*/.ROCCC/.timinginfo" where "*" is the folder of the file being compiled. This file is laid out in the following way:

```
Operation Weight
Operation Weight
...
```

The specific operations that need to be specified are the following: "Add", "Sub", "Mult", "Div", "Compare", "Mux", "Copy", "Shift", "AND", "OR", "XOR", and "DesiredTiming". All weights must be positive integers. The weight for "DesiredTiming" must be greater than or equal to any other weights.

3.2 Compiling Procedure

Once the necessary information files have been created, the main compilation is handled by the script "compile_hardware.sh" which is created during installation. This script creates and calls other scripts as shown in Figure 1.

The first thing the compile_hardware.sh script does is call a program called "createScript" which takes the optimization file and creates a one-shot script called "compile_suiftohicirrf.sh" to process all of the Hi-CIRRF transformations in the correct order. After "compile_suiftohicirrf.sh" is called, two files called "hi_cirrf.c" and "roccc.h" are created. These two files (in addition to the timing info file) are passed to the script "compile_llvmtovhdl.sh" in order to generate VHDL. The last thing the compile_hardware.sh script does is clean up any temporary files and move all generated VHDL files into a local directory named "vhdl."

3.2.1 compile_suiftohicirrf

This script first compiles the original C file with the modified gcc 4.0.2 and creates an abstract syntax tree file with the extension .t02.original. This is then read by gcc2suif and creates a .suif file. This file is then passed to the suifdriver program with a list of all the passes to perform in order. The result of this are two files called "hi_cirrf.c" and "roccc.h."

3.2.2 compile_llvmtovhdl

The script that handles Lo-CIRRF compilation first compiles the hi_cirrf.c file using the llvm-gcc appropriate for the host machine. Then the script calls the llvm program "opt," calling all of the individual passes in the correct order. The result of calling this script is either one VHDL file for a compiled module or four VHDL files for a compiled system.

4 Gcc2SUIF

In the ROCCC distribution there is a bash script called "gcc2suif" and a directory called "gcc2suif" with code that compiles into a program called "parser." The script does some cleanup on generated files and calls "parser," so the tool described in this section refers to the code implementation in the gcc2suif directory.

The code used in converting gcc into suif can be viewed as a standard compiler in its own right. The input language is the abstract syntax tree of GCC as output by the –dump-tree-original-raw flag, the implementation language is C++, and the target language is SUIF. In order to accomplish this, gcc2suif uses flex and bison to describe the language of the abstract syntax tree and directly creates suif code using the suif toolset. Not every possible node output by gcc is supported. There is one ambiguity in the recognizable abstract syntax tree, the letter 'C' can either be a string or a keyword and it is impossible to tell based upon context. Because of this, we translate all variables with the name 'C' into something else and it is not recommended to use a variable named 'C'.

The parser generated by bison will create a tree that is iterated through and three stages are performed: connecting, flattening, and generating suif.

A program consists of a list of functions. A function contains a vector of all of the nodes that make up the function definition. A node is a hierarchical polymorphic data structure that forms a tree. Each node contains a number as referred to by the abstract syntax tree we read in. We then connect all of the nodes to form a tree as opposed to a flat list. We then flatten the tree to remove some of the indirection (nodes point to options which point to nodes, this is reduced to nodes pointing directly to nodes). Each type of instruction generated by the gcc abstract syntax tree has its associated node type, all of which inherit from the base node class.

Once the tree has been fully constructed, the tree is recursively traversed and a suif format tree is generated.

5 Hi-CIRRF Compilation

5.1 Hi-CIRRF Passes

All Hi-CIRRF passes are located in a subdirectory under roccc-compiler/src/hi_cirrf_pass/basepasses. There are several files that are currently not used, but still included because they might be used in the future.

Each pass is responsible for one transformation of the C code.

- EvalTransformPass

  Location: global_transforms

  Purpose: Replace all SUIF evaluation statements generated by the gcc2suif tool and replace them with an appropriate Call Statement, Store Statement, or Store Variable Statement.

  This pass exists to because the structure of the abstract syntax tree generated by gcc might produce evaluation statements for call expressions that do not return a value when this is handled by a Call Statement with a NULL destination in SUIF.

- ForLoopPreprocessingPass

  Location: gcc_preprocessing_transforms

  Purpose: Find all for loops and move the statement directly before the for loop into the for loop structure itself.

  This pass exists because of the different way that gcc and SUIF handle for loops. SUIF places the first statement of the for loop as a child of the for loop node, gcc moves the statement before the for loop and does not treat it as a child of the for loop.

- FlattenStatementListsPass

  Location: utility_transforms

  Purpose: Removes blank statement lists and compresses statement lists that contain statement lists into one list.

  Several of our passes create empty statement lists or statement lists that contain other statement lists. This pass is responsible for flattening them out into sensible statement lists.

- NormalizeStatementListsPass

  Location: utility_transforms

  Purpose: Makes sure that every for loop and every if statement has a statement list as a body and not just a single statement.

  In a lot of the ROCCC passes we make certain assumptions about the structure of the code we are processing. One of those is that every loop and if statement will contain a statement list and not a singular statement as its body. This pass makes sure that the program is transformed into such a style.

- PointerConversionPass

  Location: global_transforms

  Purpose: Turn all pointer accesses into array references.

  This pass is part of the uncompleted set of passes to allow systems to be composed of other systems. Generic pointers are still not supported by ROCCC 2.0.

- UnrollPass2

  Location: loop_transforms

  Purpose: Unrolls a given loop by a given amount. Does not create other loops to handle excess loop iterations.

  This pass takes two arguments, the label associated with the loop that we wish to unroll and the number of times to unroll the loop. If the number of times to unroll is greater than the end limit of the loop we fully unroll. This pass assumes that all loops start at 0 and count up to some number in the condition, which is checked with a less than comparison.

- HandleCallStatements

  Location: loop_transforms

  Purpose: When loops are unrolled, all call statements that have outputs need to be replicated. This pass takes care of that.

  Code that is passed to LLVM is translated into single static assignment form. To the LLVM side, multiple uses of the same variable are not transformed into multiple definitions as we would like, causing problems. This pass makes sure that if a variable is written to from two distinct call statements these values are split into distinct variables so they cause no conflict when passed to LLVM.

- HandleCopyStatements

  Location: loop_transforms

  Purpose: When loops are unrolled, copy statements require things to be put in a sort of SSA state.

  This pass performs a very similar operation for store variable statements. All variables that have multiple definitions (not if statements) are split into distinct variables. This is mainly a problem thanks to loop unrolling.

- IdentificationPass

  Location: verifyPass

  Purpose: Analyzes the code and determines if we are compiling a module, system, or composite system.

  Due to previous transformations, the type determined by the IdentificationPass might not be what the code started out as (i.e. Systems transformed into Modules through full loop unrolling).

- ControlFlowSolvePass

  Location: control_flow_analysis

  Purpose: Perform control flow analysis on the graph.

  This pass annotates the SUIF graph with information regarding control flow information. These annotations must be remade after any transformation that duplicates or rearranges code.

- DataFlowSolvePass2

  Location: bit_vector_data_flow_analysis

  Purpose: Perform data flow analysis on the graph and treats call statements as modules correctly.

  This pass annotates the SUIF graph with information regarding data flow. The control flow solve pass must be run before this pass. The information gathered in this pass must be redone after any transformation that duplicates or rearranges code.

- UD_DU_ChainBuilderPass2

  Location: bit_vector_data_flow_analysis

  Purpose: Sets up the use-def and def-use chains between all uses and definitions. All uses are in LoadVariableExpressions and all definitions are in StoreVariableStatements and CallStatements.

  All use/definition chains and definition/use chains are stored as annotations on both the variable symbols and the statements themselves. The data flow solver must be run before this pass. Any information gathered in this pass must be redone after any transformation that duplicates or rearranges code.

- ConstantPropagationAndFoldingPass

  Location: global_transforms

  Purpose: Performs constant propagation and folding.

  This pass does not propagate or fold any constant value that is located inside of a constant array.

- ConstantQualedArrayPropagationPass

  Location: array_transforms

  Purpose: Propagates constant values that are in constant qualified arrays.

  This pass does not perform any constant folding.

- TransformUnrolledArraysPass

  Location: fifoIdentification

  Purpose: When loops have been unrolled fully, arrays need to be changed from array access into either scalars or array accesses of lower dimension.

  This pass must be called after constant qualified array propagation.

- LoopInfoPass

  Location: control_flow_analysis

  Purpose: Annotate all of the for loops with the nesting level and any associated label.

- PreprocessingPass

  Location: data_dependence_analysis

  Purpose: Annotates all of the array references with the associated indices.

- ScalarReplacementPass

  Location: array_tranforms

  Purpose: Performs scalar replacement. All array accesses are replaced with scalars.

- IfConversionPass

  Location: global_transforms

  Purpose: Transforms if statements into boolean select statements.

- IfConversionPass2

  Location: global_transforms

  Purpose: Transforms if statements into boolean select statements.

  This pass converts if statements with exactly one assignment to the same location in both the "then" and "else" portion into a boolean select statement. This pass handles struct accesses and direct stores to memory for ROCCC 2.0 compatability.

- PredicationPass

  Location: global_transforms

  Purpose: Convert arbitrary if statements into predicated statements.

  The only type of if statement ROCCC 2.0 supports in hardware is the boolean select variety, where one variable gets assigned one of two different values based upon a condition. This pass converts arbitrary if statements into a sequence of predicated statements that fit the boolean select pattern.

- PseudoSSA

  Location: loop_transforms

  Purpose: Convert the body of the innermost loop into a single static assignment format and detect all possible feedbacks.

  This pass combines the tasks done in the HandleCopyStatements, HandleCallStatements, SolveFeedbackVariables, and SolveFeedbackCalls into one integrated pass.

- SolveFeedbackCalls

  Location: bit_vector_data_flow_analysis

  Purpose: Detect and split variables that act as feedback in all call statements.

- SolveFeedbackVaraiblesPass3

  Location: bit_vector_data_flow_analysis

  Purpose: Identify all of the variables that are read before they are written in the innermost loop.

  Variables are identified as feedback if they are read before they are written in the innermost loop and they do not have a definition before any uses.

- OutputIdentificationPass

  Location: global_transforms

  Purpose: Identify scalars that are either input or output. Input scalars are variables that are read before they are written in the innermost loop. Output scalars are variables that are written in the innermost loop that have no subsequent use (and are not used as feedback).

- FifoIdentification

  Location: fifoIdentification

  Purpose: Changes fifos into stream functions for the hi-cirrf generation.

- TransformSystemsToModules

  Location: global_transforms

  Purpose: When a system has all of its loops fully unrolled, no streams exist and all input and output values turn into scalars. The resulting code is treated as a module and compiled as such. This pass creates and fills up the struct that gets passed and returned.

- VerifyPass

  Location: verifyPass

  Purpose: Identifies if we are compiling a system or a module and verifies that all of the code was written correctly for the type of function.

- OutputPass

  Location: jasonOutputPass

  Purpose: Output the "hi_cirrf.c" and "roccc.h" files. This performs the actual output of the C code based upon the tree. After this pass all of the information we have collected is discarded.

- StripAnnotesPass

  Location: utility_transforms

  Purpose: Remove all of the annotations that we added to the suif graph.

- LibraryOutputPass

  Location: libraryOutputPass

  Purpose: Update the suif repository file and the "roccc-library.h" file.

- RemoveLoopLabelLocStmtsPass

  Location: utility_transforms

  Purpose: Removes labels from loops. This is necessary when unrolling an outer loop that contains an inner loop with a label and then trying to fuse all of the created loops.

- FusePass

  Location: loop_transforms

  Purpose: Fuse loops that have the same bounds and no dependencies.

- MEM_DP_BoundaryMarkPass

  Location: utility_transforms

  Purpose: Adds a mark statement between the hoisted memory reads and writes from the datapath proper.

- RawEliminationPass

  Location: array_transforms

  Purpose: Removes reads after writes.

- ScalarRenamingPass2

  Location: global_transforms

  Purpose: It eliminates all Anti and Output dependencies between scalar variables by renaming scalar variables.

- CopyPropagationPass2

  Location: global_transforms

  Purpose: Performs copy propagation that is compatible with ROCCC 2.0.

- SolveFeedbackVariablesPass2

  Location: bit_vector_data_flow_analysis

  Purpose: Deprecated pass that identifies feedback variables.

- FifoIdentifiactionSystolicArray

  Location: fifoIdentification

  Purpose: Creates fifos for systolic array generation. This involves splitting the two dimensional array into two separate one dimensional arrays, one for input and one for output.

- RemoveNonPrintablePass

  Location: jasonOutputPass

  Purpose: Finds all statements that have been previously annotated as "NonPrintable" and removes them. Some statements are not removed in certain passes but should not be output to the final hi_cirrf file. These are given the annotation "NonPrintable."

- LeftoverRemovalPass

  Location: global_transforms

  Purpose: Go through and remove the extraneous information generated during systolic array generation that might cause problems if passed to llvm. This includes all of the statements outside of the loops that were generated during an intermediate step.

- TemporalCSEPass

  Location: global_transforms

  Purpose: Perform temporal common subexpression elimination.

- RemoveUnusedVariables

  Location: global_transforms

  Purpose: To remove all of the variables that are no longer used in the resulting hi-cirrf file that is output from the symbol table.

- CleanupStoreStatementsPass

  Location: global_transforms

  Purpose: Change store statements that store into a load variable expression into a store variable statement.

  When arrays have been changed into scalars, we just replace all array reference expressions into load variable expressions. This is fine for most nodes, but store statements must be changed into store variable statements where appropriate.

- CleanupUnrolledCalls

  Location: loop_transforms

  Purpose: Create duplicate variables for destinations of call statements that have been unrolled.

- PreferencePass

  Location: jasonOutputPass

  Purpose: Process the preference file to control some optimization strategies and pass additional information to the lo-cirrf side.

- CastPass

  Location: jasonOutputPass

  Purpose: Locate all automatic casts between different types (including different bit sizes of integers) and create explicit calls so the lo-cirrf side has the correct information.

- IdentifyDebugPass

  Location: jasonOutputPass

  Purpose: Identify which variables correspond to variables marked for debugging in the GUI and pass that information on to the lo-cirrf side.

  This pass is not yet completed but is essential in the addition of debug variables to the overall ROCCC compilation.

Other passes are either no longer supported or will be adjusted and called in the future.

6 Hi-CIRRF File Format

Although you should not write Hi-CIRRF files directly, we describe in this section all of the statements that will appear in a Hi-CIRRF file.

The Hi-CIRRF that we generate consists of two files, roccc.h and hi_cirrf.c. The roccc.h file contains all of the declarations of the functions we use, although there is no corresponding definition of any of these functions. Hi-CIRRF itself is ANSI C with additional function calls that have no definition. Expressions and statements appear in hi-cirrf nearly identically to the original C after it has been transformed.

6.1 Fakes

Directly after the declarations in the Hi-CIRRF file are load instructions for most declared variables that we refer to in the code as "Fakes." These load instructions are in the form of "X = *((int*)(0)) ; " These instructions are necessary as the Hi-CIRRF code is not necessarily runnable C code, but rather an internal representation that we expect. When translating Hi-CIRRF into the LLVM binary format, any variable that does not have a definition before a use will be translated into an undefined reference. Because of this we make sure to give every variable a definition, even if it is meaningless.

6.2 Functions

There are several functions that we call in the Hi-CIRRF code that have no definition but do have a specialized meaning to the ROCCC compiler. Almost all of them exist to pass information discovered in the high level transformations directly to the code generation portion of the compiler. These functions are as follows:

- ROCCC_loadPrevious(int, int)

  When a variable is identified as a feedback variable, meaning the value calculated at the end of one iteration is used in the next iteration, we pass that value back through a temporary feedback register. A call to load previous tells the Lo-CIRRF passes what variable to associate with the previous iterations value. All loadPrevious calls also imply an initial value that is passed in as a scalar to be used before any values have been calculated.

- ROCCC_storeNext(int, int)

  The second half of feedback identification, a storeNext call will store the value calculated in one iteration into a variable for use in future iterations. All storeNexts happen at the end of the loop iteration at the same time.

- ROCCC_systolicNext(int, int)

A call to systolicNext has the same function as a storeNext call, with the exception that the systolicNext happens when the data is ready and doesn't get updated at the end of a loop iteration.

- ROCCCSystolicPrevious(int, int)

  A call to SystolicPrevious is the matching load that happens at the beginning of the loop for systolic array generation.

- int ROCCCInfinity()

  ROCCC supports infinite loops, but llvm expects for loops to appear in a very rigid format with a starting value of 0 and a step of 1. The end value for a loop must also be an integer, so when the high level transformations detect an infinite loop a call to the function ROCCCInfinity is output in place of an end value.

- float ROCCCFPToFP(float, int)

  When converting from a floating point value to a floating point value of a different bit width either through automatic or explicit casting, we output a call to a conversion function to instantiate specific conversion hardware. The second parameter is the bit size of the destination.

- int ROCCCFPToInt(float, int)

  When converting from a floating point value to an integer either through automatic or explicit casting we output a call to a conversion function to instantiate specific conversion hardware. The second parameter is the bit size of the destination.

- float ROCCCIntToFP(int, int)

  When converting from an integer to a floating point value either through automatic or explicit casting we output a call to a conversion function to instantiate specific conversion hardware. The second parameter is the bit size of the destination.

- int ROCCCIntToInt(int, int)

  When converting from an integer of one bit size to an integer of another bit size we output a call to a conversion function. On the lo-cirrf side this corresponds either to a sign extension, zero extension, or truncation. The second parameter passed is the bit size of the destination.

- ROCCC_output_C_scalar

  The high level optimizations detect all scalar values that are to be output once at the end of execution. These are explicitly identified in modules as variables in the struct with the suffix "_out" and inferred from live variables in systems. All variables that are output scalars are listed in a call to the function ROCCC_output_C_scalar in no particular order and used by the lo-cirrf side.

- ROCCC_init_inputscalar

  Similar to output scalars, the high level optimizations detect all scalar values that are read once at the beginning of execution. These are explicitly identified in modules as variables in the struct with the suffix "_in" and inferred in systems as all variables that are read but not written in the body of the innermost loop. All identified input variables are listed in no particular order in a function call to ROCCC_init_inputscalar.

- ROCCCName(const char*, ...)

  Every variable gets changed by LLVM into a temporary name. Passing the name as a string allows LLVM to find the original name and store it away.

- ROCCCModuleStructName(const char*)

  For generating a PCore, the GUI must know the name of the struct that is created with modules. The way we get that information back is through the database. The Hi-CIRRF section does not modify the database, so we pass the information along to the Lo-CIRRF side in this function.

- ROCCCPortOrder(const char*)

  The order of the ports in the structs for modules get transformed by the gcc abstract syntax tree and may not be in the same order as when they were defined in the original C. The Hi-CIRRF takes the port order that it knows as one big string and passes it to the LLVM side. The LLVM side then puts this information in the database so the GUI may access this to ease use of module instantiation.

- ROCCCInvokeHardware(const char*, ...)

  Every call to a hardware block is done by passing the name of the IP core as the first element of the InvokeHardware call. If this is not found in the database during the lo-cirrf compilation, an error will be raised. The values passed after the name consist of all of the inputs to the module followed by all of the outputs to the module.

- ROCCC_boolsel(int, int, int)

  If statements in the C code are transformed into calls to boolean select hardware (muxes) at the VHDL level. The actual transformation occurs at the hi-cirrf level so when the lo-cirrf manages the code there is no control flow resulting in phi nodes when dealing with SSA form.

- ROCCCSize(int, ...)

  Every variable has a bit width associated with it. If no bit width was explicitly specified then this value is the default of the machine compiling on. This function is used to pass the bit width information of each individual variable to the lo-cirrf passes.

- ROCCCStep(int, int)

  In order for LLVM to identify a for loop and collect information about that loop correctly, the for loop must have a step size of one. ROCCC supports any step size, so the hi-cirrf for loops are output with a step size of one and the actual step size is passed to the lo cirrf side through this function, which is called once per loop index.

- ROCCC_output_fifoX(int, ...)

  Any array we have identified as being written to is passed to the lo cirrf side in this function. The array should have had all of its memory accesses replaced with scalars. This function sends the original array name, the indices that all accesses are based off of, and the scalars that correspond to different offsets. The X is either 1 or 2, based upon the dimensionality of the array being passed.

- ROCCC_input_fifoX(int, ...)

  Nearly identical to the output fifo call, this function passes information about any arrays identified as input arrays.

- ROCCC_sbX(int, ...)

  This is identical to the output fifo call with the exception that there should be reuse between different iterations.

- ROCCCFunctionType(int)

  This passes 1 if we are compiling a module and 2 if we are compiling a system.

- ROCCC_state_scalar()

  This function is currently output but ignored and has no function.

- ROCCCNumIncWords(int, ...)

  This is called for every input stream and specifies the number of simultaneous incoming words that the stream expects every clock cycle.

- ROCCCNumMemReq(int, ...)

  This is called for every input stream and identifies the number of outstanding memory requests supported.

- ROCCCMaximizePrecision(int)

  This passes 1 to the low level optimizations to indicate if all arithmetic operations should increase their precision and round only on assignment or passes 0 to indicate rounding at every step.

7 Lo-CIRRF Passes

All lo-cirrf passes are located under the directory roccc-compiler/src/llvm-2.3.

7.1 LLVM Changes

LLVM was modified slightly to allow ROCCC to built on top of the framework that was already in place in LLVM. Specifically, the Function class and BasicBlock class were modified to allow two classes to each derive from them, DFFunction and DFBasicBlock respectively. These two classes contain additional ROCCC specific information, and are used to construct the ROCCC DFG using the LLVM CFG framework. By extending these two classes, it is also possible to detect when a given Function or BasicBlock object actually represents a ROCCC DFG construct, and not an LLVM CFG construct.

Because of linking issues with the version of gcc we use to compile LLVM, DFFunctions and DFBasicBlocks do not retain their runtime type information across module boundaries. To fix this, we had to add a single function to each of Function and BasicBlock that returned a DFFunction pointer or a DFBasicBlock pointer if the base object was actually of that type. Instead of dynamic_casting a Function or BasicBlock object to a DFFunction or DFBasicBlock, it is necessary to call the getDFFunction() or getDFBasicBlock() in the base class.

7.2 Passes

All of the Lo-CIRRF passes are implemented as a class derived from ModulePass or FunctionPass and are called from the program "opt". The passes we call, in order, are as follows:

- maximizePrecision

  Location: lib/Transforms/RocccCFGtoDFG/MaximizePrecision.cpp

  Purpose: Calling this pass tells the low level that all integer operations should use the maximum precision possible, truncating as a final step. Whether or not to call this pass is handled by the compile_llvmtovhdl.sh script.

- renameMem

  Location: lib/Transforms/RocccCFGtoDFG/RocccCFGtoDFG.cpp

  Purpose: Renames each load instruction to be a derivative of the first operand. This is not strictly necessary, but makes the resulting VHDL more readable.

- mem2reg

  Location: Builtin

Purpose: LLVM by default places all C variables into a memory location and need to be explicitly transformed into register uses. This pass performs this transformation and lumps all values without an initialization together as undefined.

- removeExtends

  Location: lib/Transforms/RocccCFGtoDFG/RemoveExtends.cpp

  Purpose: The low level can handle truncation and extension of integers to different sizes without the use of explicit extension operations. This pass removes any explicit extend operations, so as to minimize the overhead of unnecessary operations.

- ROCCCfloat

  Location: lib/Transforms/FloatPass/FloatPass.cpp

  Purpose: Floating point operations and integer divides are supported in the VHDL through IP cores. However, to support naturally written C code, these are written as C operators and have to be transformed into calls to an IP core. The FloatPass searches for instructions that are either integer divides or floating point operations, then replaces them with a call to the correct IP core from the database.

- flattenOperations

  Location: lib/Transforms/RocccCFGtoDFG/FlattenOperations.cpp

  Purpose: This optional pass performs tree balancing on commutative and associative operations, namely addition and multiplication, to perform as many possible operations in parallel, rather than serially.

- dce

  Location: Builtin

  Purpose: After flattening operations, dead code is generated. This pass eliminates that dead code, and cleans up for later passes.

- undefDetect

  Location: lib/Transforms/RocccCFGtoDFG/UndefDetectionPass.cpp

  Purpose: After running the mem2reg pass any variables that were declared in C code that have uses before any definition are referenced by the variable "undef." This pass searches all instructions and looks for any instance of an undefined variable and errors out if any are detected. If all goes well, this pass does not modify the code in any way.

- functionVerify

  Location: lib/Transforms/RocccCFGtoDFG/FunctionNameVerifyPass.cpp

  Purpose: This pass looks through the instructions and checks that any call instructions adhere to three rules: no call instructions are allowed except

for ROCCC specific call instructions, all calls to ROCCCInvokeHardware must exist in the database as an IP core, and all calls to ROCCCInvoke-Hardware must have the same number of arguments as the IP core has ports.

- rocccCFGtoDFG

  Location: lib/Transforms/RocccCFGtoDFG/RocccCFGtoDFG.cpp

  Purpose: This pass has several duties, all of which are related to transforming the LLVM-created control flow graph into a data flow graph with ROCCC specific additions. It requires two sub-passes, rocccFunctionInfo and rocccIfCompact, which are responsible for detecting and saving loop-specific information and for compacting the boolSelect calls, respectively. The ROCCC DFG uses one BasicBlock per instruction and connects the BasicBlocks to one another using switch statements. All of the ROCCC inputs, such as input FIFOs, loadPrevious calls, and input scalars are pulled to the top of the DFG and are direct successors of an empty BasicBlock called the sourceHead. All ROCCC outputs, such as output FIFOs, storeNext calls, and output scalars are pulled to the bottom of the DFG and are direct predecessors of an empty BasicBlock called the sinkHead.

  The ROCCC DFG is not a direct translation of the already-existing LLVM DFG ; several functions, such as ROCCCInputScalar are uses in LLVM, but are considered definitions in the ROCCC DFG.

- detectLoops

  Location: lib/Transforms/PipelinePass/DetectLoopsPass.cpp

  Purpose: Most of the later passes assume that the graph we create is acyclic, so this pass makes sure that no loops exist and errors out if there are any loops. This pass is a prerequisite for pipeline numbering, retiming, and inserting copies.

- pipeline

  Location: lib/Transforms/PipelinePass/RetimingPass.cpp

  Purpose: Up to this point, the pipeline separates all dependent instructions into their own pipeline stage. The pipelining pass combines instructions into a single pipeline level, allowing for a shallower pipeline at the potential cost of frequency. This is often a desirable optimization as there are usually other constraints that limit the frequency and so a shallower pipeline can reduce area and latency. This is a renumbering algorithm and does not reshape the DFG.

  The algorithm is based off of the FEAS algorithm presented in "Retiming Synchronous Circuitry" by Charles E. Leiserson and James B. Saxe.

  We first convert the ROCCC DFG into a more suitable, lightweight form to perform the retiming calculations on. We construct a graph in which

both the nodes and edges have an associated weight. The weight of each node is the delay of that node, which is user configurable. The weight of each edge is initially set to 0 and is used to represent the number of registers at that edge. The algorithm inserts registers until the period of the graph is less than or equal to the desired period.

- minimizeCopies

  Location: lib/Transforms/PipelinePass/CopyMinimization.cpp

  Purpose: This optional pass attempts to minimize the total number of register copies created, by moving the pipeline level that operations are performed in.

- insertCopy

  Location: lib/Transforms/PipelinePass/InsertCopyPass.cpp

  Purpose: For each edge in the DFG, if the difference in pipeline level is greater than 1 along that edge, copies need to be inserted to maintain the value across pipeline stages. The InsertCopyPass first calls the AnalyzeCopyPass, which goes through and sees which values need to be copied at each pipeline level. Multiple copies of the same value at the same pipeline level are combined into the same copy. Then, the InsertCopyPass creates a copy block and a copy instruction for each value that needs to be copied at each pipeline level, and replaces each instruction that uses that copy value as an operand with the newly created copy value.

- arrayNorm

  Location: lib/Transforms/VHDLOutputPass/ArrayAccessNormalization.cpp

  Purpose: Input arrays in ROCCC offsets based off of an index plus a negative number. However, the InputControllerPass which outputs the FSM for handling all of the input streams, input scalars, and load previous values expects streams to be normalized greater than zero. This pass looks at each input stream, and if the stream has any indices in the form [i-n] with i being the loop induction variable and n being a constant int, then the largest value n found is added to all of the indices that use that loop induction variable.

- vhdl

  Location: lib/Transforms/VHDLOutputPass/VHDLOutputPass.cpp

  Purpose: This pass encompasses four difference passes: LoopControllerPass, InputControllerPass, OutputControllerPass, and VHDLOutputPass. Each pass uses a common VHDL library to construct a representation of the final VHDL purely in memory and then outputs the VHDL after it has been checked for basic syntactical correctness. The loopController is a component that manages the iteration of the loop induction variables; this is used by the inputController and outputController to manage accesses to stream elements. The inputController takes all input scalars and input

streams, and makes sure that they are pushed onto the datapath simultaneously. The outputController serializes the output streams, as well as manages when the overall component completes processing.

- componentInfo

  Location: lib/Transforms/RocccCFGtoDFG/ComponentInformationPass.cpp

  Purpose: The database contains information on each system and module compiled. This pass writes that information to the database. Specifically, for modules it writes the delay of the component, the original struct name use, and the port ordering necessary to call the module in C. There is currently no system information written to the database in this pass, although area and frequency estimations will be written to the database here.

- printPortNames

  Location: lib/Transforms/VHDLOutputPass/PortNamePrinter.cpp

  Purpose: This pass writes the list of ports for each system and module to the database. Ports are organized as either stream or register ports. If they are register ports, the direction, size, original C name, and VHDL name are written to the database. For stream ports, the direction size, original C name of the stream, and VHDL name of each component of the stream are written to the database. Streams have the following components - any number of data channels, a 1-bit valid port, a 1-bit pop port, a 32-bit address port, and a 1-bit address ready port. For each component that belongs to the same stream, the C name will be the same name as the name of the original stream.

- deleteAll

  Location: lib/Transforms/RocccCFGtoDFG/RemoveExtends.cpp

  Purpose: All of the previous transformations leave the llvm tree in a state that is not internally consistant. This pass deletes everything, so as to leave the tree in an internally consistant state.

8 Database

The database used to store information used at all stages of the compiler is implemented using sqlite3 and contains four tables - ComponentInfo, ModulesUsed, Ports, and Version. All tables other than Version are updated whenever a component is compiled.

ComponentInfo contains information about every component that has been compiled with ROCCC. The columns of the ComponentInfo table are as follows:

- componentName : Contains the name of every generated component and should be unique.
- type : Contains either the string "MODULE" or the string "SYSTEM", or contains an intrinsic identifier. The intrinsic identifiers currently are: "INT_DIV", "INT_MOD", "FP_ADD", "FP_SUB", "FP_MUL", "FP_DIV", "FP_EQUAL", "FP_NOT_EQUAL", "FP_LESS_THAN", "FP_GREATER_THAN", "FP_LESS_THAN_EQUAL", "FP_GREATER_THAN_EQUAL", "FP_TO_INT", "FP_TO_FP", "INT_TO_FP", and "INT_TO_INT".
- portOrder : Contains a string that is used by the GUI to initialize an instantiation of a module. When the user double clicks on a module in the IPCore view, the module is inserted into the currently open file with the parameter list being populated with the portOrder string.
- delay : Contains the number of clock cycles required to generate data for a module. Used by the Lo-CIRRF passes to properly pipeline module calls.
- structName : Contains the name of the struct that was originally compiled when creating modules. Used when generating C level code from the GUI, such as PCore generation which will soon be supported.

Whenever a component is compiled, all rows in the ComponentInfo table with the componentName equal to the compiled component's name are dropped, and then the relevant information is inserted. This assures that there only ever exists one row with any given componentName.

The ModulesUsed table lists the dependencies between components. There are two columns - componentName and moduleName. Each row of the ModulesUsed table is one dependency. For example, as shown in Figure 2, BitWidthTest only instantiates int_div32, so there is one row with "BitWidthTest" as the componentName. On the other hand, MDFloat uses fp_sub, fp_mul, and fp_add, so there are three separate rows with "MDFloat" as the componentName.

In the ModulesUsed table, the "componentName" column contains a non-unique name of the dependent component. The "moduleName" column contains the non-unique name of the module that the dependent component depends on.

Whenever a component is compiled, all rows in the ModulesUsed table with the componentname of the compiled component's name are dropped, and then a row for each dependency is inserted. Thus it is possible to have multiple rows with the same componentName, but they are from a single compilation.

TABLE ModulesU: Search Show All Add Record Duplicate Record Edit Selected Delete Selected

rowid	componentName	moduleName
1	BitWidthTest	int_div32
3	MDFloat	fp_sub
4	MDFloat	fp_mul
5	MDFloat	fp_add

Figure 2: Modules Used Table

The Ports table lists all the ports of every component compiled. Each row of the Ports table is a single port. The columns of the Ports table are as follows:

- componentName : Contains the name of the component this port belongs to. This value is not unique.
- readableName : Contains the name of the original C value (if available) that this port is associated with. This value is not unique due to the multiple ports generated from the same C value for streams.
- type : Contains one of the following strings - "REGISTER" which is a single, registered value; "STREAM_CHANNEL", which is a data port associated with a stream; "STREAM_VALID", which is the valid port for a stream; "STREAM_POP", which is the pop port for a stream; "STREAM_ADDRESS", which is an address port for a stream; or "STREAM_ADDRESS_RDY", which is the address ready port for a stream. There can be multiple STREAM_CHANNEL ports for any given componentName and readableName; the rest of the types are unique to a given componentName and readableName.
- portNum : Contains an integer value that lists in what order the ports come in the component. This is used by the GUI to correctly generate components. In particular, this value is the only way to know the ordering of the STREAM_CHANNELS for a given stream.
- vhdlName : Contains the name of the port in the actual VHDL.
- direction : Contains either the string "IN" or "OUT" to denote the actual direction of the port.
- bitwidth : Contains the integer value that is the width of the port in bits.
- dataType : Contains the data type of the original C variable this port is associated with, if any. For example, a floating point input scalar port may have dataType = "float", while the data port of a two dimensional integer output stream may have dataType = "int**".

The Version table simply keeps track of the version of the database that is currently in use. This is used to determine when to update certain aspects of the GUI when performing automatic update.