User Manual for the SPARK Parallelizing High-Level Synthesis Framework Version 1.1

Sumit Gupta

Center for Embedded Computer Systems
University of California at San Diego and Irvine
sumitg@cecs.uci.edu
http://mesl.ucsd.edu/spark

April 14, 2004

Contents

Chapter 1

About this Manual

This is a user manual for the *Spark* parallelizing high-level synthesis software tool. In this manual, we document the various command-line flags and formats for hardware resource library file and the script file.

The manual is organized in the following manner: in the next chapter, we give an overview of the *Spark* methodology and framework. In Chapter 3, we present a quick start guide to get the user up and running immediately. For more details about the command-line flags and the way the scripts and hardware description files can be modified, the user is directed to Chapter 4. Appendices A and B list a sample hardware description file and a scheduling script file respectively.

1.1 Copyright

1.2 Disclaimer

- The *Spark* software program and the documentation is copyrighted by the Regents of the University of California. The following terms apply to all files associated with the software unless explicitly disclaimed in individual files.

- The software program and documentation are supplied ”as is”, without any accompanying services from The Regents. The Regents does not warrant that the operation of the program will be uninterrupted or error-free. The end-user understands that the program was developed for research purposes and is advised not to rely exclusively on the program for any reason.

- In no event shall the University of California be liable to any party for direct, indirect, special, incidental, or consequential damages, including lost profits, arising out of the use of this software and its documentation, even if the University of California has been advised of the possibility of such damage. The University of California specifically disclaims any warranties, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. the software provided hereunder is on an ”as is” basis, and the University of California has no obligations to provide maintenance, support, updates, enhancements, or modifications.

1.3 Reporting Bugs

The *Spark* distribution comes with no official bug fixing support or maintenance and we are not obliged to provide any updates or modifications. However, you may report bugs to:

spark@ics.uci.edu

Subject: BUG: Brief description of bug

1.4 Acknowledgments

The *Spark* framework was developed by Sumit Gupta with major contributions to the underlying framework by Nick Savoiu. Mehrdad Reshadi and Sunwoo Kim also contributed to the code base. Professor Rajesh Gupta, Nikil Dutt and Alex Nicolau led the *Spark* project. This project was funded by Semiconductor Research Corporation and Intel Incorporated.

1.5 Change Log

Date	Changes
1/12/04	Introduced documentation for the newly introduced Windows version of SPARK. Also, some additional notes in the VHDL section since output VHDL is now synthesizable by Xilinx XST
12/24/03	Clarified how ordering of resources in [Resources] section affects scheduling Added an example to clarify the difference between resource bound and unbound code
12/23/03	Added an example of how to make arrays one-dimensional in Appendix
11/11/03	Added clarifications and details to restrictions on input “C” code section. Added more examples in the Appendix that demonstrate how to modify code to make it synthesizable.
09/1/03	First release of document

Chapter 2

Introduction to the *Spark* High-Level Synthesis Framework

Spark is a *parallelizing high-level synthesis* framework that synthesizes a behavioral description using a set of aggressive compiler, parallelizing compiler and synthesis techniques [1, 2, 3]. An overview of the *Spark* framework is shown in Figure 2.1. *Spark* takes a behavioral description in ANSI-C as input albeit with no support for pointers, function recursion and gotos. The output of *Spark* is synthesizable register-transfer level (RTL) VHDL. As shown in Figure 2.1, *Spark* also takes as input additional information such as a hardware resource library, resource and timing constraints, data type information, and user controlled scripts that guide the various heuristics and transformations.

The code parallelization and transformation techniques in *Spark* have been grouped into pre-synthesis, scheduling, dynamic, and basic compiler optimizations – this grouping improves controllability over the transformations applied to the input description. The *pre-synthesis* transformations are applied at a source-level, whereas the scheduling and dynamic transformations are applied during scheduling. The basic compiler transformations are applied at each stage of synthesis – during pre-synthesis, scheduling and post-scheduling.

The pre-synthesis transformations include coarse-level transformations such as loop unrolling and compiler transformations such as common sub-expression elimination (CSE), copy propagation, dead code elimination, loop-invariant code motion and so on. These compiler transformations aim to remove unnecessary and redundant operations and reduce the number of operations within loops.

The pre-synthesis phase is followed by the *scheduling* phase (see Figure 2.1). Resource allocation and module selection are done by the designer and are given as input to the synthesis tool through a hardware resource library. The *transformations toolbox* in the scheduler contains the Percolation and Trailblazing code motion techniques, dynamic renaming of variables, and several compiler, parallelizing compiler and synthesis transformations. The synthesis transformations include chaining operations across conditional blocks, scheduling on multi-cycle operations and resource sharing.

The core parallelizing compiler transformations are a set of *speculative code motions* that not only move operations out of conditionals but sometimes duplicate operations into the branches of conditional blocks [4, 5].

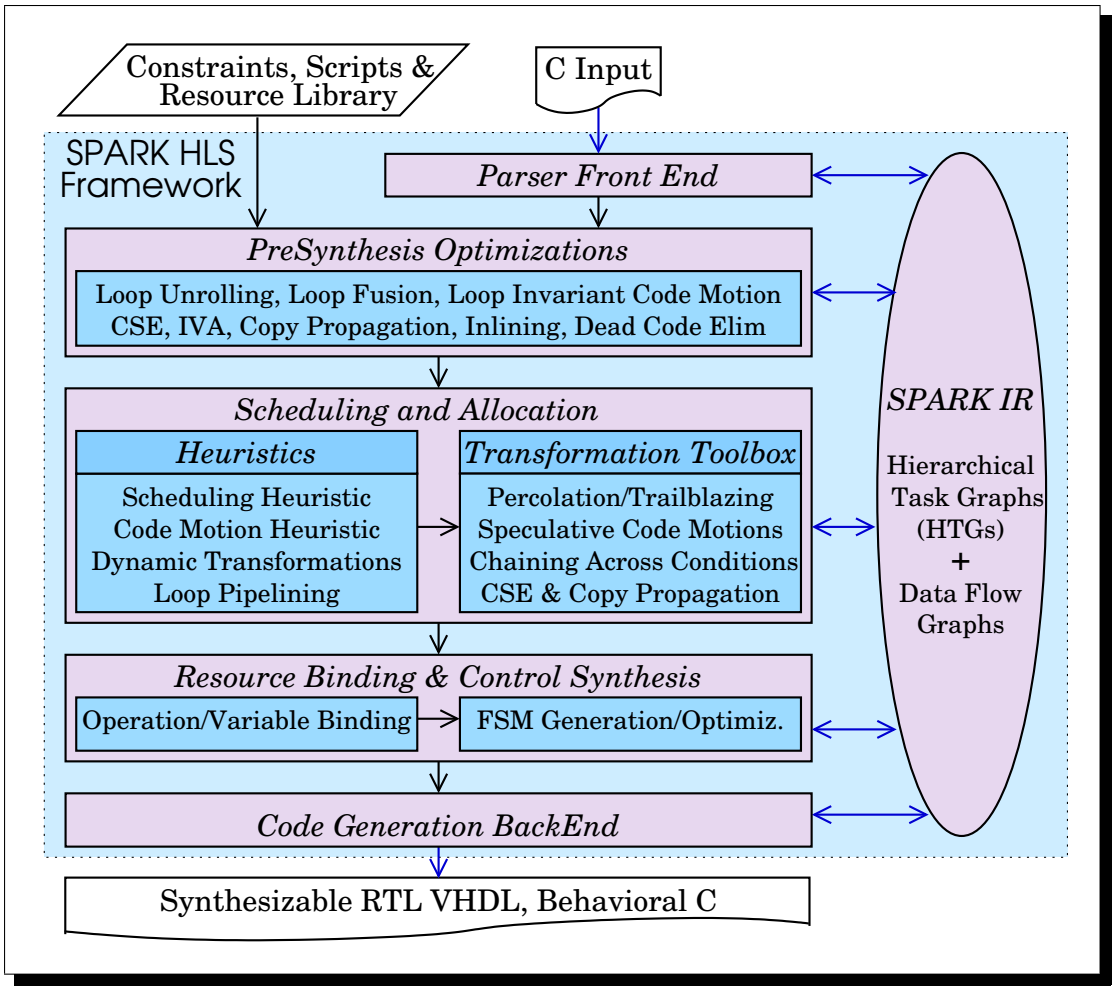

Figure 2.1. *An overview of our High-Level Synthesis Framework*

The scheduler also employs several compiler transformations applied *dynamically* during scheduling. These dynamic transformations, such as dynamic CSE, dynamic copy propagation et cetera exploit the new opportunities created by code motions [6]. A dynamic branch balancing technique dynamically adds scheduling steps in conditional branches to enable code motions – specifically those code motions that duplicate operations in conditional branches [7, 8].

The scheduling phase is followed by a *resource binding and control synthesis* phase. This phase binds operations to functional units, ties the functional units together (interconnect binding), allocates and binds storage (registers), generates the steering logic and generates the control circuits to implement the schedule. We employ an interconnect minimizing resource binding methodology by which operations are mapped to functional units and variables are mapped to registers in a way that minimizes the number of inputs to the multiplexers connected to the functional units [5, 9]. This reduces the complexity and size of the multiplexers and the associated control logic.

After resource binding, a control unit is generated using the finite state machine (FSM) controller style. We use

a *global slicing* approach to FSM generation, whereby operations in mutually exclusive control paths that execute in the same cycle are assigned the same state [1]. This reduces the number of states in the controller and also, gives more information to the logic synthesis tool about the control generation for the mutually exclusive operations.

Finally, a back-end code generation pass generates register-transfer level (RTL) VHDL. This RTL VHDL belongs to the subset of VHDL that is synthesizable by commercial logic synthesis tools. This enables our synthesis framework to complete the design flow path from architectural design to final design netlist. Note that, the output VHDL is structural with all operations bound to resources (components in VHDL) and variables bound to registers.

Spark also has back-end code generation passes that generate ANSI-C and behavioral VHDL. These behavioral output codes represent the scheduled and optimized design. The output "C" can be used in conjunction with the input "C" to perform functional verification and also, to improve visualization for the designer on the affects of the transformations applied by *Spark* on the design.

The synthesis framework is organized as a toolbox of transformations guided by heuristics that are fairly independent of the transformations. We instrumented our synthesis framework with a scripting ability that provides the designer with knobs to experiment and tune the heuristics and to identify the transformations that are useful in optimizing overall circuit quality.

2.1 Inferring Hardware Intent from C

As mentioned above, the *Spark* synthesis framework accepts input in ANSI-C, but does not support the synthesis of pointers (arrays are supported) and code with function recursion and irregular jumps (see next Chapter for details). The input "C" description is a sequential description of statements. Statements may be operation expressions, conditional constructs (if-then-else, switch-case) and loop constructs (for, while, do-while loops). Besides the operations supported by "C", we also support Boolean conditional checks; these Boolean checks are *produced* by comparison or relational tests such as $<$, $==$, $\geq$, $\neq$, et cetera. We decompose complex expressions into three-address expressions (of type a=b+c) [10], since hardware functional units are modeled in our framework as 2-input, 1-output resources. Each three-address expression is then called an *operation* in our representation.

Each function in the input description is mapped to a (concurrent) hardware block. If one function calls another function, then the called function is instantiated as a component in the calling function. We enable the designer to specify the range (bit-widths) of the various data types supported by C as a table in a hardware description file. This table has three columns: the data type, lower data range and upper data range. Hence, we can specify that an "integer" ranges from -32767 to 32768; this corresponds to a 16 bit boolean in hardware. Also, this same table can be used to make entries for specific variables from the input description. Hence, a designer can specify that a variable "myVariable" in the design description can have a range of only 0 to 15. This enables the designer to use his or her knowledge of the design to provide constraints.

The back-end code generator uses this table of data types to generate a structural hardware representation of the scheduled design in VHDL [11]. To be synthesizable, hardware descriptions require the exact range of data types

used in the design. Furthermore, high-level synthesis transformations can take advantage of the designer provided range constraints for specific variables to improve synthesis results [12].

The structure of the input description in terms of control and loop constructs are retained by our framework using a hierarchical intermediate representation (IR) [2]. This hierarchical IR consists of basic blocks[1] encapsulated in *Hierarchical Task Graphs* (HTGs) [13, 14]. HTG nodes can be of three types: *single* or non-hierarchical nodes, *compound* or nodes that have sub-nodes), and *loop* nodes that encapsulate loops. Hence, basic blocks are encapsulated into compound HTG nodes to form hierarchical structures such as if-then-else blocks, switch-case blocks, loop nodes or a series of HTG nodes.

In addition to HTGs, we also maintain control flow graphs (CFGs) that capture the flow of control between basic blocks and data flow graphs (DFGs) that capture the data dependencies between operations. Whereas HTGs are useful for hierarchical code motions and applying coarse-grain transformations, CFGs are used for design traversal during scheduling.

There is currently no support for providing specific timing information to *Spark*. Hence, *Spark* assumes that all input variables/signals are available at the start of execution of a hardware block and stay available for the duration of its execution.

2.2 Restrictions on "C" Accepted as Input

Spark uses the EDG [15] C/C++ front-end. *Spark* takes pure ANSI-C with no special constructs. Although the front-end is a complete ANSI-C parser, however, there are some features of the "C" language that *Spark* does not currently support because of fundamental problems or unresolved issues in synthesizability. These are:

- ❒ No support for pointers. However, arrays and array accesses of the type arr[index variable expression] are supported. Also, passing arguments by reference to a function is also supported.
- ❒ No support for function recursion.
- ❒ No support for irregular jumps through *goto*. Some of these can be resolved in a state machine, but they adversely affect our ability to apply transformations.

Besides these, the following features of *C* are not supported because they have not yet been implemented in the *Spark* framework:

- ❒ No support for break and continue. In general, it is possible to convert a program with breaks and continues into a program without them [14].
- ❒ No support for multi-dimension arrays. Multi-dimensional arrays can be reduced manually to single-dimensional arrays. For example: consider an array a[N][M]. This can be re-declared as a[N*M]. Any access to a[i][j] then becomes a[i*M+j].

[1] A basic block is a sequence of statements in the input code with no control flow between them.

- ❒ Poor/no support for structs and unions. In general, structs are *currently* not synthesizable. Also, no VHDL generation for user-defined data types.

- ❒ Poor support for expressions of type (a ? b : c). We advise changing this expression to the following statement:
  if (a) b
  else c

We discuss code modifications that can serve as workarounds for unimplemented features in Appendix C.

Chapter 3

Quick Start Guide

In this chapter, we explain how to download, install and start using *Spark*. We also present the files and directory setup required to run *Spark*.

3.1 Downloading and Installing *Spark*

Spark can be obtained from the download page at:

http://mesl.ucsd.edu/spark

Choose the appropriate distribution based on the operating system you want to work on. The distribution will be named "spark-[OS]-[Version].tar.gz". So let us say you are interested the 1.1 version for the Linux platform, you will download the file "spark-linux-1.1.tar.gz". For the Windows distribution, we ship a ZIP archive named "spark-win32-[Version].zip".

After downloading this file, gunzip and untar the file as follows:

```
gunzip spark-linux-1.1.tar.gz
tar xvf spark-linux-1.1.tar.gz
OR
unzip spark-win32-1.1.zip
```

Uncompressing (gunzip and untar) the distribution will create the following directory structure:

```
spark-[OS]-[Version]/
                     bin/
                     include/
                     tutorial/
                     spark-setup.csh
                     spark-setup.sh
```

where [OS] is the operating system for which you downloaded the distribution and [Version] is the *Spark* version. The "bin" directory contains the *Spark* binary and other files required to execute *Spark*. The "tutorial" directory contains the tutorial shipped with this distribution and described in the *Spark Tutorial* document. The

"include" directory contains standard "C" include files such as stdio.h et cetera that are included by some input applications. However, note that for the Windows distribution, these include files are not useful. If you do want to include system files such as "stdio.h" et cetera, please specify the path to these files using the environment variable SPARK_INCLUDES or using the command-line flag "-I /path/to/include".
To begin with source the "spark-setup.csh/sh" script as follows:

```
source spark-setup.csh   # if you are using csh/tcsh shell
. spark-setup.sh         # if you are using sh/bash shell
```

For the Windows distribution, if you are using *Spark* under CYGWIN or MSYS/MINGW, then you can source the spark-setup.sh file, else for native Windows, you can run the batch file "spark-setup.bat".

The "spark-setup" script sets up the path to the *Spark* executable, along with some environment variables required by *Spark*. You are now ready to use *Spark*. In the next section, we describe the files and directory setup required for *Spark* to run. Details on the how to run the tutorial can be found in a separate document also available on the *Spark* download website.

3.2 Files Required and Directory Setup

To run the *Spark* executable, you require:

- *default.spark* or *filename.spark*: This file is the hardware description file that contains the resource allocation, bit-widths of various data types, et cetera (see Section 4.3). A sample default.spark is listed in Appendix A.
- *Priority.rules* or any file specified under the "[SchedulerRules]" section of the default.spark file. This file contains all the rules and switches controlling the scheduling heuristic, branch balancing heuristic, code motions et cetera (see Section 4.5). A sample Priority.rules is listed in Appendix B.
- *./output* directory: This is the directory in which all the output files, including the *dotty* graphs, VHDL and C output, et cetera are generated.

Sample default.spark and Priority.rules files are included with the *Spark* distribution in the "spark-[OS]-[Version]/bin" directory.

3.3 Recommended Command-line Options for Invoking *Spark*

We recommend the following command-line options for invoking *Spark*:

spark -hli -hcs -hcp -hdc -hs -hvf -hb -hec filename.c

The command-line options that are enabled are: loop-invariant code motion (-hli), common sub-expression elimination (-hcs), copy and constant propagation (-hcp), dead code elimination (-hdc), scheduling (-hs), generation of synthesizable RTL VHDL (-hvf), interconnect-minimizing resource binding (-hb) and generation of statistics about cycle count (-hec).

3.4 Options for Synthesizing Microprocessor Blocks

To synthesize microprocessor blocks [16], we have to enable operation chaining across conditional boundaries by using the command-line option *−hch* and we have to increase the clock period to a large number so that all the operations can be packed into one clock cycle. We arbitrarily increase clock period to 10000ns: this enables up to 1000 additions to be chained together (if each addition takes 10ns). The clock period can be set in the "[GeneralInfo]" section of the default.spark file (see Section 4.3.1) and the timing of each operation in the design can be set in the "[Resources]" section (see Section 4.3.3).

Also, we have to enable full loop unrolling. This can be done by setting the number of unrolls in the "[RDLP-Params]" section of the default.spark file to the number of iterations of the loop to be unrolled (see Section 4.3.5).

Hence, for synthesizing microprocessor blocks, we recommend the following command-line options for invoking *Spark*:

spark -hch -hli -hcs -hcp -hdc -hs -hvf -hb -hec filename.c

Chapter 4

Detailed Instructions

4.1 Command Line Interface

Spark can be invoked on the command-line by the command:

spark [command-line flags] filename.c

filename.c is the name of the input file with the behavioral code to be synthesized. We can also specify multiple files on the command-line.

Some of the important command-line flags are:

Flag	Purpose
-h	Prints help
-hs	Schedule Design
-hvf	Generate RTL VHDL (output file is ./output/$filename\_spark\_rtl$.vhd)
-hb	Do all Resource Binding (operation to functional unit and variable to registers)
-hec	Some statistics about states, path lengths are printed into the backend VHDL file
-hcc	Generate output C file of scheduled design (output file is ./output/$filename\_sparkout.c$)
-hch	Chain operations across Conditional Boundaries
-hcp	Do Copy and Constant Propagation
-hdc	Do Dead Code Elimination
-hcs	Do Common Sub-Expression Elimination
-hli	Do Loop Invariant Code Motion
-hg	Generate graphs (output files are in directory ./output).
	Graphs are generated by default if scheduling is done, as: $filename\_sched$.dotty
-hcg	Generate function call graph (./output/$CG\_filename$.dotty)
-hq	Run quietly; no debug messages

Spark writes out several files such as the output graphs (see next section) and the backend VHDL and C files. All these files are written out to the subdirectory “output” of the directory from which *Spark* is invoked. This

directory has to be created before executing *Spark*.

Some useful EDG command-line flags are given below. These flags are useful when the style of C does not completely conform to ANSI-C.

Flag	Purpose
-m	For old style C (see below)
-c99	For C conforming to the C99 specification

For example, use the "-m" command-line flag for input code in which functions are specified in the following format:

```
void myfunction(variableA, variableB)
     int variableA;
     int variableB;
{
..
}
```

4.2 Viewing Output Graphs

The format that *Spark* uses for the output graphs is that of AT&Ts *Graphviz* tool [17]. Output graphs are created for each function in the "C" input file. The output graphs generated are listed in Table 4.1. The key to the nomenclature used in naming the graphs is given in Table 4.2

Graphs representing the original input file	*Graphs representing the scheduled design*
$CFG\_filename\_c\_functionName.dotty$	$CFG\_filename\_c\_functionName\_sched.dotty$
$HTG\_filename\_c\_functionName.dotty$	$HTG\_filename\_c\_functionName\_sched.dotty$
$DFG\_filename\_c\_functionName.dotty$	$DFG\_filename\_c\_functionName\_sched.dotty$
$CDFG\_filename\_c\_functionName.dotty$	$CDFG\_filename\_c\_functionName\_sched.dotty$
$CG\_filename\_c\_functionName.dotty$	-
$CG\_DFG\_filename\_c\_functionName.dotty$	-

Table 4.1. *Output Dotty graphs generated by* Spark

To view these output graphs, we use the *Graphviz* command line tool *dotty*, as follows:

$dotty\ output/graphfilename.dotty$

4.3 Hardware Description File Format: default.spark

Spark requires as input a hardware description file that has information on timing, range of the various data types, and the list of resources allocated to schedule the design (resource library). This file has to be named "filename.spark", where

Abbreviation	Explanation
CFG	Control Flow Graph: Basic blocks with control flow between them
HTG	Hierarchical Task Graph: Hierarchical structure of basic blocks
DFG	Data Flow Graph: data flow between operations
CDFG	Control-Data Flow Graph: Resource-utilization based view of the CFG
CG	Call Graph showing control flow between functions
CG_DFG	Call Graph with data dependencies and control flow between functions
filename	The name of "C" input file
functionName	The name of the function in input file

Table 4.2. *Key to abbreviations used in Dotty graph naming*

filename.c is name of the "C" input file. If a filename.spark does not exist, then the *Spark* executable looks for "default.spark". One of these files *has to exist* for *Spark* to execute.

The various have sections in the .spark files are described in the next few sections. Note that, comments can be included in the .spark files by preceding them with "//".

4.3.1 Timing Information

The timing section of the .spark file has the following format:

```
// ClockPeriod  NumOfCycles  TimeConstrained   Pipelined
[GeneralInfo]
     10              0            0                 0
```

Of these parameters, only the clock period is used by the scheduler to schedule the design. The rest of the parameters have been included for future development and are not used currently. They correspond to the number of cycles to schedule the design in (timing constraint), whether the design should be scheduled by a time constrained scheduling heuristic, and whether the design should be pipelined.

4.3.2 Data Type Information

Each data type used in the "C" input file has to have an entry in the "[TypeInfo]" section of the .spark file, as shown below:

```
//typeName           lowerRange      upperRange
//or variableName    lowerRange      upperRange
[TypeInfo]
      int              -32767          32768
      myVar              0               16
```

This section specifies the range of the various data types that can be specified in a C description (such as int, char, float, unsigned int, signed int, long, double et cetera). The format is data type, lower bound range, and upper bound range. Also, the data value range of specific variables from the input C description can be specified in this section, as variable name, lower range, upper range. This is shown in the table above my the example of a variable "myVar" whose range is from 0 to 16.

4.3.3 Hardware Resource Information

This section parameterizes each resource allocated to schedule the design as shown by the example below:

```
//name  type   inpsType  inputs  outputs number  cost  cycles  ns
[Resources]
CMP    ==,<,!=    i        2       1       1      10     1     10
```

The example given in the first line above is:

- ❒ A resource called CMP (comparator).
- ❒ The operations $==, <, != $ can be mapped to this resource.
- ❒ It handles inputs of type integer (i).
- ❒ It has 2 inputs.
- ❒ It has 1 output.
- ❒ There is one CMP allocated to schedule the design.
- ❒ Its cost is 10. The cost, although not used currently, can be integrated into module selection heuristics while selecting a resource for scheduling from among multiple resources.
- ❒ The CMP resource executes in 1 cycle.
- ❒ It takes 10 nanoseconds to execute.

We can define resources of every type supported by the "C" language in the [Resources] section of the hardware description file. A detailed example is given in Appendix A. Note that although we allow specifying multi-cycle resources in the resource description section, we do not currently support structurally pipelined resources.

A list of all basic operators/resources recognized by *Spark* is given in Table 4.1. The columns in this table list the resource type as specified in the .spark file, the corresponding operator in C, the operation name, and the number of inputs this resource accepts (number of outputs is currently one for all the resources). Complex resources can be made using these basic operators as shown in the example above for the "CMP" resource.

If the *Spark* executable reports an error that says "All ss in StatementNum are not scheduled", then look at the operators in the expressions reported after this error and add them to the hardware description file (.spark file).

Note that: The order of resources in the [Resources] section is the order in which operations from the input code will be mapped to the resources. So, if there are two resources that do additions:

```
//name  type   inpsType  inputs  outputs number  cost  cycles  ns
[Resources]
ALU      +,-      i        2       1       1      10     1     10
ADD       +       i        2       1       1      10     1     10
```

Then, the scheduler will first try to schedule addition operations to the *ALU* resource and then try to schedule to the *ADD* resource.

Resource Type	Operator in C	Operation	Number of Inputs
+	+	add	2
-	-	subtract	2
*	*	multiply	2
/	/	divide	2
!=	!=	compare	2
==	==	compare	2
>	>	compare	2
<	<	compare	2
>=	>=	compare	2
<=	<=	compare	2
<<	<<	shift	2
>>	>>	shift	2
[]	[]	array access	1
~	~	complement	1
!	!	logical not	1
&&	&&	logical and	2
||	||	logical or	2
b&	&	Boolean and	2
b|	|	Boolean or	2
call	-	function call	-

Figure 4.1. **List of all resource types, corresponding C operators, operation name, and number of inputs**

4.3.4 Specifying Function Calls as Resources in the Hardware Description File

A hardware resource has to be specified for each function call in the code. Hence, for a function call with the name "myfunction", we have to declare a hardware resource as follows:

```
//name     type   inpsType  inputs  outputs number cost cycles ns
[Resources]
myfunction call     i          0       0       2    10    1    10
```

This line specifies a function call with name "myfunction" with integer input types that takes 1 cycle and 10 ns to execute. The number of inputs and outputs is determined by *Spark* from the declaration of the function in the input code. If the function "myfunction" is defined in the input code, *Spark* performs an analysis of the function and determines the actual execution cycles and time of the function and uses this information for scheduling the calling function.

To generate correct and synthesizable VHDL, you have to specify a resource specifically for each function call. The number of components instantiated for function calls to the same function is determined by the number of resources specified above. Hence, in the example for "myfunction", we specified two resources; thus, there will be two component instantiations in the VHDL code.

The keyword "ALLCALLS" is used to match all function calls in the input code as follows:

```
//name     type   inpsType  inputs  outputs number cost cycles ns
[Resources]
ALLCALLS   call     i          0       0       1    10    1    10
```

This may be useful to capture calls to functions such as *printf* that may have been used in the *C* code. **Note that:** the ALLCALLS resource should be put **at the end** of the [Resources] section, otherwise all the function calls will be mapped to this resource.

4.3.5 Loop Unrolling and Pipelining Parameters

```
// variable maxNumUnrolls maxNumShifts  percentageThreshold ThruputCycles
[RDLPParams]
* 0 0 70 0
i 0 2 70 0
```

This section presents the parameters for loop unrolling and loop pipelining.

- ❒ Variable is the loop index variable to operate on ("*" means all loops)
- ❒ The second parameter specifies the number of times to unroll the loop.
- ❒ Number of times the loop should be shifted by the loop pipelining heuristic.
- ❒ Percentage threshold and throughput cycles are parameters used by the resource-directed loop pipelining (RDLP) heuristic implemented in our system.

The example in the second line of the "[RDLPParams]" section shown above says that the loop with loop index variable "i" should be shifted twice. To fully unroll a loop, specify number of loop unrolls to be equal to or more than the maximum number of iterations of the loop.

4.3.6 Other Sections in .spark files

The other sections in the .spark files are:

- ❒ [RDLPMetrics]: Controls the various parameters of the resource-directed loop pipelining (RDLP) heuristic.
- ❒ [SchedulerRules]: The file that *Spark* should read to get the scheduling scripts (rules and parameters). Default is: Priority.rules.
- ❒ [SchedulerScript]: The scheduling script to use: different scheduling heuristics can be employed by changing this entry. Default is "genericSchedulerScript".
- ❒ [Verification]: Specifies the number of test vectors that should be generated for functional verification of output C with input C.
- ❒ [OutputVHDLRules]: Specifies the VHDL generation rules; this is covered in more detail in Section 4.4.

4.4 VHDL Output Generated by *Spark*

Spark generates synthesizable register-transfer level VHDL code (by specifying the -hvf command-line flag). If the -hb command-line flag is also given, then the VHDL code is generated after operation and variable binding [5]. In resource-bound code, an entity-architecture pair is generated for each resource in the hardware description file (.spark file) and a component is instantiated for each instance of the resource (as specified by the number of resources in the .spark file). Processes are

created to generate multiplexers at the input of each functional unit/resource and variables are bound to registers that are explicitly declared.

In contrast, when the resource binding flag (-hb) is not specified, then only operation expressions are generated in the VHDL code. This code looks more like: $a <= b + c$ and so on. Hence, the unbound code is easier to read and understand and also, has clearer relation with input *C* code since variables from the input code are used in the VHDL. However, from a logic synthesis point of view, unbound VHDL code allows the logic synthesis tool to decide the number of resources and registers that are allocated to the synthesize the final netlist.

The following is an example of the same VHDL code in the data path process for the bound and unbound case respectively.

```
if CURRENT_STATE(0) = '1' then
  regNum0 <= res_ALU_1_out;
  regNum5 <= 0;
elsif CURRENT_STATE(1) = '1' then
  regNum1 <= res_ALU_0_out;
  regNum2 <= res_ALU_1_out;
  hT0 <= res_CMP_2_out;
```

(a)

```
if CURRENT_STATE(0) = '1' then
  hT5  <=  (x + 2);
  col  <=  0;
elsif CURRENT_STATE(1) = '1' then
  hT14  <=  (col + bytes);
  hT15  <=  (col - x);
  hT0  <=  (col < 10);
```

(b)

Figure 4.2. (a) Bound VHDL code (b) Corresponding unbound VHDL code

Note that, from release version 1.1 on, the output VHDL file generated by *Spark* has the extension ".vhd" versus the earlier ".vhdl". This is for compatibility with Windows based tools such as Xilinx XST that look for .vhd VHDL files by default.

4.4.1 Generating VHDL bound to Synopsys DesignWare Foundation Libraries

The default.spark (or filename.spark) file contains a section for controlling the type of VHDL output generated by *Spark*. This is shown below:

```
[OutputVHDLRules]
PrintSynopsysVHDL=true
```

By setting the "PrintSynopsysVHDL" to *true* in the "[OutputVHDLRules]" section of the default.spark (or filename.spark) file, we can generate VHDL that is Synopsys specific. This means that the VHDL code generated by *Spark* is synthesizable by Synopsys logic synthesis tools (Design Compiler). Hence, the VHDL code uses Synopsys libraries and components from the Synopsys DesignWare Foundation library (specifically for the multiplier and divider).

The VHDL code also generates a SPARK package and stores this package in a SPARK library and this library is then used in the code. Hence, this SPARK library has to be mapped to your work directory. For Synopsys tools, this is done using the *.synopsys_vss.setup* file.

4.4.2 Generating VHDL Synthesizable by Other Logic Synthesis tools

If you are using logic synthesis tools from another vendor (besides Synopsys), then set the "PrintSynopsysVHDL" to *false* in the "[OutputVHDLRules]" section of the default.spark (or filename.spark). This can be done as follows:

```
[OutputVHDLRules]
PrintSynopsysVHDL=false
```

Also, the VHDL uses the SPARK package (use work.spark_pkg.all;) that is stored in the SPARK library. Thus, you have to edit your setup files to map the SPARK library to the work directory. Additionally, You will have to explicitly instantiate multi-cycle components such as the multiplier and divider from the standard cell library of your technology vendor. This has to be done in the architecture description of *res_MUL* and *res_DIV* in the VHDL code.

From release version 1.1 on, when the "PrintSynopsysVHDL" is false, *Spark* resets the conditional variables in the data path process (DP: Process) rather than the SYNC process. This is for ensuring synthesizability by Xilinx XST.

4.5 Scripting Options for Controlling Transformations and Heuristics: Priority.rules

Spark allows the designer to control the transformations applied to the design description by way of synthesis scripts. In this section, we discuss the scripting options available to the designer.

The scripting options can be specified in the file given by the "[SchedulerRules]" section of the .spark file (see previous section). The default script file *Spark* looks for is "Priority.Rules". This file has three main sections: the scheduler functions, the list of allowed code motions (code motion rules) and the cost of code motions. We discuss each of these in the next three sections. Note that in this file, "//" denotes that the rest of the line is a comment.

4.5.1 Scheduler Functions

An example of the scheduler functions section from a sample *Priority.rules* file is given in Figure 4.3. An entry in this section is of type "FunctionType=FunctionName". We have given explanations for each function type in comments on each line in this figure.

Of these we use the following flags for the experiments presented in this thesis: *DynamicCSE*, *PriorityType*, *BranchBalancingDuringCMs* and *BranchBalancingDuringTraversal*.

4.5.2 List of Allowed Code Motion

This section of the "Priority.rules" file has the list of code motions that can be employed by the scheduler. Each code motion can be enabled or disabled by setting the flag corresponding to it to "true" or "false". An example of the list of allowed code motions sections is as given below.

```
// all the following can take values true or false
[CodeMotionRules]
RenamingAllowed=true            // Variable Renaming allowed or not
AcrossHTGCodeMotionAllowed=true // Across HTG code motion allowed or not
SpeculationAllowed=true         // Speculation allowed allowed or not
ReverseSpeculationAllowed=true  // Reverse Speculation allowed or not
EarlyCondExecAllowed=true       // Early Condition Execution allowed or not
ConditionalSpeculationAllowed=true // Condition Speculation allowed or not
```

```
//line format: functiontype=functionvalue
[SchedulerFunctions]
CandidateValidatorFunction=candidateValidatorPriority   // Candidate Validation Algorithm
CandidateMoverFunction=TbzMover                         // Use Trailblazing for code motions
CandidateRegionWalkerFunction=topDownGlobal             // Design traversal for candidate operations
ScheduleRegionWalkerFunction=topDownBasicBlock          // Design traversal for scheduling
PreSchedulingFunction=initPriorities                    // Calculate priorities of operations before scheduling
PriorityType=max                                        // Calculate priority as max or sum of dependent
                                                           operations data dependencies
PostSchedulingStepFunction=postSchedulingPriority       // Reverse speculate all unscheduled operations at each
                                                           time step of scheduling
PreSchedulingStepFunction=preSchedPriority              // Do early condition execution before each time step
                                                           of scheduling
LoopSchedulingFunction=RDLP                             // Loop unrolling is done by RDLP
PreLoopSchedulingFunction=prepareForRDLP                // Initialization functions for loops
PostLoopSchedulingFunction=constantPropagation          // Optional post loop scheduling pass
ReDoHTGsForDupUp=false                                  // Whether to reschedule HTGs for possible
                                                           duplication-up true or false
ReassignPriorityForCS=true                              // Reassign priorities to favor operations
                                                           within basic blocks
RestrictDupUpType=targetBBUnsched                       // Restrict duplication-up
DynamicCSE=true                                         // Whether Dynamic CSE is enabled or not
BranchBalancingDuringCMs=true                           // Enable branch balancing during code motions
BranchBalancingDuringTraversal=true                     // Enable branch balancing during design traversal
```

Figure 4.3. **The scheduler functions section in a sample Priority.rules file.**

4.5.3 Cost of Code Motions

This section was developed to experiment with incorporating costs of code motions into the cost function based on which the operation to schedule is chosen. An example of this section is given below. Here all the code motions are assigned a cost of 1.

```
[CodeMotionCosts]
WithinBB                1
AcrossHTGCodeMotion     1
Speculation             1
DuplicationUp           1
```

The total cost of scheduling an operation is determined as:

*Total Cost = - Basic Cost of operation * Cost Of Each Code Motion required to schedule the operation*

where basic cost of the operation is the priority of the operation [2] and cost of each code motion is as given in the "[CodeMotionCosts]" section of the Priority.rules file. Since this function generates a negative total cost, the operation with the lowest cost is chosen as the operation to be scheduled.

Appendix A

Sample default.spark Hardware Description file

```
//NOTE: do no put any comments within a section
// Currently NumOfCycles, TimeConstrained and Pipelined are not used
// ClockPeriod  NumOfCycles  TimeConstrained   Pipelined
[GeneralInfo]
10      1       1       0

//typeName           lowerRange      upperRange
//or variableName    lowerRange      upperRange
[TypeInfo]
char                    0               8
signed_char             0               8
unsigned_char           0               8
short                   0               16
int                  -32767            32768
unsigned_short          0               16
unsigned_int            0               32
long                    0               64
unsigned_long           0               64
long_long               0               128
unsigned_long_long      0               128
float                   0               32
double                  0               64
long_double             0               128
myVariableFromInput     0               4

// all cycles in resources have to be ns/ClockPeriod = cycles
//name  type  inpsType inputs outputs number  cost    cycles  ns
```

```
[Resources]
ALU     +,-     i       2       1       1       10      1       10
MUL     *       i       2       1       1       20      2       20
CMP     ==,<    i       2       1       1       10      1       10
SHFT    <<      i       2       1       2       10      1       10
ARR     []      i       1       1       5       10      1       10
LOGIC   &&,||   i       2       1       5       10      0       0
GATE    b&,b|   i       2       1       5       10      0       0
UNARY   ~,!     i       1       1       5       10      0       0
ALLCALLS call   i       0       0       2       10      1       10

// variable  maxNumUnrolls  maxNumShifts  percentageThreshold cycleThruput
[RDLPParams]
* 0 0 70 0

//unroll, shift, resetUnroll, and resetShift metrics
[RDLPMetrics]
UnrollMetric=RDLPGenericUnrollMetric
ShiftMetric=RDLPGenericShiftMetric
ResetUnrollMetric=RDLPGenericResetUnrollMetric
ResetShiftMetric=RDLPGenericResetShiftMetric

//lists file that has scheduler rules/functions
[SchedulerRules]
Priority.rules

//function that drives scheduler
[SchedulerScript]
genericSchedulerScript

// numOfTestVectors
[Verification]
20

[OutputVHDLRules]
PrintSynopsysVHDL=true
```

Appendix B

Recommended Priority.rules Synthesis Script file

We found the following choice of options in the synthesis script produces the best synthesis results for data-intensive designs with complex control flow.

```
//line format: <functiontype>=<functionvalue>
[SchedulerFunctions]
ScheduleRegionWalkerFunction=topDownBasicBlockNoEmpty
CandidateValidatorFunction=candidateValidatorPriority
CandidateMoverFunction=TbzMover
LoopSchedulingFunction=RDLP
CandidateRegionWalkerFunction=topDownGlobal
PreSchedulingStepFunction=preSchedulingPriority
PostSchedulingStepFunction=postSchedulingPriority
PreLoopSchedulingFunction=prepareForRDLP
PostLoopSchedulingFunction=constantPropagation
PreSchedulingFunction=initPriorities
ReDoHTGsForDupUp=false  // true or false - false is better
ReassignPriorityForCS=true      // true or false - true is better
PriorityType=max  // max, sum, maxNoCond - max is best
RestrictDupUpType=targetBBUnsched // none, afterSchedOnce, targetBBUnsched
BranchBalancingDuringCMs=true // true or false - true is better
BranchBalancingDuringTraversal=true   // true or false - true is better
DynamicCSE=true

[CodeMotionRules]
RenamingAllowed=true
AcrossHTGCodeMotionAllowed=true
SpeculationAllowed=true
ReverseSpeculationAllowed=true
```

```
EarlyCondExecAllowed=true
ConditionalSpeculationAllowed=true

// the higher the cost, the more profitable a code motion is
// total cost = basicCost * CostOfEachCodeMotion
[CodeMotionCosts]
WithinBB                1
AcrossHTGCodeMotion     1
Speculation             1
DuplicationUp           1
```

Appendix C

Modifying Input Code to be Synthesizable by *Spark*

C.1 Input Code with Structs

Consider the input source code shown in Figure C.1(a) and consider that we want to synthesize the function “structEx”. Since *Spark* does not currently support structures, we can split this function into two and rewrite the code as shown in Figure C.1(b). Now we can put the function “structEx” in a separate file and execute *Spark* on that file.

```
struct myStruct {
int fir;
int sec;
};

void structEx(myStruct a, int *b)
{
  *b = a->fir + a->sec;
}
```

```
struct myStruct {
int fir;
int sec;
};
void callStructCode(myStruct a, int *b)
{
  structEx(a->fir, a->sec, b);
}

void structEx(int A_fir, int A_sec, int *b)
{
  *b = A_fir + A_sec;
}
```

Figure C.1. **(a) Input code that uses a structure (*myStruct*). (b) The function can be split into a calling function and a called function. The calling function calls the called function after enumerating the elements of the structure that are processed by the called function. The called function (*structEx*) is now synthesizable.**

C.2 Input Code with Pointers

```
/* arr is an array that has been
declared outside this function */
void pointerCode(int *arr, int *b)
{
  int *ptr;
  int i;

  ptr = arr;
  for (i = 0; i < 10; i++)
   {
     *b += *ptr;
     ptr++;
   }
}
```

```
/* arr is an array that has been
declared outside this function */
void pointerCode(int *arr, int *b)
{
  int i;
  int arr_index;

  arr_index = 0;
  for (i = 0; i < 10; i++)
   {
     *b += arr[arr_index];
     arr_index++;
   }
}
```

Figure C.2. **(a) Input code that uses a pointer to address an array. (b) The pointer can be replaced by a array index variable (*arr_index*) that determines which array element to access.**

Consider the input source code shown in Figure C.2(a) and consider that we want to synthesize the function “pointerCode”. In this case, it is straight forward to replace the pointer *ptr* with the array variable *arr*. The code can, hence, be rewritten as shown in Figure C.2(b). We have replaced the pointer with an array index variable instead. The code is now synthesizable by *Spark*.

C.3 Input Code with Breaks

```
void breakCode(int a, int *b)
{
  int i;

  for (i = 0; i < N; i++)
   {
     *b += a;
     if (*b > 100)
       break;
   }
}
```

```
void breakCode(int a, int *b)
{
  int i;
  int breakFlag;

  breakFlag = 0;
  for (i = 0; i < N; i++)
   {
     if (breakFlag == 0)
     {
       *b += a;
       if (*b > 100)
         breakFlag = 1;
     }
   }
}
```

Figure C.3. **(a) Input code with a *break* inside a for-loop. (b) The *break* can be replaced by a flag (*breakFlag*) that determines if future iterations of the loop body are executed or not.**

Consider the input source code shown in Figure C.3(a). Consider that we want to synthesize the function "breakCode". We can remove the *break* in this code by creating a flag that is checked for each iteration of the loop as shown in Figure C.3(b). If the *breakFlag* is set, then the statements in the loop body are not executed.

C.4 Input Code with Continues

```
void continueCode(int a, int *b)
{
  int i;

  for (i = 0; i < N; i++)
   {
     *b += a;
     if (*b < 100)
       continue;
     b = b/2;
   }
}
```

```
void continueCode(int a, int *b)
{
  int i;
  int continueFlag;

  for (i = 0; i < N; i++)
   {
     continueFlag = 0;
     *b += a;
     if (*b < 100)
       continueFlag = 1;
     if (continueFlag == 0)
      {
        b = b/2;
      }
   }
}
```

Figure C.4. **(a) Input code with a *continue* inside a for-loop. (b) The *continue* can be replaced by a flag (*continueFlag*) that determines if the rest of the code in the loop is executed or not.**

Consider the input source code shown in Figure C.4(a). Consider that we want to synthesize the function "continueCode". We can remove the *continue* in this code by creating a flag that is checked for each iteration of the loop as shown in Figure C.4(b).

In the modified code, when b is less than 100, then the *continueFlag* is set and hence, the $b = b/2$ code does not execute. The *continueFlag* has to be reset to zero at the beginning of each loop iteration to retain the semantics of the original code.

C.5 Input Code in which an Argument is Modified

```
void argumentCode(int a, int *b)
{
  int i;

  for (i = 0; i < N; i++)
   {
     *b += a;
     a = a/2;
   }
}
```

```
void argumentCode(int a, int *b)
{
  int i;
  int temp;

  temp = a;
  for (i = 0; i < N; i++)
   {
     *b += a;
     temp = temp/2;
   }
}
```

Figure C.5. **(a) "C" code with input variable *a* being modified in the function body. (b) Variable *a* is stored in a local variable *temp* and all instances of *a* are replaced by *temp*. This is done because VHDL semantics do not allow an input variable to be modified in the body of the code.**

The "C" language allows a function to modify a variable that has been passed as an argument to the function. For example, consider the code in Figure C.5(a). The modifications to the variable *a* are local to the function *argumentCode* since variable *a* has not been passed by reference.

When *Spark* generates VHDL code from this input C code, variable *a* will be declared as *input* in the VHDL code and there will be a statement updating *a* in the process body. However, VHDL semantics do not allow an input variable to be updated in the architecture body. Hence, to make this input code synthesizable, we have to modify the C code to store variable *a* in a temporary variable and update the temporary variable. The modified code is shown in Figure C.5(b).

C.6 Input Code with “?” if-then-else

Consider this code with an if-then-else written in a question format:

```
d = (a > 10) ? (a + b) : (a + c);
```

This code has to be converted explicitly into if-then-else as shown below:

```
if (a > 10)
  d = a + b;
else
  d = a + c;
```

C.7 Input Code with Multi-Dimensional Arrays

Input code that contains multi-dimensional arrays can be modified by making the arrays single-dimensional. Consider the following example:

```
int myArray[N][M];

...

temp = myArray[i][j]
```

The two-dimensional array “myArray” can be converted into a single-dimensional array as follows

```
int myArray[N * M];

...

temp = myArray[i*M + j]
```

Bibliography

[1] S. Gupta. *Coordinated Coarse-Grain and Fine-Grain Optimizations for High-Level Synthesis*. PhD thesis, University of California, Irvine, 2003.

[2] S. Gupta, N.D. Dutt, R.K. Gupta, and A. Nicolau. SPARK: A high-level synthesis framework for applying parallelizing compiler transformations. In *International Conference on VLSI Design*, 2003.

[3] SPARK parallelizing high-level synthesis framework website. http://www.cecs.uci.edu/~spark.

[4] S. Gupta, N. Savoiu, S. Kim, N.D. Dutt, R.K. Gupta, and A. Nicolau. Speculation techniques for high level synthesis of control intensive designs. In *Design Automation Conference*, 2001.

[5] S. Gupta, N. Savoiu, N.D. Dutt, R.K. Gupta, and A. Nicolau. Conditional speculation and its effects on performance and area for high-level synthesis. In *International Symposium on System Synthesis*, 2001.

[6] S. Gupta, M. Reshadi, N. Savoiu, N.D. Dutt, R.K. Gupta, and A. Nicolau. Dynamic common sub-expression elimination during scheduling in high-level synthesis. In *International Symposium on System Synthesis*, 2002.

[7] S. Gupta, N.D. Dutt, R.K. Gupta, and A. Nicolau. Dynamic conditional branch balancing during the high-level synthesis of control-intensive designs. In *Design, Automation and Test Conference*, 2003.

[8] S. Gupta, N.D. Dutt, R.K. Gupta, and A. Nicolau. Dynamically increasing the scope of code motions during the high-level synthesis of digital circuits. *Invited Paper in Special Issue of IEE Proceedings: Computers and Digital Technique: Best of DATE 2003*, 150(5), September 2003.

[9] S. Gupta, N. Savoiu, N.D. Dutt, R.K. Gupta, and A. Nicolau. Using global code motions to improve the quality of results for high-level synthesis. Technical Report CECS-TR-02-29, Center for Embedded Computer Systems, Univ. of California, Irvine, 2002.

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

[11] Z. Navabi. *VHDL: Analysis and Modeling of Digital Systems*. McGraw-Hill, 1993.

[12] M. Molina, J. Mendias, and R. Hermida. High-level allocation to minimize internal hardware wastage. In *Design, Automation and Test in Europe*, 2003.

[13] A. Nicolau and S. Novack. Trailblazing: A hierarchical approach to Percolation Scheduling. In *International Conference on Parallel Processing*, 1993.

[14] M. Girkar and C.D. Polychronopoulos. Automatic extraction of functional parallelism from ordinary programs. *IEEE Trans. on Parallel & Distributed Systems*, Mar. 1992.

[15] Edison Design Group (edg) compiler frontends. http://www.edg.com.

[16] S. Gupta, T. Kam, M. Kishinevsky, S. Rotem, N. Savoiu, N.D. Dutt, R.K. Gupta, and A. Nicolau. Coordinated transformations for high-level synthesis of high performance microprocessor blocks. In *Design Automation Conference*, 2002.

[17] AT&T Research Labs. Graphviz - Open source graph drawing software. http://www.research.att.com/sw/tools/graphviz/.