AUTOMATION OF IP CORE INTERFACE GENERATION FOR RECONFIGURABLE COMPUTING

Zhi Guo *Abhishek Mitra* *Walid Najjar*
Department of Electrical Engineering *Department of Computer Science & Engineering*
University of California, Riverside
{zguo, amitra, najjar}@cs.ucr.edu

ABSTRACT

Pre-designed IP cores for FPGAs represent a huge intellectual and financial wealth that must be leveraged by any high-level tool targeting reconfigurable platforms. In this paper we describe a technique that automates the generation of IP core interfaces allowing these to be used as C functions transparently from within C source codes using a reconfigurable computing compiler. We also show how this same tool can be used to support run-time reconfiguration on FPGAs by generating a common wrapper that interfaces to multiple cores.

1. INTRODUCTION

Compilers for reconfigurable platforms have two major roles: the automation of code transformations and optimizations, and increasing the productivity of the application developer. On the other hand, industry has invested tremendous financial and technical efforts on pre-designed intellectual property (IP) cores for FPGA-based platforms that are not only very efficient but have been thoroughly tested and verified. These IP cores come in the form of synthesizable HDL code or even lower level descriptions. They vary drastically with respect to their control and timing protocol specifications, which are intended to be interfaced to HDL-based designs.

Compilers for FPGA-based reconfigurable systems must therefore leverage that huge wealth of IP designs by allowing the user to import these into high-level language (HLL) source codes. To do so would require a wrapper structure that would hide the timing and stateful nature of the IP cores. It would make each core look, to the HLL compiler, as an un-timed side-effect free function call.

In this paper we describe a mechanism for the automatic generation of such a wrapper from a high-level description that is based on C with timing information. This approach is integrated in our ROCCC compiler. Run-time reconfiguration, where a sub-section of the circuit on an FPGA is switched between two functions, also require a careful and transparent interface between the static and dynamic parts. We show how this same approach can be used to support run-time reconfiguration.

The rest of the paper is organized as follows. Next section reviews related work. Section three introduces our compiler system. Section four presents our heuristic approaches to automate IP wrapping. The tool's support to dynamic partial reconfiguration is presented in section five. We validate our approaches in section six. Section seven concludes the paper.

2. RELATED WORK

Substantial amounts of effort have been devoted on standardizing or interfacing pre-designed IP cores.

Companies and organizations tried to define IP bus standards. For example, VSIA [5] specifies interface standards that allow IP cores to fit into "virtual sockets". Cores are designed using a standard internal interface and wrappers have to be provided to retarget cores into other buses. However, the current condition is that numerous standards exist and no standard is adopted widely.

Several projects focus on bus wrapping that connects IP cores with microprocessors. Glue logic is generated in [6] to connect processors to peripheral devices and hardware co-processors. A prefetching technique is introduced in [7] to improve bus wrapper's performance. The work in [8] raises the abstraction level and reuse IPs by extending traditional HDLs. A customization model for IP wrapping using UML class diagrams is proposed in [9]. In [11] the authors describe a system level approach for interfacing IP blocks generated by the behavioral synthesis tool itself. The I/O pins and timing information is fixed and known by the tool. This information, however, is not visible at the C level and the user cannot modify it.

Trident [10] is a compiler framework for floating point algorithms. The floating-point units are pre-designed IP units with known pipeline delay.

We have developed the ROCCC (Riverside Optimizing Compiler for Configurable Computing) system. ROCCC accepts applications written in untimed C code and generates synthesizable VHDL code for FPGA hardware. ROCCC also wraps IP cores when fed with wrapper abstractions in C with timing information.

3. ROCCC SYSTEM OVERVIEW

3.1. Compiler Overview

Figure 1 shows an overview of the ROCCC framework. We have separated the front and back ends to achieve some

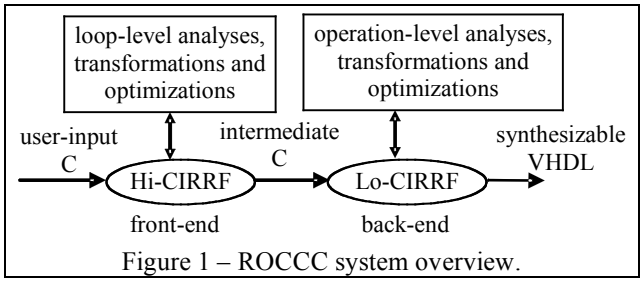


Figure 1 – ROCCC system overview.

modularity and eventually allow the use of other tools for either end.

ROCCC is built on the SUIF2 [1] and Machine-SUIF [2] platforms. It compiles C code into VHDL code for mapping onto the FPGA fabric of a CSoC device. Information about loops and memory accesses is visible in front-end intermediate representation (IR), Hi-CIRRF (Compiler Intermediate Representation for Reconfigurable Fabrics). Accordingly, most loop level analysis and optimizations are done at this level. ROCCC performs a very extensive set of loop analysis and transformations, aiming at maximizing parallelism and minimizing area.

Machine-SUIF is an infrastructure for constructing the back end of a compiler. Machine-SUIF's existing passes, like the Control Flow Graph (CFG) library, Data Flow Analysis library and Static Single Assignment library [3] provide useful optimization and analysis tools for our compilation system. We build the back-end using Machine-SUIF. The compiler's back-end converts the input from control flow graph (CFG) into data flow graph (DFG), and generates synthesizable VHDL codes. We rely on commercial tools to synthesize the generated VHDL codes.

3.2. Pipelining and Scheduling

For an original CFG, we categorize basic nodes into to two types: do-all nodes (parallel) and non-do-all nodes (sequential).

For do-all nodes, ROCCC exploits both instruction-level and loop-level parallelisms and aggressively pipelines the loop body to be able to execute multiple loops simultaneously [4].

A non-do-all basic node either belongs to a non-do-all loop or does not belong to any loop at all. The compiler utilizes predication to schedule the execution of non-do-all nodes' instructions. A predicator guards each pipeline stage. Multiple instructions might belong to the same pipeline stage and can be executed simultaneously. Predicators are passed inside basic nodes by *PFW* (predicator forward) instructions.

For the IP cores that are fully pipelined with compile-time known number of pipeline stages, ROCCC can simply synchronize its own pipelined circuit with the IP by adding more latches. In this paper, however, we discuss the most general cases, in which the IP cores use handshaking signals to communicate with external interface.

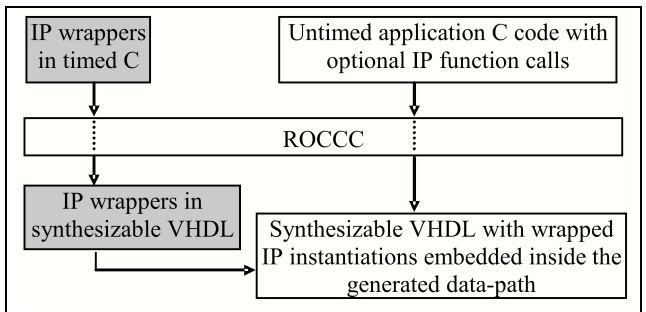


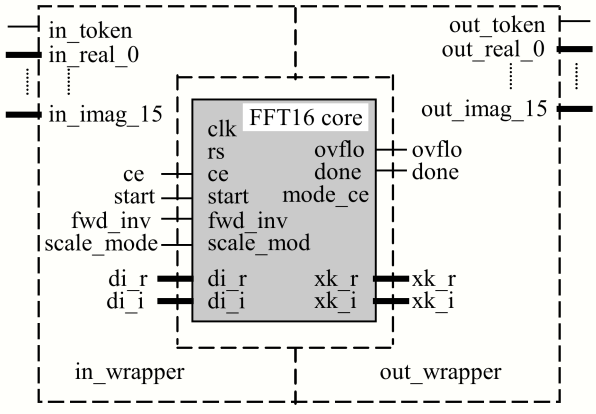
Figure 2 – ROCCC's workflow when IP function call present. Notice that an IP core is not necessarily a mandatory element of the main un-timed application C code. The grayed out part on the left only exists when there is an IP instantiation in the source code.

4. INTERFACE SYNTHESIS

As introduced in the system overview section, the ROCCC compiler generates synthesizable VHDL code for applications written in untimed C. In this section, we present our approach using the ROCCC system to wrap IP cores. The workflow is shown in Figure 2. Taking the high-level wrapper abstractions as input, ROCCC generates synthesizable wrappers in VHDL separately and these wrappers are instantiated as components in the outer circuit. We start with a 16 samples complex FFT IP core, taken from the Xilinx website that we use to demonstrate out approach in this paper.

4.1. An IP Core Example

The grayed out part of Figure 3 is a 16-point discrete fast



The grayed out square is the FFT16 IP core. The dashed area on the left is the in-wrapper and the dashed area on the right is the out-wrapper. A wrapper's interface to outside is always composed of one or more data ports and one token (either input or output token).

Figure 3 - *FFT16* and its wrappers

```
void in_fft16   (int in_token,  /*the core's input predicator*/
  int real_0, ... , int real_15,  /*16 real-component inputs*/
  int imag_0, ... , int imag_15,/*16 imaginary-component inputs*/
  int* CE, int* SCALE_MODE, /*pointers are output*/
  int* START, int* FWD_INV, int* DI_R, int* DI_I)
{
   int real_reg_0, ..., real_reg_15; /*internal registers to*/
   int imag_reg_0, ..., imag_reg_15; /*store the input data*/

   *SCALE_MODE = 1;
   *FWD_INV = 1;

   if(in_token == 1)     {
       wait_cycles_for(1);
       real_reg_0 = real_0;
       ......
       real_reg_15 = real_15;
       imag_reg_0 = imag_0;
      ......
       imag_reg_15 = imag_15;
       *START = 1; /*assert start signal in this cycles*/
       *CE = 1; /*assert ce signal in this cycles*/

       wait_cycles_for(1);
       *START = 0; /*de-assert start signal in this cycles*/

        wait_cycles_for(1);
        *DI_R = real_reg_0;
        *DI_I = imag_reg_0;
          ......
        wait_cycles_for(1);
        *DI_R = real_reg_15;
        *DI_I = imag_reg_15;

        wait_cycles_for(69);
        *CE = 0;   /*de-assert ce signal 69 cycles later*/
   }  }
```

store the 16 pairs of input data into internal registers in this cycle

export the 16 pairs of data into the core serially in 16 consecutive cycles

Figure 4 - Timed high-level abstraction of *FFT16*'s input wrapper in C. Comments explain the code.

Fourier transform core (*FFT16*). Pins *di_r* and *di_i* are respectively the real and imaginary serial data input; *xk_r* and *xk_i* are the output. *Ce*, clock enable, must be asserted only when the core is active. *Start* must be asserted two clock cycles ahead of the first pair of input data. *Done* is asserted when the first pair of output data is ready. *Fwd_inv* selects between forward or inverse FFT. *Scale_mode* selects from two scale-coefficients: 1/16 or 1/32. The *ovflo* pin indicates the core has generated an arithmetic overflow. *Mode_ce* input indicates when to sample *fwd_inv* and *scale_mode*.

4.2. High-level Wrapper Abstraction

An IP core requires a wrapper for both its input and output interfaces. In some cores these two interfaces have common signals that handle synchronization and handshaking. In our implementation the outer circuit within which the core is embedded covers this role.

Figure 4 lists the code for the input wrapper of FFT16 C. We use pointer type to distinguish output signals from input signals in the function declaration. The input set, which communicates with the outside, is composed of one token and several data variables. The output wrapper, not shown here, has the same structure. Thus both the input and output interfaces have the same structure as shown in Figure 3.

By its very nature, an interface to an embedded core must support timed activity. We thereby call it timed C. It can be written either by the end user or by the IP designer and possibly modified by the end user. In Figure 4, the function call *wait_cycles_for(n)* indicates the statements behind it must be executed *n* cycles later. Any statements between two adjacent *wait_cycles_for* function calls must be executed in one clock cycle. For example, *FFT16*'s timing protocol requires that *start* signal needs to be high for one clock cycle, two clock cycles ahead of the first pair of input data. In order to describe this timing requirement, the user just assigns *start* to one, calls *wait_cycles_for(1)*, de-asserts *start*, calls *wait_cycles_for(1)* again, and begins assigning input data into the core, as shown in Figure 4. The timing of signal *Ce*, which needs to be asserted for a 87-cycle period, are expressed in the same way. Parallel to serial converse is also describes naturally in the timed C code. At the beginning of the function body, *scale_mode* and *fwd_inv* are statically assigned to high. If desired, they can also be assigned by the wrapper's input signals at run-time in the same way as assigning *start* or *ce*. That way, the *FFT16* core can be easily switched between a forward FFT and an inverse FFT. This wrapping approach keeps the original IP core's functionality to a great extent and still stays at high-level. This wrapper plays a role of a bridge between the timing diagram in an IP core's data-sheet and the automatically generated synthesizable wrapper in VHDL.

```
for( each basic node b in cfg)
{
  for( each "WCF n" instruction instr in b)
       if( n > 1) {replace instr with n "WCF 1" instructions}

  assign the instructions between two adjacent WCF
            instructions into the same pipeline stage

  replace all "WFC 1" instructions in b into "PFW $vr1, $vr2"

 for( each PFW instruction instr in b)  {
       guard all instructions at the same pipeline stage as instr
       using instr's source operand as the predicator
     }

  if( b ends in a conditional branch instruction instr)
          convert instr to a Boolean instruction
}

 { add more combinational Boolean instructions if necessary to
  pass predicators appropriately from predecessor nodes to
  successor nodes.
 }
```

Figure 5 – Wrapper pipelining and scheduling heuristic

```
(1) .in_fft16
(2)  Node 1, {0} {2, 3}
(3) [L0] mov  $vr85.u1 <- in_fft16.in_token
(4) [L0]   mov  $vr84.u16 <- in_fft16.real_0
    ……
(5) [L0]  mov  $vr53.u16 <- in_fft16.imag_15
(6) [L0]  mov  $vr52.p1 <- in_fft16.CE
(7) [L0]  mov  $vr51.p1 <- in_fft16.SCALE_MODE
(8) [L0]  mov  $vr50.p1 <- in_fft16.START
(9) [L0]  mov  $vr49.p1 <- in_fft16.FWD_INV
(10) [L0] mov  $vr48.p16 <- in_fft16.DI_R
(11) [L0] mov  $vr47.p16 <- in_fft16.DI_I
(12) [L0] str    0($vr51.p16) <- 1 /*configure SCALE_MODE*/
(13) [L0] str    0($vr49.p16) <- 1 /* configure FWD_INV*/
(14) [L0] sne    $vr559.u1 <- $vr85.u1, 1 /*set if not equal*/

(15) Node 2, {1} {3}
(16) [L0]   not    $vr560.u1 <- $vr559.u1
(17) [L87] pfw  $vr471.u1 <- $vr560.u1

(18) [L87, P] mov   $vr167.u16 <- $vr84.u16,    $vr560.u1
……  /* latch the input data to the internal registers. */
(19) [L87, P] mov   $vr198.u16 <- $vr53.u16,    $vr560.u1

(20) [L87, P]  str  0($vr50.p1) <- 1, $vr560.u1/*assert START*/
(21) [L87, P]  str   0($vr52.p1) <- 1,  $vr560.u1 /*assert CE*/
(22) [L86]      pfw $vr472.u1 <- $vr471.u1
(23) [L86,P] str 0($vr50.p1)<-0, $vr471.u1/*de-assert START*/

(24) [L85]      pfw $vr473.u1 <- $vr472.u1
(25) [L85, P]  str   0($vr48.p16) <- $vr167.u16,    $vr472.u1
(26) [L85, P]  str   0($vr47.p16) <- $vr183.u16,    $vr472.u1
…/*export the 16 pairs of data elements serially to the IP core*/
(27) [L70]      pfw $vr488.u1 <- $vr487.u1
(28) [L70, P]  str   0($vr48.p16) <- $vr182.u16,    $vr487.u1
(29) [L70, P]  str   0($vr47.p16) <- $vr198.u16,    $vr487.u1

(30) [L69, E69]     pfw   $vr489.u1 <- $vr488.u1
……   /* wait for 69 clock cycles */
(31) [L2]               pfw     $vr556.u1 <- $vr555.u1
(32) [L1]               pfw     $vr557.u1 <- $vr556.u1
(33) [L1, P] str  0($vr52.p1) <- 0,    $vr556.u1 /*de-assert CE*/

(34) Node 3, [-1], {2, 1} {4}
(35) in_fft16._no_while_iTmp0:
(36) [L0]               ior     $vr558.u1 <- $vr557.u1, $vr559.u1
(37) [L0]               ret     $vr558.u1
```

The L field is the latch-level, a P field marks a predicated instruction. Line 12 and 13 configure the IP core. Line 18 through line 19 latch the 32 input data elements into internal registers. Line 20 and 21 assert *start* and *ce,* while line 23 de-asserts *start* (one cycle later). *Ce* is de-asserted in line 33, 85

Figure 6 - Back-end IR of *FFT16*'s input wrapper

4.3. Wrapper Synthesis

The timed high-level wrapper, the code in Figure 4 for example, is passed through the ROCCC compiler in Figure 1 as user input. Currently, the front-end does not do any optimizations on IP wrappers. The back-end first gets the control flow graph (CFG) of a wrapper, and converts the CFG into static single assignment (SSA) CFG. Starting from this SSA-CFG, the back-end constructs the DFG [Figure 5]. First, the pre-process pass converts *wait_cycles_for(n)* function calls into instruction "*WCF n"*, where *n* is an immediate operand. When building the data-flow, the compiler replaces each "*WCF n*" into n consecutive "*WCF 1*" instructions. Thus a *WCF* instruction has a clear hardware timing meaning, passing the predicator to the next pipeline stage. The compiler enforces all instructions between two adjacent *WCFs* to be at the same pipeline stage, as in the pseudo-code shown in Figure 5. This constraint ensures that the back-end's pipelining consists with the high-level C's timing semantics, and thereby satisfies the IP cores' timing requirement. In scheduling process, *WCF* instructions are replaced by *PFW* (predicator forward) instructions. *PFW* instructions pass predicators through the data-flow, while predicators guard other pipelined instructions.

The IR, right before VHDL emission of the FFT16 input wrapper, is shown in Figure 6. The IR records the predicated hardware actions with cycle-level timing constrains. In front of an instruction, the *L* field is the latch-level, namely, at which pipeline stage the instruction is executed. Instructions with a zero latch-level are combinational logic or even just wires if the opcode is *mov*. A predicator guards an instruction with a Boolean P field, which is the last source operand. For IP wrappers, a *str* (store) instruction with zero address offset is treated as a *mov* instruction, whose destination is the operand that the pointer is pointing to. From line 16 through line 33, the anticipated hardware does the following: monitoring the assertion of the predicator from outside and passing it (line 16 and 17), storing all the 16 pairs of input data into

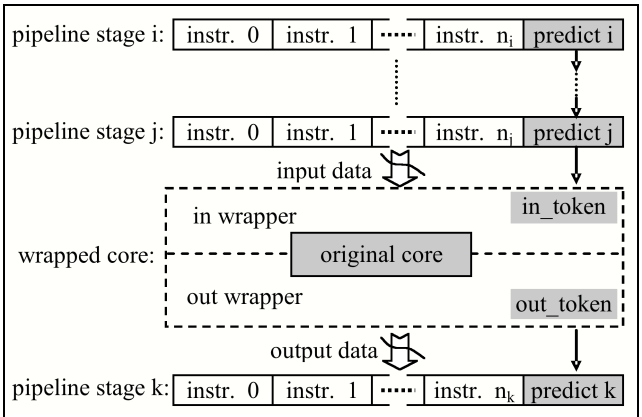


Figure 7 – The execution model of a wrapped IP inside the predicated data-path. A core's wrapper also consumes and produces predicators. From the point of view of outside, a wrapped IP core has an identical predication mechanism as other regular predicated instructions.

internal registers and asserting *start* and *ce* (line 18 through line 21), one cycle later (line 22) de-asserting *start*, one more cycle later (line 24) starting feeding input data into the core pair by pair serially (line 25 through line 29), after 69 more cycles waiting (line 30 through line 31), de-asserting *Ce*. The compiler's very last pass emits the VHDL code for the wrappers. The combinational instructions become combinational logic in hardware and pipelined instructions become sequential logic. From the point of view of outside, the generated wrappers (the wrappers of FFT16 in Figure 3, for example) have unified interface: input data ports and one input predicator at input side, and output data ports and one output predicator at output side. Figure 7 shows a wrapped IP core embedded in a compiler-generated outer circuit. The wrapped IP core has an identical interface as that of other regular predicated instructions.

5. DYNAMIC PARTIAL RECONFIGURATION

We also use our tool to support dynamic partial reconfiguration. Dynamic partial reconfiguration at runtime allows re-use of FPGA resources to obtain a plurality of functionality, from the same hardware block, but at different times, and also without affecting the static parts of the device. The compiler generates the wrappers for each IP cores that need to be dynamically reconfigured.

The design flow involves the generation of the static logic along with partial reconfigurable logic (wrapped IP cores). Thereafter the FPGA is floor planned to allocate pre-determined areas for the dynamic logic and static logic respectively. The area dedicated to the dynamic logic, also known as the PR-Block (Partial Reconfigurable Block), is such that it may allow for the largest IP block to be placed and routed within. I/O and communication of the static logic with the PR-block takes place using certain pre-configured CLBs known as slice-macros. These slice-macros need to be manually placed around the boundary of the PR-block. We have employed the Xilinx PlanAhead visual floorplanning tool for iterative design and placement. The final stage of the partial reconfigurable flow generates 'N' static bitstreams and 'N' partial bitstreams, where 'N' is the number of different IP blocks to be configured in the PR-Block. Each of the 'N' static bitstream contains the static design with the partial-reconfigurable block numbered 'N' already programmed into the stream, while each of the 'N' partial bitstreams contains the logic to just program the PR-Block with the functionality of the 'N'th IP core. Thus the system may choose to start with one of the static bitstreams during power-up and thereafter reprogram the PR-Block with the desired functionality.

6. EXPERIMENTAL RESULTS

We have used four Xilinx IP cores, shown in Table 1, in our experimental evaulation. *Cordic* performs a rectangular-to-polar vector translation. The input is a vector (X, Y) in a Cartesian coordinate and the IP's outputs are the magnitude and the angle in a polar coordinate. *DCT8* performs a one-dimensional 8-point discrete cosine transform. *FFT16* is the IP core shown in Figure 3. *RS encode* is a (15, 13) Reed-Solomom encoder. It has a 13-symbol code block and outputs 15 symbols, including 13 original data symbols followed by two check symbols. In Table 1, *Total area* is the total circuit including the input and output interfaces, and the IP core itself. *Area (slice)* and *Area (%)* are the area utilization in the number of slice and in percentage with respect to the entire circuit, respectively. *Addtl Cycl* is the number of extra clock cycles after the addition of the wrappers. *Total cycle* is the total number of clock cycles to compute on one set of input data. *DCT8's* input data size is 8-bit while its output data size is 19-bit. *RS encode*'s input and output data sizes are 4-bit. Both *Cordic* and *FFT16's* input and output sizes are 16-bit. The target architecture is the Xilinx Virtex-II XC2V8000-5 FPGA having 46592 slices.

Table 1 - Results of the wrappers for four Xilinx IPs

		Cordic	DCT8	FFT16	RS encode
input	area (slice)	2	55	532	53
	area (%)	3	6.7	24	64
	addtl cycl.	1	1	1	1
output	area (slice)	2	426	290	9
	area (%)	2	52	13	11
	addtl cycl.	1	1	1	1
total area	area (slice)	663	817	2183	83
	clock (MHz)	123	68.7	45.0	96.4
	total cycles	23	23	200	20

Cordic has only two inputs and two outputs, and a simple handshaking protocol. *DCT8's* input wrapper latches all eight 16-bit input data. These are fed serially into the IP core. The wrapper asserts the *new_data* signal to be high during the data transmission and de-asserts it right after the transmission, following the timing requirement of the *DCT8* IP core. The output wrapper monitors the *output_ready* signal from the core and starts receiving the eight serial output data elements once it is asserted high. On the next clock cycle after all the eight output elements have been collected, the wrapper exports them all in parallel. *FFT16* requires similar serial to parallel and parallel to serial conversions, except that the IP imports and exports data in pairs, one real component

Table 2 - Reconfiguration time for static and partial reconfiguration on a Xilinx Virtex-2 PRO (XC2VP30)

design type (Static/Partial)	# of slices	Btstrm size (Kbits)	prgrm. time JTAG(ms)	program. time SelectMAP(ms)
static conf	13696	1415	2318	48
DCT8 prtl recnf	378	216	354	7.3
FFT8 prtl recnf	512	426	698	14.3

and one imaginary component. *FFT16's* input timing is different in the way that *start* and *ce* (clock enable) have certain cycle-level specifications described in the previous section. The generated interface meets all those timing requirements. The *FFT16* core's overflow output pin, *OVFLO*, is duplicated and exported by the wrapper to the outside data-path for further use. In *RS_encode*'s output, the first 13 data elements are the data symbols that were fed into the IP. From the point of view of the outside data-path, these data are known and do not necessarily need to be recovered from the IP core again, and only the two check symbols, which follow the first 13 data elements, are needed. The *RS_encode* IP core utilizes output signal *info* to indicate the present of the check symbols. The generated wrapper monitors *info*'s de-assertion and latches the check symbols in an appropriate timing.

ROCCC wraps these IPs so that they have unified outside interface. These four examples illustrate ROCCC's capability to meet various timing protocols of IP cores. The execution time overhead at both the input side and output side for these four examples is one clock cycle. The area of wrappers accounts for 2% ~ 64% of the corresponding wrapped cores. Most of the wrappers' area cost comes from the registers used to do serial to parallel and parallel to serial conversion. Compared to modern FPGAs' capacity, this overhead is quite small.

We measured the time required to load a static bitstream as well as the time required for programming partial bitstreams on the FPGA [Table 2]. JTAG and SelectMAP are two interfaces for reconfiguration of the FPGA. Since the partial bitstreams are smaller in size than the static bitstreams, a partial reconfiguration can be achieved in a shorter time vis-à-vis complete reconfiguration.

7. CONCLUSION

Increasing silicon capacity requires both higher level design methods and easier intellectual property core reuse. ROCCC, a reconfigurable computing compiler, is designed to take applications in C as input and generate RTL VHDL code. In this paper, we introduced one aspect of ROCCC's functionalities, the IP wrapper generation.

As the input to the ROCCC system, users write IP wrappers in high-level timed C. Clock cycle delays are described as function calls and users do not have to implement any cycle-level details in the input abstraction. Constrained by the delay function calls, ROCCC converts the wrapper from control flow graph to data flow graph. The compiler schedules pipelined instructions using predication. Wrapped IP cores have identical interface compared with the outer predicated circuit that also generated by ROCCC.

The wrappers of the IP core examples meet the various timing protocol requirements, and unify the IP cores' interface with the outer compiler-generated circuit. The results show that the execution time and area overhead are reasonable low. We also show the same tool can be used to support run-time reconfiguration on FPGAs by generating one wrapper that interfaces to multiple cores.

8. REFERENCES

[1] SUIF Compiler System. http://suif.stanford.edu, 2006

[2] Machine-SUIF. 2006 http://www.eecs.harvard.edu/hube/research/machsuif.html

[3] G. Holloway. The Machine-SUIF Static Single Assignment Library. Division of Engineering and Applied Sciences, Harvard University 2002.

[4] Z. Guo, B. Buyukkurt, W. Najjar and K. Vissers. Optimized Generation of Data-path from C Codes for FPGAs, Int. ACM/IEEE Design, Automation and Test in Europe Conference (DATE 2005). Munich, Germany, March, 2005.

[5] Virtual Socket Interface Association (VSIA), http://www.vsi.org/ , 2006

[6] P.Chou, G. Ortega, G. Borriello. Interface co-synthesis techniques for embedded systems, Int. Conf. on Computer Aided Design, San Jose, USA,1995.

[7] R. Lysecky and F. Vahid. Prefetching for Improved Bus Wrapper Performance in Cores, ACM Transactions on Design Automation of Electronic Systems, Vol. 7, No. 1, pp. 58-90, January 2002.

[8] J. Zhu. MetaRTL: Raising the abstraction level of RTL design, Design, Automation, and Test in Europe, Munich, Germany, 2001

[9] V. Stuikys, R. Damasevicius. Soft IP Customisation Model Based on Metaprogramming Techniques, Informatica, Lith. Acad. Sci. 15(1): 111-126 (2004)

[10] J. Tripp, K. Peterson, C. Ahrens, J. Poznanovic, M. Gokhale. Trident: An FPGA Compiler Framework for Floating-Point Algorithms, int. Conference on Field Programmable Logic and Applications (FPL 2005). Finland, 2005.

[11] R. Mukherjee, A. Jones, P. Banerjee, System Level Synthesis of Multiple IP Blocks in the Behavioral Symthesis Tool, Int. Conf. on Parallel and Distributed Computing and Systems (PDCS), November 2003.