Integrating Scheduling and Physical Design into a Coherent Compilation Cycle for Reconfigurable Computing Architectures

Kia Bazargan
ECE Department
University of Minnesota
Minneapolis, MN 55455
kia@ece.umn.edu

Seda Ogrenci
CS Department
UCLA
Los Angeles, CA 90095-1596
seda@cs.ucla.edu

Majid Sarrafzadeh
CS Department
UCLA
Los Angeles, CA 90095-1596
majid@cs.ucla.edu

ABSTRACT

Advances in the FPGA technology, both in terms of device capacity and architecture, have resulted in introduction of *reconfigurable computing machines*, where the hardware adapts itself to the running application to gain speedup. To keep up with the ever-growing performance expectations of such systems, designers need new methodologies and tools for developing reconfigurable computing systems (RCS). This paper addresses the need for fast compilation and physical design phase to be used in application development / debugging / testing cycle for RCS. We present a high-level synthesis approach that is integrated with placement, making the compilation cycle much faster. On the average, our tool generates the VHDL code (and the corresponding placement information) from the data flow graph of a program in less than a minute. By compromising 30% in the clock frequency of the circuit, we can achieve about 10 times speedup in the Xilinx placement phase, and 2.5 times overall speedup in the Xilinx place-and-route phase, a reasonable trade-off when developing RCS applications.

1. INTRODUCTION

Reconfigurable computing systems (RCS) are the next promising alternatives to costly high performance multi processors. Such systems usually consist of a host processor, (tightly) connected to a reconfigurable "co-processor" called Reconfigurable Functional Unit (RFU). The RFU can be an FPGA (Field Programmable Gate Array) which resides either on the same die as the host processor, or on a separate chip, connected to the processor through a bus. An example of an RCS architecture is shown in Figure 1-a (for a survey on RCS, refer to [4]). As the FPGAs get larger and faster, both the number and complexity of the modules to load on them increase, hence better speedups can potentially be achieved by exploiting FPGAs in hardware systems.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
DAC 2001, June 18-22, 2001, Las Vegas, Nevada, USA.
Copyright 2001 ACM 1-58113-297-2/01/0006 ...$5.00.

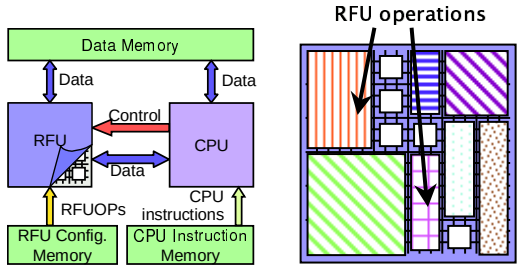

Figure 1: (a) The architecture of an RCS (b) An FPGA configured with seven RFUOPs.

In the past decade, we have witnessed notable benefits in exploiting reconfigurable computing, despite the fact that such benefits are limited to specific classes of applications. Hauck has reported many such applications in [5]. Figure 1-b shows an FPGA which is configured with seven RFU operations (RFUOPs). Examples of RFUOPs are adders, constant multipliers, shifters, register and multiplexers.

1.1 Previous Work and Current Challenges

Despite all the success stories on FPGA implementations of some applications, FPGAs are not as commonly used today as one would expect. Among the reasons for the limited use of FPGAs are small on-chip memories and low communication bandwidth between the host processor and the RFU.

On the other hand, if the current trend in the FPGA technology improvements continue, the RFU systems of the future will have enough on-chip memory both for RFU configuration and program data. There will be specialized hardware in the RFU fabric to handle floating point operations. Furthermore, as the SOC technology becomes prevalent, both the host CPU and the RFU can be placed on the same die and as a result, high bandwidths between the two can be achieved.

Apart from the architectural challenges, there are serious obstacles in realizing a general-purpose RCS in terms of design techniques, as well as runtime support systems. Today's RCS designer should have very good programming skills as well as detailed knowledge of the target hardware. There are currently no compilers that can directly compile a C program to configuration bit stream for commercial FP-

GAs[1]. Other compilers that convert C programs to hardware description languages have been reported in the past few years, although they are still in their developing stages (e.g., [6]). Some compilers compile from C-like languages to hardware languages.

The process of compiling hardware description languages to hardware (mostly logic synthesis, placement and routing) is very time-consuming: minutes for small designs and hours for larger ones. Long synthesis/physical design cycles mean longer application development and debugging time, in turn a hurdle for the widespread use of RCS in common computing platforms. The place-and-route algorithms proposed for FPGAsare generally very slow or do not generate high quality placements. The only fast placement algorithm reported in the literature is a work by Callahan *et. al.* [3] which is a linear time algorithm for mapping and placement of data flow graphs on FPGAs, but places the datapath modules linearly, obviously eliminating lots of good placement solutions.

1.2 Goals

Ideally, a programmer should be able to write his/her code (e.g., DSP, encryption, compression, video/graphics, vector calculations, etc.) in a high-level language and compile it as we compile C programs today. Most of the optimizations should be done automatically (yet allowing the programmer to interfere), and the compilation/debugging cycle should be short. Such a scenario does not leave room for logic synthesis, placement and routing of the circuit at the gate-level. Extensive use of versatile IP (intellectual property) libraries, as well as templates for routing and placement seem to be a necessity.

In this paper, we focus on high-level synthesis (HLS) and physical design (PD) of statically reconfigurable systems. That is, the RFU configuration for each program is generated and stored at compile time, and mapped to the RFU fabric before the program starts running. We have presented fast, high quality RFUOP allocation, operation binding to RFUOPs and operation scheduling (high-level synthesis) combined with a computationally efficient placement method (part of the physical design process) that can place RFUOPs compactly.

We will show that better results can be achieved by allowing the HLS and PD stages to interact so that the transition from the front-end tools (compiler and HLS) to the back-end tools (PD) is done more smoothly. The result is faster convergence to the desired design specifications. In short, the scope of this paper is datapath generation and physical design and their interactions. So, for example, we will not deal with runtime support features such as initializing the RFU with the data for the computations.

1.3 Outline

The rest of this paper is organized as follows. Section 2 describes the overall flow of our method. Section 2.4 describes our tool and its different componenets, namely, high-level synthesis and the placement method. Section 3 focuses on our scheduling algorithm. Section 4 contains the experiments, and finally, Section 5 concludes the paper and discusses future directions for our work.

[1]There are, however, compilers for prototype RCS architecutres such as CMU's Chimaera, Berkeley's Garp and University of Washington's RaPiD.

2. COMPILATION AND SYNTHESIS FLOW

Figure 2 shows the flow we propose to compile C programs to object code (to run on CPU) and RFU bit streams (modules representing RFUOPs). First, a modified "gcc" compiler [8] is used to convert loop bodies of a program to data flow graphs (DFGs). For simplicity, we discard all the bit-width information and only deal with the types of operations (refer to Section 5 for more discussion on this issue).

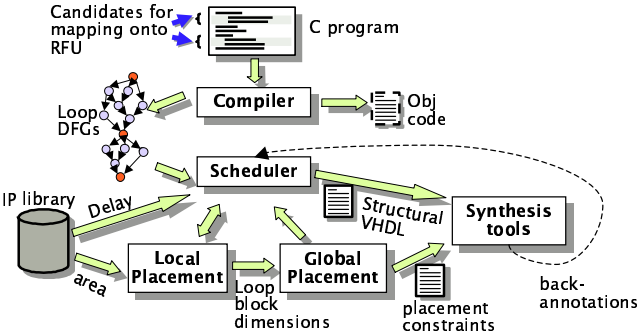

Figure 2: Our hardware/software compilation flow

The loop bodies are initially all candidates for being mapped to RFU fabric as *loop blocks* (the selection of the parts of the program to be mapped to the RFU, is known as hardware/software partitioning. See [7]). A *loop block* is a set of RFUOPs that perform the operations in a loop body. It consists of different RFUOPs (such as adders, shifters, registers, multiplexers, etc.). The scheduler uses the area and delay information in the IP library (we used Xilinx CoreGen 2.1i in our experiments) to assess the speedup and the area of each of the loop blocks. The scheduler calls the *local* and *global* placement algorithms (Section 2.4) to determine the location of the individual RFUOPs on the chip. At the end, the hierarchical structural VHDL code for the datapaths (that is, loop blocks), as well as the control path for the loop bodies are created.

The VHDL code is translated into netlist using Synplicity. The netlist, together with the *placement constraints* (provided by our placement algorithms), is used as input to Xilinx Design Manager to generate the final place-and-route. Although we have used Xilinx chips as the target architectures for the RFU, our methods are not limited to any specific products, as long as the IP library is provided.

Currently we do not generate the object code for the host processor and the interfacing routines between the RFU and the CPU (the dotted box in Figure 2). Also missing from our flow is any automatic feedback from the commercial synthesis tools to the previous stages (the dotted curve in Figure 2). We intend to add these features to our work in future.

2.1 Physical Design Specification

Our tool starts with reading in a library description file. The library file contains all operation types and their delays on the CPU, as well as the information on the IP modules (RFUOPs). For each IP module, the library file specifies the delay, dimensions, and the operation types it can handle.

The user can specify hardware properties such as how tightly the CPU and RFU are coupled and the size of the RFU fabric. Also, the user has control over a number of scheduling (e.g., how much effort should be put in minimizing the number of connections between RFUOPs as opposed

to minimizing the latency) and placement parameters (e.g., the quality/time adjustments for the loop block placements, how the loop blocks should be placed on the chip, etc.).

2.2 High-level Synthesis

The input to the scheduler is the data flow graph of the loop bodies in a program. After reading the DFG the scheduler (Section 3) schedules each of the loop blocks *tentatively* on the RFU. Based on the communication model, the operations in a loop block might be scheduled entirely on the RFU or on both the RFU and the CPU.

For each loop, the scheduler calculates the latency gain (i.e., the gain in latency when run on RFU as opposed to entirely running on CPU), and stores this number so that after generating the local placements for the loop blocks, it can decide on which loops to actually map to the RFU (based on the speedup/area numbers). The user can override such decisions and manually select individual loops for mapping onto the RFU.

2.3 Physical Design Aware HLS

An important feature in our tool is its *physical design aware* high-level synthesis. This feature makes the tool powerful, and makes the interfacing with different back-end tools easier. By interacting with the physical design stages, our high-level synthesis phases generate better schedules for the back-end tool. Generally, the higher levels of the design cycle can make more profound impacts on the qualities of the design than the lower levels can. So, by making high-level decisions that are more consistent with the back-end tool capabilities, the convergence to a desired solution would be faster.

Examples of the physical design considerations in our high-level algorithms are:

- Considering the congestion
  Our scheduler considers the *connectivity* (defined in Section *3.2.2*) of the RFUOPs when scheduling operations on the allocated RFUOPs. This in turn results in less congestion and hence easier routing by the back-end tools (Section 4.2 presented the experimental results supporting this claim).

- Avoiding high fanin modules
  At the binding step (i.e., when deciding the RFUOP on which an operation should be scheduled), the tool not only considers how early a resource can finish the operation, but it also considers how large the fanin set of the resource is.

2.4 Hierarchical Placement

By taking a hierarchical approach to the placement problem, our tool is able to generate high quality placements very quickly. After the scheduling is done for all the loops, a hierarchical two-stage placement algorithm is used to determine which loop blocks fit on the RFU and the location of those that fit. The placement method consists of the *local placement* step that determines the relative locations of RFUOPs inside a loop block, and the *global placement* phase that determines the location of the loop blocks on the RFU.

The reason that we treat the local and the global placement problems differently is that the two are different in nature. The number of interconnections between RFUOPs in a loop block is considerably more than the number of wires connecting loop blocks. The local placement stage tries to minimize the area and the wire length of each loop block individually. In our current implementation, we use Wong-Liu's simulated annealing floorplanning algorithm for this stage. Since the number of modules in each loop block is usually small (tens of modules in each loop block), and there is not much wires between them, the annealing process converges much faster compared to ASIC floorplanning. The local placement algorithm keeps a number of best shapes (in terms of the cost function, which can be a combination of area, wire length and aspect ratio of the loop block) for each loop block[2]. Keeping multiple shapes for each loop block gives more flexibility to the global placement phase. As the experiments show, the local placement generates placements for all the loop blocks in about 50 seconds on the average, although that is adjustable too.

The user can control various parameters in the local placement (e.g., how much effort should be put into minimizing wire length as opposed to the area of the loop blocks). Each loop block can be placed using different values of such parameters. This is particularly useful if we would like to minimize the wire length on a loop block with the most critical path, while our main concern with regards to another loop block might be its large area.

After the local placement is generated for each loop block, the tool tries to globally place the loop blocks on the RFU. The global placement is done in decreasing order of *loop gain*. The loop gain of a loop block is its speedup divided by its area. If there is no room for a loop block, the next loop block in the list is considered for placement. The global placement method uses a very fast algorithm called "KAMER-BF Decreasing" that we have developed. KAMER-BF-Decreasing sorts the modules based on their area, and inserts them using the KAMER-BF method [2].

2.5 VHDL Code Generation

Finally a set of hierarchical VHDL files are generated which describe the datapath and corresponding control path for each loop block. The placement information is written in a file for the back-end tools. In our experiments, we used Sinplicity 6.0 for compiling the structural VHDL code to .edif netlist format and Xilinx Design Manager 2.1 as the back-end tool. The placement information was written in the .ucf (user constraints file) format for the Xilinx place-and-route stage.

3. STATIC SCHEDULING

As we stated earlier, we have addressed the scheduling problem for the statically configured RFU that is loosely coupled with the CPU (i.e., the data communication speed between the two is not high, e.g., through PCI bus). There are multiple RFUOPs configured on the RFU for each loop block. At any time, any number of RFUOPs can compute in parallel. Choices of the RFUOPs from the IP library, for placing on the RFU (as loop blocks), as well as their locations on the chip, is made at compile time. Once the set of RFUOPs and their locations is decided, the configuration bit-stream is generated and stored as a "hardware code". When the program is loaded into memory, the configuration is streamed to RFU, and then the program starts running.

[2]Our experiments showed that keeping ten shapes is enough, although this number could be changed.

Currently, we do not deal with the initialization of the loop blocks, i.e., transferring data from the CPU to corresponding loop block registers and RFU memory blocks. Also, we do not map the vector variables (arrays, structures, etc.) used in the input C program to memory blocks on the RFU. We convert all array accesses to scalar variable accesses and map those scalar variables to registers on the RFU. Our future plan is to implement array variables as well.

3.1 Static Scheduling Problem Formulation

A program written in a high-level language can be represented by a control/data flow graph (CDFG). We only consider loop bodies for optimization purposes, and hence use the data flow graph (DFG) instead. We define set $OpTypes$ as the set of all operation types that the hardware can perform, either on CPU or on RFU (CPU can perform all operation types). A function $cpuDelay : OpTypes \mapsto Z^+$ maps operation types to number of cycles each takes on CPU. Parts of the program that we are considering for scheduling (partly) on RFU are represented in a DFG as defined in Equation 1. Nodes in DFG represent operations and edges show precedence relation between operations.

$$DFG =< V, E, type >, E \subseteq V \times V, type : V \mapsto OpTypes \quad (1)$$

The IP library is represented as set $Library$ (Equation 2). Each element in $Library$, which represents an RFUOP, is a 3-tuple (T_i, d_i, S_i). T_i is the set of all operation types that the RFUOP can perform. d_i is the delay of the operation on RFU, and S_i is the set of shapes (width, height) the RFUOP can take.

$$Library = \{(T_i, d_i, S_i) \,|T_i \subset OpTypes, d_i \in Z^+, S_i \subset (Z^+)^2\} \quad (2)$$

The static scheduling for reconfigurable computing systems is defined as the problem of finding the set $resources$ (Equation 3) and functions $bound : V \mapsto (CPU \cap resources)$ and $start : V \mapsto Z$ such that the schedule is valid and the total latency is minimized. $Resources$ is the set of modules *instantiated* from the library. There could be multiple copies of the same $Library$ type on the chip. We define function $finish$ in Equation 4.

$$\begin{aligned} resources \quad = \quad & \{r_i = (l_i, x_i, y_i, w_i, h_i)| \\ & l_i = (T_j, d_j, S_j) \in Library, x_i, y_i \in Z, \\ & (w_i, h_i) \in S_j, w_i \leq W, h_i \leq H\} \end{aligned} \quad (3)$$

$$finish : V \mapsto Z^+, finish(op) = start(op) + d_{bound(op)} \quad (4)$$

The latency of the schedule is defined as the time the last operation ends, i.e., $max\{finish(op)\}$. It has been shown that even when all resources have the same delay, the scheduling problem is NP-hard. Our problem is somewhat similar to scheduling operations with different voltage levels (for power optimization), but with different cost models.

3.2 Static Scheduling Algorithm

Our scheduling method is non-preemptive, i.e., once an operation starts execution, it will not be interrupted before completion. We have modified the well-known list-scheduling algorithm to account for different running times of the operations on different resources.

For a scheduled operation, we define function $selfGain$ as in Equation 5. Intuitively, this function is a measure of how early an operation can be scheduled if a new resource, capable of implementing the operation, is added to RFU. In other words, $selfGain$ is an indication of how an operation is bound to be scheduled at its current time due to dependency constraints, as opposed to resource constraints (see Figure 3). The difference in latency of the new resource and $bound(op)$ should also be added to $selfGain$, but we did not show it in Equation 5 for simplicity.

$$\begin{aligned} selfGain \quad : \quad & V \mapsto Z \\ & selfGain(op) = start(op) - \\ & max\{finish(pred)|(pred, op) \in E\} \end{aligned} \quad (5)$$

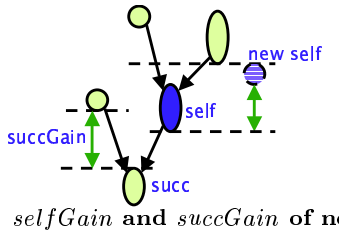

Figure 3: $selfGain$ **and** $succGain$ **of node** $self$**.**

Similarly, function $succGain$ is defined in Equation 6[3]. This function is a rough estimation of how the latency of the whole schedule will be affected if the operation is scheduled on a new resource. It speculates how early the successors of a node can be scheduled, if the operation is moved to a new resource r.

$$\begin{aligned} succGain \quad : \quad & V \times resources \mapsto Z \\ & succGain(op, r) = \\ & max\{finish(pred)|(pred, op) \in E\} + d_r \\ & -min\{start(succ)|(op, succ) \in E\} \end{aligned} \quad (6)$$

Our method first allocates one resource for each operation type that appears in the loop block (e.g., if there are three additions and two shifting operations, the method allocates one adder and one shifter). Then, in a resource allocation loop, prospective new resources are examined for potential gain in the overall latency, and the space they occupy on the RFU. The candidate resource types are examined in the order of the speculative gain on latency, the gain being a linear combination of $selfGain$ and $succGain$.

If the new resource results in latency improvements, it is added to the $resources$ set and the nodes are rescheduled using list scheduling. Otherwise, the next candidate resource in the list is examined.

3.2.1 Operation Scheduling (list-scheduling)

After a new resource is allocated, the DFG operations are scheduled on the instantiated resources using the list-scheduling method. When an operation is extracted from the list scheduling queue, allocated resources are examined for compatibility (i.e., of the same type) and the one which can finish the operation the earliest is chosen. Ties are broken by using the *connectivity* of the resource to the resources which implement operation's predecessors.

3.2.2 Register and Multiplexer Instantiation

After the operations are scheduled on the current resources, registers are inserted where a value is needed more than a cycle later than the time it is generated. A left-edge heuristic

[3]The function we used in our code is more complex than what is defined in Equation 6, but still the same time complexity, i.e., $O(d)$ where d is the maximum density of the nodes in the DFG.

method is used to minimize the number of registers. After register instantiation, the fanins of the RFUOPs as well as registers are examined and multiplexers are inserted at the inputs of the resources that get input from more than one source.

An important difference between our scheduling problem, and the traditional one is that here the area of steering logic components is comparable to functional units (e.g., an 8-bit 2:1 multiplexer takes 4 CLBs, the same number as a registered 8-bit adder takes). For this reason, we provide special heuristics in our scheduling algorithm to address area minimization in the higher levels of the design. This results in a more smooth transition to the back-end tools and less surprises for the designer. Section 4.2 presents experimental results that show the effectiveness of our approach compared to a traditional method.

Since routing resources are limited in FPGAs, high congestion and large fanin/fanouts should be avoided[4]. To do so, we employ two heuristic methods: one during the list-scheduling and another one during multiplexer insertion. The former has only approximations on how the final connections between the modules would be (because the registers and multiplexers have not been instantiated yet), where the latter knows the exact connections.

The heuristic used in list-scheduling to minimize congestion and fanin, uses the *connectivity* parameter, defined in Equation 7. The heuristic used in multiplexer insertion, replicates high-fanin RFUOPs and registers as a post processing phase.

$$\begin{aligned} &connectivity : resources \times resources \mapsto Z, \\ &\quad connectivity(r_1, r_2) = |C| \\ &\quad C = \{(i,j) | (op_i, op_j) \in E,\ bound(op_i) = r_1, \\ &\quad bound(op_j) = r_2\} \end{aligned} \tag{7}$$

4. EXPERIMENTAL RESULTS

We used Xilinx XC4000XL series as the target FPGA architecture in our Experiments (further experiments with Virtex devices show similar results). As the IP library, we used Xilinx CoreGen 2.1i IP blocks. We generated only 8-bit RFUOPs, although our methods are capable of handling larger modules as well. We assumed that the configurable component runs at half the clock speed of the host processor (e.g., Virtex's grade 5-6 speed can achieve 150 MHz on the data path, and the host CPU be a 300MHz Pentium processor). Since we target loops for optimization, we used the pipelined version of the operations in the IP library. We used Honeywell adaptive computing benchmark programs as well as some MediaBench programs. Table 2 shows the test programs we used for scheduling.

4.1 Scheduling Experiments

In this experiment, we ran our scheduling method on different benchmarks. The RFU area was set large enough so that no loop blocks are rejected due to lack of RFU space. The tool removed loop blocks with no speedup from the RFU, though (i.e., loops that run faster on CPU). Table 2 summarizes the results. The second column ("Total number of loop blocks") shows how many loop blocks were represented in the DFG file. The third column shows the the latency on a uni-processor host machine. The fourth column lists the number of loops that actually showed speedup on the RFU. Column five ("Scheduled cycles") show the latency of the whole DFG after scheduling, counting both loop blocks that were scheduled solely on the CPU and those which gained speedup on the RFU (overall speedup is the ratio of columns 5 and 3). Finally, the last two columns correspond to the latency and the speedup of the mapped loop blocks (i.e., those which gained speedup on the RFU).

Benchmark programs	# loops	CPU cycles	loops mapped	schd cycles	mapd cycles	spdup /mppd iter.
DCT	2	36	2	18	18	2.00
DFT	2	29	1	27		1.50
FFTGen	4	47	4	36	36	1.30
Image	15	220	9	125	82	2.15
Comp/dcmp	19	191	12	141	90	1.55
Jpeg	8	195	7	90	84	2.25

Table 2: Original number of cycles and scheduled number of cycles per iterations of the loops.

As we can see, even though the RFU fabric is working with half the frequency of that of the CPU, we still can gain speedups of about 2. Some individual loops show even speedups of around 4. The scheduling time was less than a second in all the cases.

4.2 Connectivity and Routing Experiments

As described in Section *3.2.2*, we employ heuristics at the operation scheduling step to decrease the number of connections in the final design. High connectivity not only increases congestion and makes the routing harder and slower, but it can also indirectly affect the area of the design. The increase in the area is caused by resource replication. Since some IP libraries have limited number of multiplexer types (e.g., we found that using multiplexers with more than three inputs causes routing failure in some cases), the HLS tool has to either use cascaded multiplexers or replicate functional units if the number of fanins is more than the maximum multiplexer inputs.

		L. Bound		No Con		Min Con	
DFG(ID,#op)*	λ	con	#m	con	#m	con	#m
Image (L15, 39)	22	72	39	74	35	**71**	**34**
Image (L5, 84)	22	**197**	85	205	87	205	**84**
versat (L2, 34)	16	94	33	97	**31**	**90**	**31**
JPEG (L5, 25)	16	70	12	75	25	**68**	**9**
JPEG (L7, 57)	24	252	**91**	262	103	**248**	94
Average		137	52	143	56	**136**	**50**

*ID is the loop identifier. "#op" is # DFG nodes.
#m is the number of modules in the loop block.

Table 3: Number of connections after register and multiplexer instantiation.

The purpose of this experiment is to show the effectiveness of the heuristic that we use to reduce the number of connections and multiplexers. We tested our method (called "Min Conn" in Table 3) against two other algorithms. The descriptions of the three algorithms are listed below.

- Min Conn: Described in Section 3.2. Considers *connectivity* during list scheduling.
- Lower Bound: Allocates the minimum number of resources (not considering registers and multiplexers) for the minimum latency schedule.

[4]The former makes the routing phase hard, while the latter has a negative impact both on the routing and clock speed

	Without .ucf			With .ucf				Ratio	
Benchmark programs	Xilinx place time (s)	Xilinx route time (s)	Crit. path (ns)	Xilinx place time (s)	Xilinx route time (s)	Crit. path (ns)	Our place time (s)	PD speed	Clock period
DCT	46	71	14.538	17	43	16.788	6.9	175%	1.15
DFT	53	90	12.912	17	52	16.945	4.3	195%	1.31
FFTGen	57	106	13.309	16	66	13.569	4.8	188%	1.02
Cmp/Dcmp(a)*†	2146	2683	15.000	50	619	18.638	23	698%	1.24
Cmp/Dcmp(wl)*‡	402	926	19.713	95	653	19.042	23.1	172%	0.97
Image(a)*†	1218	1614	14.355	95	558	25.884	82.5	385%	1.8
Image(wl)** ‡	224	919	27.449	62	536	36.778	83.4	168%	1.34
JPEG(a)*†	848	1103	17.433	45	367	32.16	111	373%	1.84
JPEG**	167	534	25.458	38	377	32.23	112.8	133%	1.27
Average	573.44	894.00	17.80	48.33	363.44	23.56	50.20	276%	1.33

Note: all designs mapped to Xilinx XC4028 (32x32 CLBs) unless otherwise specified.
*Mapped to Xilinx XC4062 (48x48 CLBs) **Mapped to Xilinx XC4085 (56x56 CLBs)
†Loop blocks optimized for minimum area. ‡Loop blocks optimized for min. wire length.

Table 1: Back-end place-and-route improvements when using our placement constraints.

- No Conn: This is similar to the "Min Conn" case, but the connectivity is not considered at all.

The columns marked with "#m" in Table 3 report the total number of modules, including RFUOPs as well as registers and multiplexers. The "con" columns report the actual number of connections in the generated VHDL file. "λ" is the minimum latency.

The bold faces show the minimum values in each row. In every case, our method results in either less number of wires or less resource duplications (due to large fanin RFUOPs). It is noteworthy that our heuristic scheduling methods almost always achieve the minimum latency. Also, the scheduling time in all the cases were less than a second.

4.3 Placement Experiments

To test our methods, we use our tool to generate VHDL files and generate a placement for the datapaths represented by the input DFG. Then using only the VHDL files, we use FPGA vendor tools to place-and-route the design. Then we run the FPGA vendor tools again, but this time we force our placement on the design.

Table 1 summarizes the results of this set of experiments. In all cases, not only our placement generates placements very quickly (50 seconds on the average), but it also causes considerable speedup in the Xilinx place-and-route phase (276% on the average).

In terms of the quality of the placement, our method followed by Xilinx place-and-route generates results that are consistently around 1.3 times worse than the place-and-route generated using Xilinx alone. It is interesting to note that the only cases that deviate from the 1.3 factor, are those optimized for minimum area. If we leave those two cases out, the average ratio would be 1.2.

The one to last column ("PD speed") is the ratio between the runtimes of Xilinx place-and-route without .ucf (second coulumn+third) and our placement followed by Xilinx place-and-route with .ucf ($5^{th} + 6^{th} + 8^{th}$ columns). It can be seen that our placement followed by Xilinx place-and-route is 276% faster than the case where Xilinx tools do the whole place-and-route with no hints in the form of user constraint files. Comparing the Xilinx placement times with and without .ucf, the former runs 10.69 times faster than the latter on the average. On the average, the routing time with .ucf is 2.2 times faster than the case with no .ucf. Ideally, by having an accurate delay estimator we can bypass back-end tools altogether during the development cycle. At the last optimization stage, the back-end tools can be run.

5. CONCLUSION

We presented a fast compilation flow for reconfigurable computing. Using "physical design aware" scheduling and a hierarchical two-stage placement algorithm, our method is able to convert C programs to placed VHDL datapaths in less than a minute. For more details on our work, see [1]. There are lots of directions to which our work can be extended. Handling nested control paths is our immediate future goal. Providing back-annotations from the back-end tools, computation precision management, more advanced solutions to the hardware/software partitioning problem are just a few more to name.

6. REFERENCES

[1] K. Bazargan. *"Designing CAD Tools for Reconfigurable Computing"*. PhD thesis, Department of Electrical and Computer Engineering, Northwestern University, 2000.

[2] K. Bazargan and M. Sarrafzadeh. "Fast Online Placement for Reconfigurable Computing Systems". *Symposium on FPGAs for Custom Computing Machines*, pp. 300–302, 1999.

[3] T. J. Callahan, P. Chong, A. DeHon, and J. Wawrzynek. "Fast Module Mapping and Placement for Datapaths in FPGAs". *Symposium on Field Programmable Gate Arrays*, 1998.

[4] A. DeHon and J. Wawrzynek. "Embedded Tutorial: Reconfigurable Computing: What, Why, and Implications for Design Automation". *Design Automation Conference*, pp. 610–615, 1999.

[5] S. Hauck, "The Roles of FPGAs in Reprogrammable Systems", *Proceedings of the IEEE*, 86(4):615–638,1998.

[6] T. Maruyama and T. Hoshino. "A C to HDL Compiler for Pipeline Processing on FPGAs". *Symposium on FPGAs for Custom Computing Machines*, 2000.

[7] M. Weinhardt. "Compilation and Pipeline Synthesis for Reconfigurable Architectures". *Reconfigurable Architectures Workshop (RAW)*, 1997.

[8] Z. A. Ye, N. Shenoy, and P. Baneijee. "A C compiler for a processor with a reconfigurable functional unit". *Symposium on Field-Programmable Gate Arrays*, pp. 95–100, 2000.