Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2013, Article ID 849545, 9 pages
http://dx.doi.org/10.1155/2013/849545

Hindawi

Research Article

A Heuristic Scheduler for Port-Constrained Floating-Point Pipelines

Zheming Jin and Jason D. Bakos

Department of Computer Science and Engineering, University of South Carolina, Columbia, SC 29208, USA

Correspondence should be addressed to Jason D. Bakos; jbakos@cse.sc.edu

Received 3 October 2012; Revised 2 January 2013; Accepted 16 January 2013

Academic Editor: Miriam Leeser

Copyright © 2013 Z. Jin and J. D. Bakos. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

We describe a heuristic scheduling approach for optimizing floating-point pipelines subject to input port constraints. The objective of our technique is to maximize functional unit reuse while minimizing the following performance metrics in the generated circuit: (1) maximum multiplexer fanin, (2) datapath fanout, (3) number of multiplexers, and (4) number of registers. For a set of systems biology markup language (SBML) benchmark expressions, we compare the resource usages given by our method to those given by a branch-and-bound enumeration of all valid schedules. Compared with the enumeration results, our heuristic requires on average 33.4% less multiplexer bits and 32.9% less register bits than the worse case, while only requiring 14% more multiplexer bits and 4.5% more register bits than the optimal case. We also compare our results against those given by the state-of-art high-level synthesis tool Xilinx AutoESL. For the most complex of our benchmark expressions, our synthesis technique requires 20% less FPGA slices than AutoESL.

1. Introduction

Over the past twenty years there has been a wide range of research in the field of high-level synthesis for pipelined datapaths [1–3]. High-level synthesis methods can often be guided by user-specified constraints that result in throughput and area tradeoffs for the generated circuits. For example, designers may wish to generate circuits with as few resources as possible at the expense of increased multiplexer fan-in that generally results in lower maximum clock frequency. On the other hand, designers may want the generated circuits to achieve maximum clock speed without regard to the resultant resource requirement.

While most HLS tools consider these types of constraints, few incorporate input port-constraints when optimizing resource usage. In other words, many HLS tools do not allow the designer to explicitly constrain the synthesis process for memory bandwidth.

As an example, consider the following arithmetic expression, the phylogenetic likelihood function (PLF), which is widely used in likelihood-based phylogenetic tools and is shown in [4]

$$\vec{L}^{(k)}_{N\in\{A,C,G,T\}}(c) = \left(\sum_{S\in\{A,C,G,T\}} P_{NS}(i)\,\vec{L}^{(i)}_{S}(c)\right) \times \left(\sum_{S\in\{A,C,G,T\}} P_{NS}(j)\,\vec{L}^{(j)}_{S}(c)\right). \quad (1)$$

Written in another way, the PLF for a single symbol can be computed using the following line of high-level code:

$$\text{out_A} = (\text{AA1}\times\text{A1} + \text{AC 1}\times\text{C1} + \text{AG1}\times\text{G1} + \text{AT1}\times\text{T1}) \times (\text{AA2}\times\text{A2} + \text{AC 2}\times\text{C2} + \text{AG2}\times\text{G2} + \text{AT2}\times\text{T2}). \quad (2)$$

This expression requires sixteen floating-point inputs. To achieve maximum throughput when implemented as a pipeline, all sixteen inputs must be read and fed into the pipeline in each clock cycle. Since there are no dependencies in this expression, this would allow one output value to be produced every clock cycle after the pipeline fills. However,

this would require that there is sufficient memory bandwidth to keep the pipeline supplied with data, requiring a sustained bandwidth of sixteen floating-point values per clock. This may require more memory bandwidth than available to the processing element. Stated another way, the expression contains 16 right-hand-side (RHS) variables and the corresponding data flow graph (DFG) would therefore contain 16 logical input ports. When this expression is synthesized onto a platform that presents less than 16 physical ports to the synthesized logic due to limitations in input bandwidth, the synthesizer is given an opportunity to optimize the synthesized logic.

Assume that our platform does not have sufficient memory bandwidth to read all inputs in a single cycle but instead is limited to reading only two input values per clock. This creates a *port constraint*. In this case, the pipeline requires eight cycles to read the inputs and is thus limited to a maximum throughput of one result every eight cycles. In other words, the pipeline has a *data introduction interval* (DII) of 8. The circuit can achieve this throughput using only two multipliers and one adder, as opposed to instancing one functional unit for each arithmetic operator in the expression. This is because one multiplier calculates a product each cycle, and an adder is required for accumulation, and a second multiplier for the outermost multiply.

Even when an HLS tool takes advantage of port constraints to minimize functional unit usage, before generating hardware it must choose from a set of valid pipelines. Unfortunately, the number of valid port constrained pipelines for a given DFG grows exponentially as a function of scheduling flexibility for each DFG operation. This scheduling flexibility depends on the depth of the functional units and the structure of the DFG, but most importantly on the degree to which the input ports are constrained (the DII). Although all of these pipelines generate hardware having the same functional behavior, there can be significant differences in the logic resources and clock speed. More specifically, the number of register bits and degree of fan-in and fan-out will vary depending on which schedule is chosen to generate the pipeline.

In this paper, we describe a heuristic for generating a port-constrained pipeline for an arbitrary acyclic DFG and compare its results—in terms of resource overhead (outside of the functional units) and routing complexity (in terms of fan-in and fan-out)—against those of an exhaustive branch-and-bound enumeration and those of a state-of-the-art commercial HLS tool. Our approach is a resource-constrained scheduling algorithm that builds upon the idea of input port priorities, but the key novelty of our approach is the order in which operations are scheduled and the analysis of how this ordering affects performance metrics such as maximum fan-in and fan-out. Note that exhaustive enumeration of all valid schedules is not generally feasible when synthesizing complex arithmetic expressions, but for this paper we used this technique to provide comparative results.

The contribution of this work is a port-constrained pipeline scheduler that employs a heuristic to reduce register and multiplexor usage. It is capable of synthesizing program loops containing no loop dependencies.

2. Previous Work

HLS algorithms for synthesizing arithmetic pipelines have been under development for 40 years. In this section we will highlight some of the seminal works that focus on resource-constrained scheduling.

Sehwa [5] was the first tool developed for synthesizing a pipelined datapath and used three types of scheduling algorithms based on time and cost constraints. Sehwa breaks pipeline synthesis into three steps: scheduling, resource allocation, and register-transfer synthesis. High-level synthesis tools developed since continue to follow these three basic steps. In addition, the authors of this tool derived a lower bound to the number of functional units in case of pipeline scheduling and implemented an allocation table to manage the binding of the functional units.

Over the years, researchers have sought to improve the scheduling algorithms used for high-level pipeline synthesis. Paulin and Knight developed the force-directed scheduling algorithm, which improved upon earlier scheduling algorithms in that it was capable of balancing the number of operations in each control step [6].

The ALPS synthesis tool formulated the scheduling problem as an integer linear programming model and was capable of performing functional pipelining using multicycle (as opposed to pipelined) functional units [7].

Park and Kyung described an iterative refinement method based on the graph-bisection problem for rescheduling some of the operations in the given schedule [8]. The method produced near optimal results in polynomial time but did not incorporate port constraints.

More recent work in high-level synthesis has focused on resource sharing and loop optimizations targeting fully pipelined functional units. Sun et al. introduced a pipeline synthesis flow which exploits resource sharing and module selection, yielding 2-3 times reduction in resources as compared to existing approaches [9]. While most work in high-level synthesis addresses latency-constrained and resource-constrained scheduling problems, few consider input port bandwidth, which limits the amount of data a circuit can receive every clock cycle.

Scrofano et al. used one floating-point core of each operation type to evaluate an expression whose inputs arrive sequentially [10]. The area-efficient architecture and algorithm reuses the same core for a series of floating-point computations which are dependent upon one another. However the algorithm is limited to generating pipelines that can receive only one input every clock cycle.

Ishimori et al. assigned priority to a fixed number of input ports based on port usage frequency, number of successors, and divider tree coverage to address the input data bandwidth limitation in their reconfigurable computing platform ReCSiP [11]. In their scheduling algorithm, an input port with higher priority will be scheduled first. They were able to reduce FPGA resources by 17.57% on average without a significant reduction in clock speed. Their proposed method of port priority assignment reduced the pipeline latency and

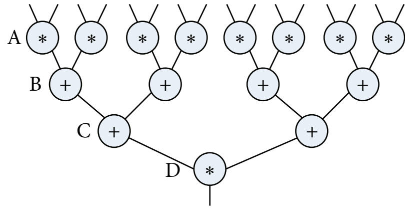


Figure 1: A data flow graph of (1).

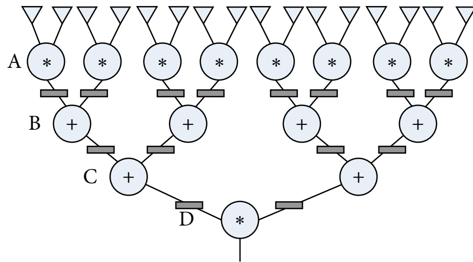


Figure 2: RTL structure of (1) with 16 input ports and maximum resource usage.

hardware costs for most arithmetic expressions. However, their scheduling algorithm did not produce minimum number of floating-point operators for each expression in the benchmark set.

3. Overview of Parallel Pipeline Scheduling

In this section we give a brief overview of parallel pipeline scheduling. We will use the PLF expression, described in Section 1, as a motivating example.

In pipeline scheduling, an arithmetic expression—referred to as a task—is partitioned into a sequence of subtasks. For the purpose of this discussion we assume that each subtask is executed in one clock cycle. Consecutive tasks are initiated at an interval of DII cycles.

Figure 1 shows an example data flow graph (DFG) of the PLF arithmetic expression (1) described in Section 1 partitioned into four subtasks labeled as A, B, C, and D. All the operations in a subtask are executed in parallel.

Figure 2 shows its RTL structure, including pipeline registers (shown as gray boxes) assuming 16 input ports. This implementation is highly parallelized, but unless all the input data can be delivered in each clock cycle it is also inefficient because there is no resource sharing.

4. Port-Constrained Pipeline Scheduling

In the following descriptions, we refer to logical ports—the inputs and output of an expression's DFG—simply as "ports." We refer to physical ports—the physical mechanism by which input data is delivered to and output data is produced from the synthesized pipeline each cycle—as "physical ports".

4.1. Lower Bound to the Number of Functional Units. In conventional scheduling it is sufficient to provide at least one functional unit of each required functional unit type to ensure that a schedule exists. In case of pipelining scheduling, we use Theorem 4 of Sehwa [5] to calculate a lower bound to the minimum number of functional units of a functional unit type using a ceiling function, that is, $R \geq \text{ceil}\,(M/\text{DII})$ where M is the number of operators of a type in the DFG and DII the data introduction interval described in Section 1. This lower bound is proven to be optimal [12].

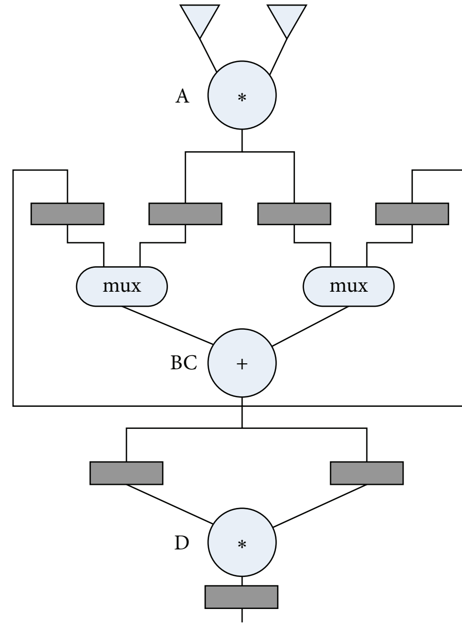


Figure 3: RTL structure of PLF using two ports and minimum functional operators.

For example, assume PLF expression is to be implemented on an FPGA whose input bandwidth is limited to accepting only two input values per clock cycle. In this situation, the circuit would have two physical ports reading two values per cycle. Since the circuit requires 16 inputs when only two can be read per cycle, the DII would therefore be 8.

A resource-constrained version of the circuit is shown in Figure 3. It has two input ports. The multiplier labeled as A is shared by all the multiplication operations at the input ports and the adder labeled as BC is shared by all the addition operations in subtasks B and C. The multiplier labeled as D at the output is not shared. Seven registers and two multiplexers (mux) are used in the port-constrained datapath implementation.

4.2. Scheduling. "as soon as possible (ASAP)" and "as late as possible (ALAP)" are two scheduling techniques widely used in datapath synthesis. ASAP repeatedly schedules the ready operations to a time slot in a manner of first-come-first-served. The start time of each operation is the minimum allowed by its dependent operations and their corresponding latencies. In contrast, ALAP scheduling algorithm provides the corresponding maximum values of the start times, computed by reversing the directions of the edges in the DFG and

Table 1: Two schedules of PLF.

Node	Op	S_1	S_2	Mobility
1	mul	3	3	[3, 9]
2	mul	5	5	[5, 11]
3	mul	7	7	[7, 12]
4	mul	9	9	[9, 13]
5	mul	11	11	[11, 14]
6	mul	13	13	[13, 15]
7	mul	15	15	[15, 16]
8	mul	17	17	17
9	add	15	8	[8, 17]
10	add	14	12	[12, 18]
11	add	19	16	[16, 19]
12	add	20	20	20
13	add	22	17	[15, 22]
14	add	23	23	23
15	mul	26	26	26

generating an ASAP schedule in reverse from the output to the inputs.

Both scheduling techniques can be extended to handle pipelined resources by allowing the scheduling of overlapping operations with different start times and no data dependencies.

Consider again the PLF arithmetic expression (1) described in Section 1. The number of input-port operations are 16 multiplications and the number of unique operation types is two (i.e., addition and multiplication). The resource conflict occurs when two different operations at different scheduled times activate the same operation at the same time. In other words, resource-sharing usage is violated when the time interval between two operations of the same type is an integer multiple of DII of a pipelined datapath. The pipelined nature of the datapath requires that if the same two operations are scheduled at time t_a and t_b, the corresponding arithmetic unit can be shared unless $t_a \equiv t_b$ (mod DII). In this case, a new operator of the conflicting type is required.

For each unscheduled DFG, there exist different schedules that meet the minimum resource constraint. Each operation in a schedule can be allocated into a time slot inside the range of schedule cycles. With pipelined ASAP and ALAP scheduling we can determine the range of scheduled cycles for every operation in the schedule. We refer to the range of scheduled cycles of an operation as the *mobility* of an operation. Note that the number of possible schedules can be estimated as the product of the mobilities of all the operators in the DFG.

Table 1 shows two valid, arbitrarily chosen schedules S_1 and S_2 of the PLF expression subject to the constraints of one input port (DII = 16) and minimum resource usage (one adder and one multiplier). Note that the example in Figure 3 assumed two input ports and therefore required an additional multiplier. For this example we assume the latency of both operations is one. For example, the mobility of node 9 is from cycle 8 to cycle 17 while the mobility of node 10 is from cycle 12 to cycle 18.

Any schedule can be obtained by scheduling one operation at a time. An operation can be moved into an early or later time slot as long as it does not violate the data dependence constraints. However, different schedules will affect the resource usage (e.g., maximum fanin, number of multiplexers and registers) of a circuit synthesized from the schedule.

Figures 4 and 5 illustrate the diagrams of datapaths synthesized from schedules S_1 and S_2 listed in Table 1.

In Figure 4, the datapath requires six multiplexers, the maximum fanin of which is 6. The number of registers allocated is 7 (excluding the output register). The output of the mux with maximum fanin goes to the left input port of the adder. Though the adder is shared by all six additions in the DFG, the delay of scheduling the add nodes in S_1 reduces the availability of the adder's inputs for sharing.

In Figure 5, there are four multiplexers. The maximum mux fanin is 2 and the number of registers is 6.

4.3. Proposed Scheduling Approach. Our proposed method is shown as Algorithm 1. The proposed algorithm only considers fixed, maximum-rate (i.e., minimum DII) pipelining and its goal is to achieve maximum hardware utilization, minimum resource requirement, and high throughput. As DII is a function of the number of physical ports, the proposed method considers minimum DII for a given number of physical ports. More resources are required when the design has more physical ports and the maximum-rate pipelining is considered. We assume that the inputs and operators are single precision floating point. We also assume that there are no control flow or loop dependencies in the DFG.

Before the scheduling operation begins, we assign port priority to all input port operators. This procedure associates a scheduling step within the data introduction interval in which each input arrives at the physical ports of the pipeline. This procedure is performed according to the port constraints as defined by the user, and the port priorities are assigned using the method of Ishimori et al. [11].

To minimize the total number of pipelined floating-point functional unit, we calculate the lower bound for each function unit type and instance only this many. Then, we schedule each operation in the DFG, inserting delay into the schedule to resolve any resource conflicts among the operations in the schedule. Our secondary objective is to reduce the overheads, in terms of data path multiplexors and pipeline registers, required for the sharing of functional units. This is accomplished by load balancing the functional units—attempting to assign each functional units an equal number of DFG operations.

Next, the algorithm schedules each operator in the DFG, one at a time, after all of the operators upon which it depends have been scheduled. For each operator to be scheduled, the algorithm searches for the earliest control step in which it can be scheduled without requiring that additional functional units be added. Once an operator is scheduled, its position within the schedule is locked in the scheduling table.

Note that our approach differs from conventional ASAP scheduling. Conventional ASAP schedules the operations

```
for each port vertex v ∈ V do
    e(v) = Port_Sched(v, P, F);
    V = V − {v};
end
while V ≠ ∅ do
  for each operation vertex v ∈ V do
    if Prev_Sched (Pred(v), E) then
      if Conflict(E, RAT(o_v), DII) then
         e(v) = e(v) + 1;
      else
         e(v) = MAX (Pred(v), E);
         UpdateRAT(v, o_v, e(v));
        e(v) = e(v) + D_v;
        V = V − {v};
      end
    end
  end
end
Conflict(E, RAT(o_v), DII)
    for each functional unit r ∈ R do
       L = RAT(o_v, r);
       while L ≠ ∅ do
        if e(v) ≡ First(L) (mod DII) then
            break;
       end
       L = Next(L);
     end
     if L = ∅ then
       return false;
     end
   end
return true;
```

Algorithm 1: Heuristic minimum-resource schedule.

in a topological order without incorporating any resource constraints. In the proposed approach, we schedule the inputs according to port priority (which affects the operator schedule) and minimize the number of functional units using the lower bound and delay insertion. A functional unit binding step is simultaneously performed that assigns the scheduled operator to a functional unit using a resource allocation table.

4.4. Scheduling Algorithm. The input to the scheduling is a data flow graph (DFG), a directed acyclic graph G (V, E), which is defined as follows. V is a set of vertices comprised of logical input ports and operators. Each vertex v is associated with one or more outgoing edges.

The scheduling will have each edge annotated with the control step in which its corresponding result of vertex v is available in the schedule. In order to track the utilization of each functional unit, we create a data structure called the resource allocation table (RAT). The number of rows in the table corresponds to the number of unique operator types in the DFG. The number of columns for each operation row is calculated using the lower bound of the number of functional units as described in Section 3. Each cell of the table is a list that records the control steps in which each resource is busy.

O is the set of all operations. Each vertex $v \in V$ represents either an input port or an operation. For each operation vertex v, its type is defined as o_v (e.g., ADD, MUL) where $o_v \in O$. Each operation o_v can be executed in D_v control steps. The set of immediate predecessors of v—the vertices that produce intermediate results used as an operand to v—is denoted by $Pred(v)$. $e(v) \in E$ represents the scheduled control step, which is the clock cycle in which the operator v produces a result in the pipeline. $R(o_v)$ represents the minimum number of functional units of type o_v.

Port_Sched(v, P, F) returns the scheduled step for an input port constrained by the number of physical ports P and the port priority F. Prev_Sched$(Pred(v), E)$ returns true if all the nodes in set $Pred$ (v) are scheduled. The function MAX$(Pred(v), E)$ returns the earliest control step that v can be scheduled. First(L) returns the first variable in list L and Next(L) returns the following variable in list L.

The first *for* loop in the algorithm initializes the outgoing edges of all the port nodes in the DFG. In the following while loop, each iteration determines the nodes that have all their predecessors scheduled. For each schedulable operation node

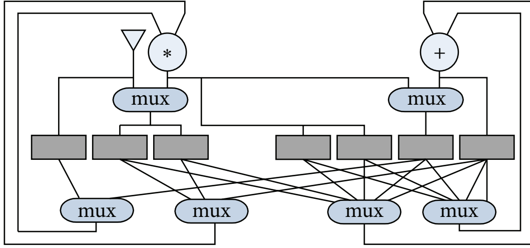


Figure 4: RTL structure of (1) from a PLF schedule S_1. It has one port and uses minimum functional units.

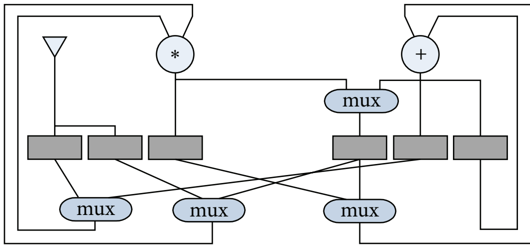


Figure 5: RTL structure of (1) from a PLF schedule S_2. It has one port and uses minimum functional units.

v, function Conflict(E, RAT(o_v), DII) looks up the row of the table RAT(o_v) to examine if the current control step can be allocated without causing resource conflict with the list of the control steps in RAT(o_v, r) where $r = \{1, 2, \ldots, R(o_v)\}$.

If there are any conflicts, the node v cannot be scheduled to the current control step and its scheduling will be delayed to the next control step. If there is no conflict, then the vertex v is assigned to the earliest possible step. UpdateRAT($v, o_v, e(v)$) updates the resource allocation for vertex v of operation o_v along with the scheduled control step. As the number of functional units of each type is fixed, UpdateRAT($v, o_v, e(v)$) selects the current least used functional unit r to achieve a balanced global binding between the operations in DFG and the functional units.

5. Experimental Results

The proposed scheduling method is implemented in the C language and the generated pipelines are described in Verilog Hardware Description Language.

Since Ishimori et al. only briefly described their modified list scheduling algorithm, we instead implemented a modified branch-and-bound method that enumerates all the feasible schedules given the constraint of minimum number of functional units [11, 13]. In order to evaluate our proposed approach we used a set of SBML ordinary differential equation (ODE) rate law expressions as benchmarks. These are a subset of the most complex expressions (in terms of number of inputs and operators) from Ishimori's work. We chose these benchmarks because they represent typical ODE expressions in that they have multiple inputs and no loop-carried dependencies. All of the experiments assume that the pipelines are generated in the most highly constrained way possible, in which only one input can be read each cycle and therefore the circuit has maximum DII given the number of inputs. We made this decision because this situation presents the largest search space to a scheduler that supports functional unit sharing, and thus the generated pipelines will have the highest variation in performance depending on the details of the scheduler implementation.

For each benchmark in Table 2, we list the number of operations of each type, DII (assuming a platform with one physical port), the number of fully pipelined Xilinx floating-point functional units [14], multiplexer (MUX) fanin and datapath fanout, the total number of MUX inputs, and the total number of registers required by the synthesized datapath. Note that MUX inputs and registers are counted in bits. The latency of the pipelined functional units is 11, 6, and 28 for adder, multiplier and divider, respectively.

Table 2 also shows the results given by synthesizing using our proposed method (P) compared with the best and worst results given by exhaustive enumeration of all minimal functional unit schedules. Since the enumeration space is the product of the mobilities of all operations, approximately one million samples are used for the expressions ordbbr, ordbur, and ppbr to avoid fully exploring their enumeration spaces that are extremely huge in practice. However, the enumeration spaces of the other expressions are fully explored.

5.1. Register Usage. For each arithmetic expression, Table 2 shows that the pipeline generated with our heuristic requires close-to-minimum (4.5% more) number of registers, while the average increase from the best case to the worst case is 56%. Registers are used to buffer intermediate results within the pipeline between functional units that produce the results and functional units that consume the results. Registers are connected serially to move the result from the cycle where it is produced to the cycle when it is needed. In other words, when there is a dependency between two operations, the steps at which the dependent operations are scheduled affects the amount of registers needed for keeping the results between the operations.

There are several methods for mapping intermediate values to registers in the generated pipeline. The left edge algorithm [15] uses the minimum number of registers but does not always lead to efficient MUX usage. A register allocation algorithm based on weighted bipartite matching (WBM) was proposed to reduce the MUX cost introduced in register allocation [16]. Chen et al. used a modified WBM method to show that register binding has great influence on the number of MUXs after scheduling and functional unit binding are fixed [17]. Later Chen and Cong presented two algorithms, register binding and port assignment for multiplexer reduction in large designs [18]. Although their proposed algorithms of multiplexer reduction were able to achieve much better results for large designs, they did not explore the use of their methods for a pipelined design.

We extended WBM to perform pipelined register binding after scheduling and functional unit bindings are finished. The ASAP nature of our scheduler schedules the dependent operations as close as possible to reduce the total number of

Table 2: Synthesis results of SMBL arithmetic expressions.

System with one input port					Max fan-in			Max fan-out			MUX inputs (bits)			Registers (bits)		
Expression	±	×	÷	DII	Min	*P*	Max	Min	*P*	Max	Min	*P*	Max	Min	*P*	Max
ucti $v = (V \times S/Km)/$ $(1 + Ka/Ac + (S/Km)(1 + Ka/Ac))$	3	2	5	5	3	4	4	3	4	5	480	480	742	726	790	1148
uuci $v = (V \times S/Km)/(1 + (S/Km)(1 + I/Ki))$	2	2	4	5	3	4	4	3	4	5	416	480	716	659	691	1081
uaii $v = (V \times S/Km)/(1 + S/Km + Ka/Ac)$	2	1	4	5	3	4	4	4	4	5	448	512	640	538	583	810
ordbbr $v = (Vf(A \times B - P \times Q/Keq))/$ $(A \times B(1 + P/KiP + KmB(A + KiA) + KmA \times B) +$ $E1)$ where $E1 = [Vf/(Vr \times Keq)] \times [KmQ \times P(1 +$ $A/KiA) + Q \times E2]$, $E2 =$ $Kmp[1 + KmA \times B/(KiA \times KmB) + P(1 + B/KiB)]$	11	16	7	14	9	12	14	7	9	14	1344	1772	2700	1771	1803	2980
ordbur $v = (Vf(A \times B - P/Keq))/$ $([A \times B + KmA \times B + KmB \times A + [Vf/(Vr \times$ $Keq)](KmP + P(1 + A/KiA)])$	6	8	4	10	6	6	8	6	6	10	928	992	1606	1474	1476	2172
ppbr $v = (Vf(A \times B - P \times Q/Keq))/$ $(A \times B + KmB \times A + KmA \times B(1 + Q/KiQ + E3))$ where $E3 = [Vf/(Vr \times Keq)][KmQ \times P(1 +$ $A/KiA) + Q(KmP + P)]$	8	12	5	13	8	8	12	6	7	13	1056	1248	2054	1246	1282	1850

registers. Furthermore, we reduced the number of registers needed at each point of intermediate result buffering based on the register binding algorithms. Assume two dependent operations o_i and o_j are scheduled at time t_a and t_b ($t_b > t_a$), respectively, then the minimum number of registers needed for buffering the intermediate results is the ceiling function of the difference of schedule time divided by DII, that is, ceil$((t_b - t_a)/\text{DII})$. That is to say, the number of registers needed is less than $t_b - t_a$.

5.2. Multiplexers Usage. In Table 2, the average increase from the best case to the worst case is 72% in the number of MUX input bits. The total number of MUX inputs required by the pipelines generated with our heuristic is, on average, 14.4% more than the best cases and 33.4% less than the worst cases.

We find that there is a relatively small variance in MUX fanin and datapath fanout but a large variance in the number of MUX inputs. Just as the minimum number of functional units is a function of DII, MUX fanin and datapath fanout are also a function of DII.

Multiplexers in a datapath can be broken down into MUXRs and MUXPs. MUXRs represent a multiplexer introduced before a register when more than one functional unit produces results and stores them into a shared register, as shown in Figure 6(a). MUXP is a multiplexer used to share a port of a functional unit when more than one register feeds data into the same port, as shown in Figure 6(b).

Table 3 compares the MUXPs and MUXRs usage of the proposed method to the best and worst results. On average, the proposed method uses 15% less MUXPs and 71% less MUXRs compared to the worst case.

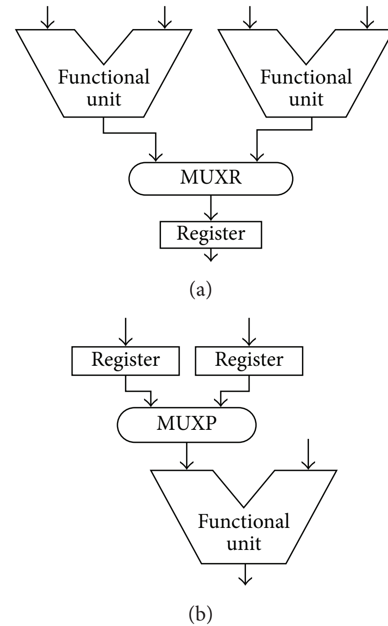


Figure 6: (a) Multiplexer introduced before a register (b) Multiplexer introduced before a functional unit port.

5.3. Fanin and Fanout. Comparing the results of our method with the best and worst cases of the branch-and-bound

Table 3: Number of MUXPs and MUXRs.

Exp.	Number of MUXPs (bits)			Number of MUXRs (bits)		
	Min	Proposed	Max	Min	Proposed	Max
ucti	352	480	512	0	64	224
uuci	288	416	448	0	64	288
uaii	288	384	384	64	128	256
ordbbr	1184	1440	2016	0	320	832
ordbur	832	928	1152	0	64	512
ppbr	1024	1120	1568	0	128	608

Table 4: Maximum fan-in and number of multiplexers given by AutoESL and the proposed.

Exp.	Max fan-in		MUX Inputs (bits)	
	AutoESL	Proposed	AutoESL	Proposed
ucti	4	4	533	480
uuci	4	4	469	480
uaii	4	4	405	512
ordbbr	14	12	1984	1772
ordbur	7	6	1104	992
ppbr	11	8	1532	1248

Table 5: Implementation results given by AutoESL and the proposed.

Exp.	AutoESL				Proposed			
	Slice	Slice regs	Slice luts	F_{max}	Slice	Slice regs	Slice luts	F_{max}
ucti	840	2525	1599	352	844	2501	1483	376
uuci	784	2429	1558	350	798	2337	1461	356
uaii	807	2459	1597	351	861	2391	1497	363
ordbbr	1417	3826	2783	270	980	2955	2349	350
ordbur	1094	3091	2129	343	910	2752	1915	351
ppbr	1197	3440	2207	250	1050	2833	2072	353

method, maximum fanin and maximum fanout using the proposed method are, on average, 20% more than the best case and 21% less than the worst case found in the exhaustive search. The operation binding using RAT gives a balanced usage of functional units. Scheduling the operations in the ASAP manner helps register binding algorithm reduce the chance of adding more registers for buffering conflicting intermediate results in a resource-constrained pipeline. The pipelined WBM takes into account the interconnection cost and further optimizes the register and multiplexer usage. For these reasons, we can assume that pipelines generated with our heuristic can be implemented with a clock speed that is close to the optimal schedule with minimal functional unit usage.

5.4. Comparison with Commercial HLS Tools. We also compare our results to those given by the commercial tool Xilinx AutoESL version 2012.1. We also evaluated three other commercial high-level synthesis tools, ImpulseC CoDeveloper [19], Synopsys Synphony C compiler [20], and Cadence's C-to-Silicon compiler [21], but none of these were suitable for comparison. ImpulseC does not incorporate knowledge of the port constraint to facilitate floating-point functional unit sharing. Synopsys Synphony C compiler incorporates functional unit sharing, but only when sharing the functional unit would result in it being used to compute a result less frequently than the pipeline's DII. However, since Synphony C does not support floating-point operations, we had to determine this using integer functional units. Cadence's C-to-Silicon compiler does not natively support floating-point operations.

AutoESL supports floating-point operations. The tool accepts, as input, a C-based design description with directives and constraints [22]. It has a built-in technology library that specifies the timing and area details of all supported Xilinx devices. We set the timing constraint to 350 MHz in AutoESL to make AutoESL generate maximum-latency Xilinx floating-point operators. The latency of floating-point addition, multiplication and division is 11, 6, and 28 respectively. AutoESL's FIFO interface is used for its interface, which allowed us to tightly control the input bandwidth. The optimization directive "set_directive_pipeline" is specified to enable AutoESL to synthesize a pipeline with DII equal to 1. We also specify the input port's data read order in the C-based loop with Ishimori's port priority. We discover this is an effective scheduling guide for AutoESL to generate a schedule with low pipeline latency. To automate generation of place-and-route and timing results of all the proposed and AutoESL's designs in Xilinx ISE 13.2, we used a Tcl-based command script that contains the same synthesis, map, place and route and timing settings. In addition, for each design obtained by AutoESL we removed the combinational logics that drive the clock-enable input of the Xilinx floating-point operators and the primary output port, ap_ready. All these settings are suitable for a fair comparison with our proposed synthesis results.

The target FPGA device is Xilinx Virtex5 LX330-FF1760-2 [23]. The datapath width of all the generated designs is 32 bit. The pipeline output rate is defined as the number of cycles between the time a task produces a result and the time the next task produces a result. As the pipeline output rate is equal to DII, it is not shown in Table 2. The resource usage is broken down into the required number of floating-point functional units of each type and FPGA resources in terms of slices, slice registers, and slice LUTs. The pipeline performance is estimated using DII, FPGA slices and design frequency in MHz. Each design is generated to have one physical port in order to evaluate the capability of resource sharing and resource usages of registers and multiplexers. The design accepts input data every clock cycle if data is ready.

Table 4 compares maximum fanin and multiplexer inputs of each synthesized design from AutoESL and our proposed approach. On average, our maximum fanin and multiplexer inputs are 8% and 3% less than AutoESL's synthesis results, respectively.

Table 5 compares FPGA results of AutoESL and ours for the benchmark expressions. For all the benchmarks, we obtain, by averaging all the results, 10% less slice registers and 8.5% less slice luts than AutoESL's results. We use, on average, 3% more slices for ucti, uuci, and uaii whereas 20%

less slices for ordbbr, ordbur, and ppbr. The designs meet the timing requirement of 350 MHz. AutoESL's results do not meet the timing constraint for ordbbr, ordbur, and ppbr though the tool takes advantage of the timing specification of the target Xilinx device in its synthesis flow. When the complexity of the expression increases, maximum resource sharing requires many control states to manage pipelined register update and the execution of floating-point operations in the RTL datapath. High fanout from the control path to the datapath and high multiplexer fanin make the circuits obtained by AutoESL unable to meet the timing.

6. Conclusion

Implementation of arithmetic expressions is a building block in many large applications. Automatic synthesis of a complex arithmetic expression with fine-granularity optimizations in area and throughput gives designers the options to meet the increasing demands of design productivity, high performance, limited memory bandwidth, and FPGA resources.

In this paper, we proposed a heuristic scheduling with port constraints. This heuristic is able to generate a datapath that achieves near-minimal register usage and low multiplexer usage. The introduction of delay into the schedule resolves the conflicts of immediate usage of floating-point units and helps achieve minimum number of functional units. A least-used resource binding in the schedule tries to achieve a balanced number of MUX inputs.

Compared to the results of executing the time-consuming branch-and-bound method, the heuristic approach gives us a fast and resource-efficient design. Compared to the results of the commercial synthesis tools, we use less resource usage and achieve the timing constraint. The bandwidth-aware and resource-constrained approach addresses the practical limitations to the platform-dependent memory bandwidth and hardware resource budget.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant no. 0844951.

References

[1] P. Coussy and A. Morawiec, *High-Level Synthesis from Algorithm to Digital Circuit*, Springer, 2008.

[2] D. Gajski, N. Dutt, and S. Lin, *High Level Synthesis: Introduction to Chip and System Design*, Kluwer Academic, 1992.

[3] P. Michel, U. Lauther, and P. Duzy, *The Synthesis Approach to Digital System Design*, Kluwer Academic, 1992.

[4] S. Zierke and J. D. Bakos, "FPGA acceleration of the phylogenetic likelihood function for Bayesian MCMC inference methods," *BMC Bioinformatics*, vol. 11, article 184, 2010.

[5] N. Park and A. C. Parker, "SEHWA: a program for synthesis of pipelines," in *Proceedings of the 23rd Design Automation Conference*, pp. 454–460, July 1986.

[6] P. G. Paulin and J. P. Knight, "Force-directed scheduling for the behavioral synthesis of ASIC's," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 8, no. 6, pp. 661–679, 1989.

[7] C. T. Hwang, J. H. Lee, and Y. C. Hsu, "A formal approach to the scheduling problem in high level synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 10, no. 4, pp. 464–475, 1991.

[8] I. C. Park and C. M. Kyung, "Fast and near optimal scheduling in automatic data path synthesis," in *Proceedings of the 28th ACM/IEEE Design Automation Conference (DAC '91)*, pp. 680–685, June 1991.

[9] W. Sun, M. J. Wirthlin, and S. Neuendorffer, "FPGA pipeline synthesis design exploration using module selection and resource sharing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, no. 2, pp. 254–265, 2007.

[10] R. Scrofano, L. Zhuo, and V. K. Prasanna, "Area-efficient evaluation of a class of arithmetic expressions using deeply pipelined floating-point cores1," in *Proceedings of the 5th International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA '05)*, pp. 119–128, June 2005.

[11] T. Ishimori, H. Yamada, Y. Shibata et al., "Pipeline scheduling with input port constraints for an FPGA-Based biochemical simulator," in *Reconfigurable Computing: Architectures, Tools and Applications*, vol. 5453 of *Lecture Notes in Computer Science*, pp. 368–373, Springer.

[12] Y. Hu, A. Ghouse, and B. S. Carlson, "Lower bounds on the iteration time and the number of resources for functional pipelined data flow graphs," in *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers & Processors (ICCD '93)*, pp. 21–24, October 1993.

[13] M. Narasimhan and J. Ramanujam, "A fast approach to computing exact solutions to the Resource-Constrained Scheduling problem," *ACM Transactions on Design Automation of Electronic Systems*, vol. 6, no. 4, pp. 490–500, 2001.

[14] *Xilinx Floating Operator Data Sheet*, Xilinx Inc, 2009.

[15] F. Kurdahi and A. Parker, "REAL: a program for register allocation," in *Proceedings of the 24th ACM/IEEE Design Automation Conference (DAC '87)*, pp. 210–215, June 1987.

[16] C. Y. Huang, Y. S. Chen, Y. L. Lin, and Y. C. Hsu, "Data path allocation based on bipartite weighted matching," in *Proceedings of the 27th ACM/IEEE Design Automation Conference*, pp. 499–504, June 1990.

[17] D. Chen, J. Gong, and Y. Fan, "Low-power high-level synthesis for FPGA architectures," in *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED '03)*, pp. 134–139, Seoul, Korea, August 2003.

[18] D. Chen and J. Cong, "Register binding and port assignment for multiplexer optimization," in *Proceedings of the ASP-DAC Asia and South Pacific Design Automation Conference*, pp. 68–73, January 2004.

[19] *Impluse CoDeveloper C-to-FPGA Tools*, Impulse Accelerated Technologies, 2012.

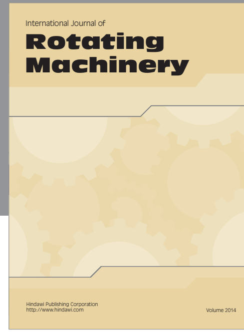
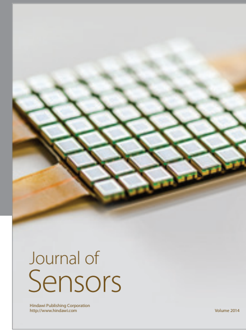
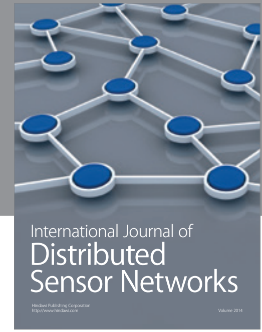
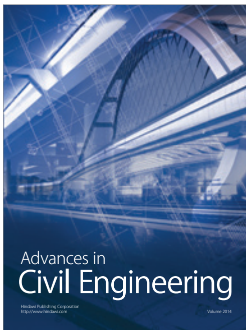
[20] *Synphony C Compiler Reference Manual*, Synopsys Inc, 2012.

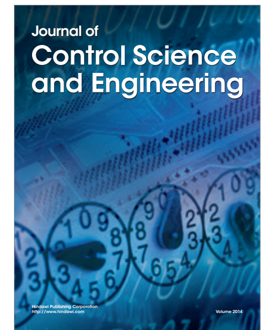
[21] *Cadence C-to-Silicon Compiler Datasheet*, Cadence Inc, 2008.

[22] *AutoESL User Guide*, Xilinx Inc, April 2012.

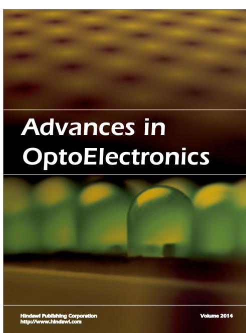
[23] *Virtex-5 FPGA User Guide*, Xilinx Inc, 2012.

Journal of
Engineering

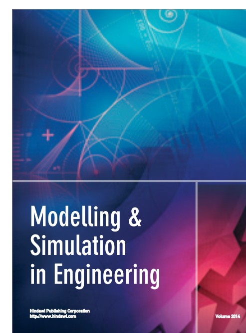
The Scientific
World Journal

International Journal of
Rotating
Machinery

Journal of
Sensors

International Journal of
Distributed
Sensor Networks

Advances in
Civil Engineering

Hindawi
Submit your manuscripts at
http://www.hindawi.com

Journal of
Control Science
and Engineering

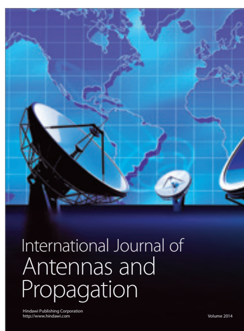
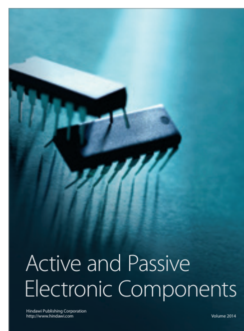
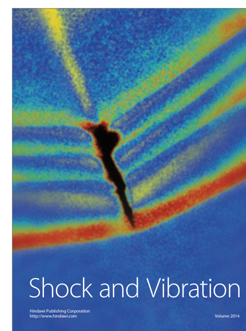
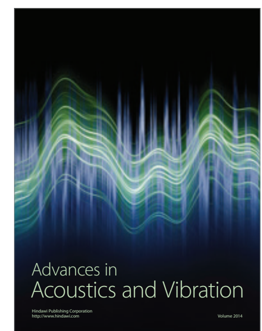
Journal of
Robotics

Journal of
Electrical and Computer
Engineering

Advances in
OptoElectronics

VLSI Design

International Journal of
Navigation and
Observation

Modelling &
Simulation
in Engineering

International Journal of
Aerospace
Engineering

International Journal of
Chemical Engineering

International Journal of
Antennas and
Propagation

Active and Passive
Electronic Components

Shock and Vibration

Advances in
Acoustics and Vibration