Balanced Scheduling and Operation Chaining in High-Level Synthesis for FPGA Designs

David C. Zaretsky[1], Gaurav Mittal[1], Robert P. Dick[2], and Prith Banerjee[1]

[1] *Department of Electrical and Computer Engineering*
University of Illinois at Chicago
851 South Morgan Street
Chicago, IL 60607-7043
{dcz, mittal, prith}@uic.edu

[2] *Department of Electrical Engineering & Computer Science*
Northwestern University
2145 N. Sheridan Road, L324
Evanston, IL 60208-3118
dickrp@eecs.northwestern.edu

Abstract

In high-level synthesis for FPGA designs, scheduling and chaining of operations for optimal performance remain challenging problems. In this paper, we present a balanced scheduling routine that uniformly distributes operations across states to reduce critical timing paths in the absence of accurate functional unit delay models. On average, results show improvements in frequency and run times for balanced scheduling over ASAP, ALAP, and force-directed scheduling. Additionally, we provide a methodology for precision-based delay modeling of operations. We present a balanced chaining routine that, given a target frequency, uses this modeling technique to reduce the number of clock cycles in the design. Results show approximately 20% improvement on average in run times when incorporating our balanced chaining routine with scheduling. Applying balanced chaining in a high-level synthesis tool allowed performance improvements between 8–29× for large, complex applications. Our method for modeling operation delays is shown to be accurate in estimating delays for operation chaining during high-level synthesis.

1. Introduction

In recent years, the sizes and complexities of designs for Field Programmable Gate Arrays (FPGAs) and other reconfigurable hardware devices have increased dramatically. As a result, the manual approach to hardware design for these systems has become cumbersome, making high-level synthesis an increasingly attractive approach for reducing the design time of complex systems. Traditionally, the high-level synthesis problem is one of transforming an abstract model of a high-level application into a set of operations for a system, in which scheduling and binding are performed to optimize the design in terms of area, cycles, frequency, and power. Generally, scheduling is performed on a control and data flow graph (CDFG), which is made of nodes representing inputs, outputs, and operations. Most high-level synthesis tools allow tradeoffs between these metrics based on estimates. At high levels of abstraction, these estimates are prone to errors, leading to inaccurate predictions of the final implementation. Poor decisions due to inaccurate high-level estimates have direct impact on all future optimizations and the resulting design.

This material is based upon work supported by the National Science Foundation under Grant No. 0609666.

Accurate estimation plays a particularly important role in scheduling operations for hardware implementation on FPGAs. The operation nodes in the CDFG are scheduled by assigning each operation to a register transfer level (RTL) state in a finite state machine using schemes such as As-Soon-As-Possible (ASAP) and As-Late-As-Possible (ALAP) scheduling. When resources are critical, other schemes such as list scheduling and force-directed scheduling are applied.

When targeting FPGAs, infinite resources are often assumed. The objective of scheduling is to minimize the number of clock cycles (states), while maximizing the frequency and parallelism in the design, even at the cost of area. While ASAP and ALAP scheduling are quite efficient for this task, they generally hamper operation parallelism by producing a non-uniform distribution of operations among the state cycles. This results in an uneven distribution of resource usage, latency, and power. Likewise, force-directed scheduling, which balances resource utilization, is inefficient in the context of FPGA designs for two reason. Firstly, operations are generally mapped to logic blocks on the FPGA, which makes the resource optimizations less effective. Secondly, most FPGA synthesis tools are timing-driven; they commonly replicate hardware in order to improve timing performance.

Operation chaining is a technique that reduces cycles in a design by allowing the result of an operation to be used in the same cycle. It is expected that scheduling without operation chaining will produce the best overall frequency at the cost of cycles, since the critical path is limited only by the single operation in the design with the largest delay. Conversely, a naïve approach to operation chaining may result in long critical paths, low frequencies, and suboptimal performance. An optimal chaining technique is one that optimizes both frequency and clock cycles simultaneously.

In this work, we present a balanced scheduling routine in which operations are uniformly distributed across states in order to optimize the *timing performance* of the FPGA design; area utilization is not considered here since unconstrained resources for FPGA designs are assumed. By distributing operations evenly among states, we are essentially partitioning the critical path more uniformly by inserting registers in strategic places. Unlike existing FPGA scheduling algorithms, our balanced scheduling technique considers all operation types equally when balancing the number of operations per state, allowing a more efficient distribution of operations among FPGA logic blocks. We

Proceedings of the 8th International Symposium on Quality Electronic Design (ISQED'07)
0-7695-2795-7/07 $20.00 © 2007 IEEE

also present a balanced chaining routine that reduces the clock cycles while balancing the critical path of the design given a target frequency. In order to accurately determine the best-fit operation chaining, we require a delay model that considers varying bit-width for each operation implemented in the target FPGA architecture. The methodology for obtaining these models is also presented.

The remainder of this paper is organized as follows. Section 2 discusses related work and our contributions. Our balanced scheduling and chaining algorithms are presented in detail in Sections 3 and 4, respectively. Section 5 reports experimental results on a set of ten benchmarks. We also show comparison results for a set of larger applications using these scheduling techniques. Finally, conclusions and future work are presented in Section 6.

2. Related Work and Contributions

Numerous algorithms for scheduling have been developed over the years by various researchers. For a given data flow graph, scheduling determines the concurrency of the resulting implementation by assigning operations in a CDFG to specific cycles, assuming either unconstrained or constrained resources. In this paper we study the use of scheduling in the context of unconstrained resources. Local greedy algorithms based on ASAP and ALAP scheduling are often used when large problem instances prevent techniques with high computational complexity. Higher-quality, but higher computational complexity approaches based on force-directed approaches have also been proposed.

Force-directed scheduling was introduced by Paulin and Knight [2] as a means of minimizing required resources under timing constraints. The algorithm uses the ASAP and ALAP times to determine the time frame for each operation, whereby a force is computed as a distribution function to determine the best schedule for the operation. The worst-case time complexity for the algorithm is cubic in the number of operations. Efficiency improvements in force-directed scheduling were shown by Verhaegh et al. [4] through incremental force calculations that reduce the complexity to quadratic in the number of operations. Paulin and Knight [2][3] have shown how force-directed scheduling can be integrated with list-scheduling, where the force calculations are used as the priority function. Although it generally produces high-quality solutions, force-directed scheduling can be too time-consuming for scheduling large problem instances. In contrast to previous force-directed techniques, our balanced scheduling approach only considers the operation-level parallelism in the design, not resource utilization. In other words, it considers all operations equally as it attempts to uniformly distribute the number of operations per cycle within the ASAP/ALAP time frame.

Kerns and Eggers [5] introduced a balanced scheduling algorithm that schedules instructions based on an estimate of the load-level parallelism in the program. The scheduler computes load instruction weights based on a measure of the number of instructions that may execute in parallel with each load instruction. The instructions are then spread out to cover the load latency. Our balanced scheduling algorithm considers the operation-level parallelism for all single and multi-cycle operations, not just load instructions. The weight of each instruction is based solely on the number of hierarchical dependencies. Surprisingly, this results in an algorithm that efficiently produces significantly better quality of results than force-directed scheduling.

Much research has been conducted on estimating delays in high-level synthesis. However, there has been very little research in using estimated delays at high levels of abstraction in the context of operation chaining during scheduling of CDFGs for FPGA designs. Nemani and Najm [9] proposed a technique for measuring delays of combinational logic circuits, but their work is limited to simple Boolean functions and does not consider arithmetic operations. Nourani and Papachristou [10] presented a method of estimating delays for RTL logic in the context of false-path detection, in which they construct a Propagation Delay Graph to compute the critical delay in the design. Srinivasan et al. [8] described a system for estimating area and delay from behavioral RTL logic descriptions using best-fit polynomial models. Their method, however, requires logic synthesis of the design into a network of simple gates. Xu and Kurdahi [11] presented an approach for estimating area and timing delays for FPGA designs based on CLB and wire modeling, given an input logic netlist. Nayak et al. [12] developed an area and delay estimator for a high-level synthesis compiler that translates MATLAB code to RTL VHDL and Verilog for FPGAs. Their method of prediction is formulated as an equation based on constant parameters to be determined experimentally for each operation. Jiang et al. [13] presented a similar approach in which accurate high-level macro-model equations are used for estimating area, delay, and power of various RTL operations for a target FPGA architecture. Experimental values were obtained for each operation during high-level synthesis with varying bit-widths and the macro-model equation for the operations was extrapolated from a best-fit curve. Our method of estimating critical delays for our balanced chaining algorithm is based on their approach.

There are many other effective approaches to optimizing design performance, such as retiming, which is generally applied after scheduling. Given a fixed number of cycles in the design, retiming relocates registers across logic gates in order to reduce the maximum register-to-register delay [14][15][16]. In contrast, the proposed balanced chaining technique is applied during scheduling to optimize the number of cycles in the design given a target frequency.

In this work, we present a balanced scheduling routine that uniformly distributes operations across states to reduce critical paths in the absence of accurate functional unit delay models. To our knowledge, this is the first scheduling algorithm to balance operation execution intervals in order to improve timing performance for high-level synthesis of FPGA designs. On average, results show improvements in frequency and run times over the most closely-related

existing techniques when used for FPGA designs. Additionally, we present a balanced chaining algorithm for use in high-level synthesis. Given a target frequency, this algorithm uses precision-based delay modeling of operations to balance combinational paths, thereby minimizing the critical path while also reducing the number of clock cycles. Experimental results using balanced chaining have shown approximately 20% increase in performance on average over other chaining routines. By incorporating this routine in a high-level synthesis tool we have observed 8–29× improvements in FPGA performance for large, complex applications over a DSP architecture. This supports our claim that it is possible to effectively chain operations based on high-level predictions of functional unit delays.

3. Balanced Scheduling

In conventional FPGA design, performance is often optimized without regard for resource requirements. The goal is to reduce the number of clock cycles and increase the parallelism and frequency of the design, even at the cost of area. Typically, a scheduling method such as ASAP or ALAP is used, resulting in an imbalance in the number of instructions scheduled per clock cycle. In ASAP scheduling, a large number of operations are executed within the first few cycles, followed by fewer operations in the later cycles. In ALAP scheduling, fewer operations are executed early on, followed by a large number of operations in the last few cycles. Both ASAP and ALAP produce schedules with the same number of clock cycles. The benefit of balanced scheduling over ASAP and ALAP is a uniform distribution of operations among all clock cycles. Balanced operation parallelism may also result in improved resource usage, latency, power, and heat dissipation characteristics. Figure 1 illustrates the ASAP, ALAP, and balanced scheduling routines. In the diagram, each operation node has a one-cycle latency. In balanced scheduling, there is an even distribution of operations among the four cycles. The other methods show imbalanced number of operations per cycle.

Balanced scheduling is implemented in two stages. In the first stage, dependency analysis is performed on each node, in which the total number of parent dependencies in the DAG hierarchy is determined. In Figure 1, the number of node dependencies for each operation is shown for balanced scheduling. The second stage uses the number of dependencies to selectively forward nodes to later cycles in order to balance operation parallelism.

Figure 2 presents our *dependency analysis* algorithm. It accepts as arguments a block, *B*, and a mapping, *D*, of each operation node to its number of dependencies. The algorithm iterates topologically through the nodes in a block, adding all unique predecessors to the node's dependency list. Note that the elements in both *DMAP[p]* and *DMAP[n]* are sorted in the same order. This allows the addition of new nodes to *DMAP[p]* to *DMAP[n]* in linear time, thereby preserving the uniqueness and order of nodes in *DMAP[n]*. The size of each node's dependency list is then assigned to the map, *D*. The algorithm has an $O(n^2)$ worst-case time complexity.

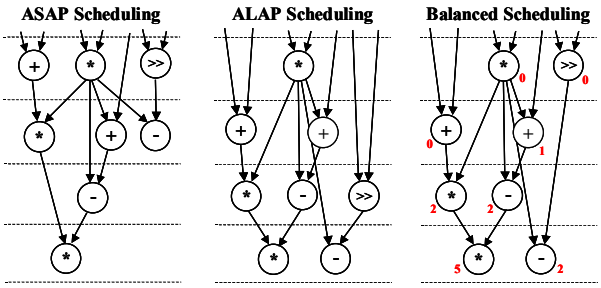

Figure 1. ASAP, ALAP, and Balanced scheduling routines.

```
  Dependency_Analysis( Block: B, Map: D )
1  topologically sort the nodes in B
2  DMAP is a mapping of nodes to their dependencies
3  for each node n in block B do
4    for each predecessor node p of n do
5      if p is an operation then
6        add each unique node in DMAP[p] to DMAP[n]
7        add p to DMAP[n]
8  for each node n in block B do
9    D[n] = DMAP[n].size()
```

Figure 2. Dependency analysis algorithm.

```
   Balanced_Scheduling( Graph: G )
 1  D is a mapping of nodes to dependency counts
 2  ASAP_Scheduling( G )
 3  for each block b in G do
 4    Dependency_Analysis( b, D )
 5    n_opers = 0
 6    for each operation node n in b do
 7      n_opers = n_opers + 1
 8      add n to T[n->GetTimeStep()]
 9    cycles = b->GetEndTime() - b->GetStartTime()
10    avg_load = n_opers / cycles
11    for each element t in T in reverse order do
12      time = t.time
13      ptime = t.time - 1
14      while T[time].size() < avg_load and
15            ptime >= b->GetStartTime() do
16        max_depend = 0
17        best_node = NULL
18        for each node n of T[ptime] do
19          bool fwd_node = true
20          wb_time = time + n->getCycles()
21          if wb_time > b->GetEndTime() then
22            fwd_node = false
23          for each successor s of n do
24            if wb_time >= s->GetTimeStep() then
25              fwd_node = false
26          if fwd_node and D[n] > max_depend
27            max_depend = D[n]
28            best_node = n
29        if best_node != NULL then
30          best_node->SetTimeStep( time )
31          T[ptime].remove( best_node )
32          add best_node to T[time]
33        else if T[time].size() < avg_load then
34          ptime = ptime - 1
35        else break
```

Figure 3. Balanced scheduling algorithm.

Our balanced scheduling routine is presented in Figure 3, in which operation nodes are uniformly distributed among cycles. The timestep for each node in the CDFG is first initialized using ASAP scheduling in line 2. In lines 3–35, it iterates over the CDFG to optimize the load balance in each basic block. Dependency analysis is performed on the

nodes in the block in line 4. Lines 5–10 count the number of operation nodes within the block, while adding each node to a map, *T*, which groups nodes by timestep. The average load balance per clock cycle is then computed. Lines 11–35 traverse *T* in reverse order, beginning with the latest cycle *(t)*. The algorithm searches for the best node to forward to that cycle by traversing each preceding cycle (*t-1*, *t-2*, *t-3*, etc.) until the earliest cycle in the block is reached. The forwarding node is selected based on two criteria: (1) the node has the largest number of dependencies and (2) forwarding the node does not violate any latency constraints, as described in lines 20–28. When a node is forwarded, *T* is updated by remapping the forwarded node to its new cycle. Once a cycle is balanced, or if it is not possible to balance it after traversing all preceding cycles, the algorithm continues on to balance the load in the next cycle in *T (t-1)*.

An analysis of the algorithm reveals that the worst-case situation occurs when forwarding is not possible for any node. This results in $O(nt)$ time complexity, where n is the number of operations in the block and t is the number of cycles after ASAP scheduling. If the DAG is very narrow, i.e., there is one node scheduled per state cycle such that $t \approx n$, the complexity resolves to $O(n^2)$. However, since most CDFGs consist mainly of tree-like structures, on average we can expect $t \approx \log n$, yielding an average time complexity of $O(n \log n)$. The worst case time complexity is therefore dominated by the $O(n^2)$ dependency analysis algorithm.

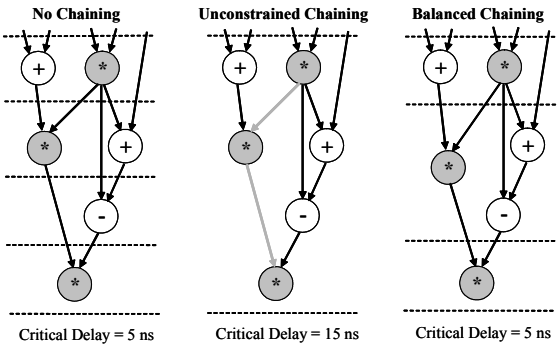

Figure 4. Comparison of chaining methods.

To illustrate balanced scheduling further, refer back to Figure 1. The CDFG is initialized with ASAP scheduling, assuming each node has a one-cycle latency. Dependency analysis is performed on each node, as shown in the figure, and the load balance is determined to be 2 (8 nodes/4 states). In a bottom-up approach, beginning at the fourth state, one node is required to balance the load. The algorithm traverses the preceding states to find a node to forward to the fourth state. Beginning in the third state, a single *subtract* operation is found, but the 1-cycle latency prevents it from being forwarded. In the second state, the *multiply* and *subtract* operators have the same number of dependencies, but only the *subtract* operation has no latency restriction. Therefore, it is selected and forwarded to the fourth state. Since the fourth state is now balanced, we continue on to the third state, which also requires a single node. Beginning with state two, the *multiply* operation has the most dependencies and no latency restriction, so it is forwarded to state three. Now the second state requires a single node to balance. Looking to the first state, all three operations have the same number of dependencies, but only the *add* and *shift* operations have no latency restriction. The *add* operation is encountered first and is forwarded to the second state. Finally, the first state is already balanced so the procedure is complete.

4. Balanced Chaining

While balanced scheduling can significantly improve the distribution of operations among cycles compared to ASAP and ALAP scheduling, it does not consider variations in the operation delays. To obtain optimal performance, it is essential to consider the operation delays when scheduling and chaining operations. In this section we present our balanced chaining algorithm, which uses a delay modeling technique to predict the critical path of a design. We also present our methodology for obtaining accurate delay models of operations during high-level synthesis. These models are then used to obtain better timing performance during scheduling by efficiently chaining RTL operations within each state of a finite state machine.

Consider the DAGs shown in Figure 4 with three different scheduling routines. We assume that multiplication operations have delays of 5 ns, while addition and subtraction operations have delays of 2 ns. In the first case, ASAP scheduling without chaining would require 4 cycles. The critical delay, or the worst-case path delay, is 5 ns, resulting in a maximum frequency of 200 MHz. It would take a total of 20 ns to complete the computations.

A naïve approach to operation chaining would schedule three multiplies within a single cycle, as shown in the second figure. The resulting implementation takes 15 ns to complete, which is faster than the unchained approach. However, after closer inspection, it is apparent that the critical path can be balanced by chaining only the add-subtract operation sequence. This results in 3 cycles and a critical delay of 5 ns, as shown in the third figure. It is interesting to note that although the balanced chaining schedule runs in the same time as the unconstrained chaining schedule, it produces a maximum allowed frequency of 200 MHz compared to 67 MHz in the latter case. This portion of the circuit will not operate in isolation: its maximum frequency will influence the performance of the *entire* design. Consequently, the higher frequency yielded by the proposed balanced chaining technique has the potential to improve the performance of the entire design.

4.1. Modeling Delays

In order to obtain an optimal scheduling of operations, it is essential to accurately model operation delays. Generally, operation delays depend on the FPGA architecture and the precision of the calculation. It is therefore necessary to obtain delay estimates for varying bit-widths of each operation for each target FPGA architecture.

Our method of delay modeling is based on previous work by Jiang et al. [13]. This work demonstrates that it is possible to accurately model the delay, area, and power of operations for ASICs with constant, linear, and quadratic equations. This process consists of two steps: acquiring the operation delays for varying bit-widths and creating high-level equations to model the delay of each operation as a function of precision.

In acquiring operation delays, one may use values from different stages during the design and synthesis process. For the sake of accuracy, values should be obtained at or below the synthesis stage. We used delay values after synthesis using *Synplicity's Synplify Pro* tool. Our goal is to obtain frequency results that well approximate those reported by the synthesis tool. Since the synthesis tool generally performs many transformations on the RTL code, it is essential to obtain accurate values for each bit-width. This is accomplished by synthesizing each arithmetic and logical operation type individually and acquiring delays for bit-widths of 2, 4, 8, 16, 32, 48, 56, and 64 bits. The delays are plotted and a best-fit curve is found to approximate the delay model as a function of precision.

Jiang et al. [13] have categorized their models into constant, linear, and quadratic equations. We have chosen to use a cubic equation in all delay models for the following reasons. We have found through experimentation that the delays of many operations in FPGAs were discontinuous linear functions of precision. These nonlinear characteristics are often due to high-level optimizations. For instance, a 2-bit add operation can be replaced with a combination of simple logic gates, resulting in reduced delay for low precision operations. Similarly, operations with higher precisions often require additional levels of logic that increase the delay. These optimizations often produce varying slopes in the model, requiring cubic expressions to attain higher-accuracy in the delay models. Piecewise-linear models may have also been used, but cubic expressions were chosen for the sake of simplicity. Incidentally, operations that exhibit constant, linear, or quadratic properties can also be modeled with cubic expressions.

In estimating the operation delays, we can expect a certain margin of error. If a consistent method is used to obtain the delays for all operations, one can expect the margin of error to be similar among all the operation delay models. Therefore, even with a margin of error, the critical path may nonetheless be identified correctly for determining the best chaining of operations.

4.2. Balanced Chaining Algorithm

In RTL VHDL and Verilog, operation chaining is accomplished by assigning the result of a computation using the blocking operator (=) rather than non-blocking operator (<=). This allows the resulting value to be used immediately instead of in the next clock cycle (state). Operation chaining is implemented on an operation node in a CDFG by assigning a cycle delay of zero. The cycle delay is used when scheduling each node in a finite state machine.

```
 Balanced_Chaining( Graph: G, double: frequency)
 1  if frequency < 1.0 then frequency = 1.0
 2  critical_delay = 1000 / frequency
 3  D is a mapping of operation nodes to delay
 4  for each block b in G do
 5    for each node n in b do
 6      D[n] = Get_Operation_Delay( n )
 7      if D[n] > critical_delay then
 8        critical_delay = D[n]
 9      if n has successors and n->getCycles()==1
10        n->setCycles( 0 )
11  for each block b in G do
12    topologically sort nodes in b
13    for each node n in b in reverse order do
14      for each predecessor node p of n do
15        D[p] = Get_Operation_Delay( p )
16        total_delay = D[p] + D[n]
17        if p->getCycles() == 0 then
18          if total_delay > critical_delay then
19            p->setCycles( 1 )
20          else if total_delay <= critical_delay
21               and total_delay > D[p] then
22            D[p] = total_delay
```

Figure 5. Balanced chaining algorithm.

Figure 5 presents our balanced chaining algorithm, which minimizes the clock cycles and critical path in the input graph, G, given an input target frequency. This is accomplished by assigning a one-cycle delay to operations in a sequence that exceeds the critical delay, thereby inserting registers in strategic places to balance the critical path. This chaining method uses a uniform cycle delay between an operation and each of its successor nodes. However, the algorithm can be easily adapted to handle varying cycle delays for each outgoing edge. The target frequency is upper-bounded by the slowest operation in the design and lower-bounded by 1 MHz, which is essentially equivalent to running unconstrained chaining. Note that even in an unconstrained chaining implementation there may exist multi-cycle operations, such as loads and stores, which limit the chaining of operations.

The algorithm begins in lines 3–10 by iterating through the nodes in the CDFG. In lines 6–8, each operation node is mapped to its predicted delay for the target architecture based on the model described above. While doing so, the critical delay is updated, which is the worst-case path delay of any sequence of chained operations in the CDFG. In lines 9–10, the CDFG is initialized to unconstrained chaining by assigning a cycle delay of zero for each operation node. Chaining is performed in lines 11–22. The nodes in the CDFG are first topologically sorted, and then traversed bottom-up. The delay of each node is recalculated based on the delay of its predecessor nodes in lines 15–16. If the combined delay of a node and its predecessor is greater than the critical delay, the predecessor node is unchained by setting its delay to one cycle in lines 18–19. Otherwise, the delay of the predecessor node is updated with the total path delay in lines 20–22. It is evident that the algorithm runs in linear time with the number of nodes in the CDFG.

5. Experimental Results

We evaluated the scheduling and chaining techniques

proposed in this paper on various benchmarks. The benchmarks were originally available in C, and compiled to assembly code using the Texas Instruments *Code Composer Studio™* software suite, targeting the C6211 DSP architecture. The assembly codes were compiled to RTL VHDL and Verilog using the FREEDOM compiler [6][7], while using the scheduling techniques outlined here to target the Xilinx Virtex II FPGA. The RTL codes were synthesized using the Synplicity *Synplify Pro* logic synthesis tool. These synthesis results were used to obtain estimated frequency results for each benchmark. The execution times on the FPGAs were measured using the number of clock cycles in the simulation and the frequency results from the synthesis process. The estimated delay models for balanced chaining were based on data gathered for this FPGA architecture using the *Synplify Pro* synthesis tool.

Table 1 shows timing results for ASAP, ALAP, force-directed, and balanced scheduling with unconstrained chaining, which produces the minimum number of cycles possible among the scheduling routines. The objective is then to maximize the frequency within this time frame. With balanced scheduling, we see that a uniform distribution of operations among the cycles improves the frequency dramatically over ASAP, ALAP, and force-directed scheduling for nearly all benchmarks. This is due to the fact that these scheduling routines may often cluster numerous time-consuming operations in sequence, resulting in long critical paths. Balanced scheduling, however, partitions the long critical paths in the design by distributing the operations evenly among the states. Variations in frequency for *laplace* and *iir* are caused by the naïve partitioning of the critical path. The minor variations in the data path can cause the back-end synthesis tool to make substantial changes in design implementations that affect the overall frequency.

Table 2 shows comparisons of the same benchmarks with no chaining, unconstrained chaining, and balanced chaining. For the latter case, a target frequency of 500 MHz was chosen to achieve the maximum frequency in the design. As expected, the number of clock cycles for balanced chaining increased over that of unconstrained chaining due to reduced chaining. However, we see dramatic increases in frequency as well as significant improvements in performance over that of an unconstrained chaining approach. The delay estimations for *ellip, laplace,* and *sobel* were hampered by extra logic inserted by the synthesis tool in the critical path. However, the difference was within a 10% margin of error, as compared to the best results. On average we observe approximately a 20% speedup over all benchmarks as compared to the best results yielded by the other chaining methods. It is interesting to note that the frequency results for balanced chaining are very close to those reported for scheduling without chaining, which one would expect to be the maximum achievable frequency for each benchmark. This supports our claim that it is indeed possible to effectively predict the critical delays of a design at an abstract level in order to optimize the chaining of operations during scheduling.

Table 1. Scheduling comparison for Xilinx Virtex II.

	Cycles	ASAP		ALAP		FORCE-DIRECT		BALANCED	
		Freq (MHz)	Time (µs)	Freq (MHz)	Time (µs)	Freq (MHz)	Time (µs)	Freq (MHz)	Time (µs)
dotprod	1204	54.8	**22.0**	68.5	**17.6**	97.5	**12.3**	111.5	**10.8**
Iir	2704	54.6	**49.5**	50.1	**54.0**	69.6	**38.9**	61.9	**43.7**
matmul	111909	103.0	**1086.5**	70.9	**1578.4**	90.4	**1237.9**	103.0	**1086.5**
Gcd	66	215.6	**0.3**	211.1	**0.3**	167.8	**0.4**	215.6	**0.3**
Diffeq	58	20.4	**2.8**	26.6	**2.2**	28.4	**2.0**	42.4	**1.4**
Ellip	53	124.9	**0.4**	146.8	**0.4**	129.9	**0.4**	166.9	**0.3**
laplace	5528	150.5	**36.7**	109.7	**50.4**	120.1	**46.0**	131.8	**41.9**
fir16tap	14948	103.7	**144.1**	71.3	**209.6**	90.5	**165.2**	103.7	**144.1**
fircmplx	2852	54.4	**52.4**	47.8	**59.7**	62.3	**45.8**	78.7	**36.2**
Sobel	18891	108.8	**173.6**	80.6	**234.4**	74.5	**253.6**	111.7	**169.1**

Table 2. Chaining comparison for Xilinx Virtex II.

	NO CHAINING			UNCONSTR. CHAINING			BALANCED CHAINING		
	Cycles	Freq (MHz)	Time (µs)	Cycles	Freq (MHz)	Time (µs)	Cycles	Freq (MHz)	Time (µs)
Dotprod	1654	145.1	**11.4**	1204	111.5	**10.8**	1304	146.2	**8.9**
Iir	6306	103.8	**60.8**	2704	58.2	**46.5**	4204	136.0	**30.9**
Matmul	171528	146.1	**1174.0**	111909	103.0	**1086.5**	120101	146.2	**821.5**
Gcd	118	187.8	**0.6**	66	215.6	**0.3**	67	215.6	**0.3**
Diffeq	156	143.1	**1.1**	58	42.4	**1.4**	94	143.3	**0.7**
Ellip	66	168.1	**0.4**	53	166.9	**0.3**	59	161.4	**0.4**
Laplace	9221	189.5	**48.7**	5528	150.5	**36.7**	6638	173.1	**38.3**
fir16tap	23386	145.1	**161.2**	14948	103.7	**144.1**	15912	145.2	**109.6**
fircmplx	3924	102.3	**38.4**	2852	78.7	**36.2**	3012	102.3	**29.4**
Sobel	33563	148.0	**226.8**	18891	111.7	**169.1**	22611	132.0	**171.3**

Table 3. Performance results in a high-level synthesis tool.

	TI C6211 DSP		Xilinx Virtex II FPGA				
	Cycles	Time (µs)	Cycles	Freq (MHz)	Time (µs)	Speedup (Cycles)	Speedup (Time)
MPEG-4 Video Decoder							
Texture_idct	275156	917.2	3584	111.9	32.0	**76.8**	**28.6**
Motion_comp	73447	244.8	4176	130.8	31.9	**17.6**	**7.7**
memory_ctrl	202294	674.3	4838	148.7	32.5	**41.8**	**20.7**
Texture_update	34888	116.3	1105	118.3	9.3	**31.6**	**12.5**
Viterbi Decoder							
Decode_acs	2285337	7617.8	71029	131.5	540.1	**32.2**	**14.1**
Flush_decoder	36743	122.5	1304	106.4	12.3	**28.2**	**10.0**
JPEG 2000 Encoder							
fdct_ifast	313024	1043.4	4850	110.6	43.9	**64.5**	**23.8**
fdct_islow	422404	1408.0	6101	116.6	52.3	**69.2**	**26.9**
mcu_huff	227206	757.4	5654	119.3	47.4	**40.2**	**16.0**

In order to verify that the proposed techniques function on large, complex problems, we compare their performance on three benchmarks, each of which has a complexity on the order of many thousands of operations. The designs were compiled from C to the TI C6211 DSP architecture, and the assembly codes were then compiled to the Xilinx Virtex II FPGA with the FREEDOM compiler using our balanced chaining routine to optimize the timing performance. Table 3 shows these results compared to that of the TI C6211 DSP with a clock frequency of 300 MHz, the maximum frequency for that architecture. Interestingly, the balanced chaining technique allowed us to obtain frequency results for all kernels in excess of 100 MHz, and speedups ranging from 8–29× over the DSP.

The *MPEG-4 decoder* showed a performance speedup of 29× for the largest block, *texture_idct*. The algorithms implemented in this block are more computationally intensive and require fewer memory accesses. The motion compensation block contains numerous memory accesses that cannot be reduced through optimizations. Hence it

shows the least gain: an 8× speedup. The *Viterbi decoder* showed the best-case speedup of 14× for the larger *decode_acs* module. This module is less memory bound and has very few conditional control structures. The second module is larger in code size and contains some control structures that cannot be eliminated. It also makes several function calls in order to calculate decision metrics. Operations cannot be moved across function boundaries by the compiler. This lowers the amount of fine grain parallelism available in the hardware design. Nevertheless, a 10× speedup has been observed for this block. The *JPEG 2000 encoder* showed a speedup of 24× for the fast DCT and 27× for the slow one, while the Huffman encoder showed a speedup of 16×.

6. Conclusions

When good delay estimations are not available for FPGAs, high-level synthesis tools often use unconstrained chaining in scheduling to reduce the number of cycles in the design. In this paper we present a balanced scheduling routine that uniformly distributed operations among states in $O(n^2)$ time, on average. This effectively breaks up large critical paths in the design and improves the frequency by distributing operations among logic blocks in FPGAs more efficiently. Results indicate that this technique performs better than ASAP, ALAP, and force-directed scheduling.

With good delay estimation and modeling methods, better-quality chaining is possible. Towards this effort, we have developed precision-based delay models to estimate operation delays in FPGAs. This technique was incorporated in our balance chaining routine. Given a target frequency, balanced chaining uses these delay models to reduce the cycles and critical path in the design by chaining operations within the given critical delay in $O(n)$ time. Results on ten benchmarks show that the proposed balanced chaining technique significantly improves frequency and run times. Furthermore, our method for modeling operation delays is shown to accurately identify the critical paths of complex designs during high-level synthesis for different FPGA architectures. Consequently, when using balanced chaining, the balanced scheduling technique is no longer essential.

Our balanced scheduling technique was tested on a set of large applications, including an MPEG-4 decoder, Viterbi decoder, and a JPEG 2000 encoder. Experimental Results indicate that the balanced chaining technique can successfully produce FPGA designs that operate at frequencies in excess of 100 MHz even for large applications. Balanced chaining have shown approximately 20% increase in performance on average over other chaining routines. By incorporating this routine in a high-level synthesis tool we have shown 8–29× improvements in FPGA performance for large, complex applications over a DSP architecture. This supports our claim that it is indeed possible to effectively predict the critical delays of a design at an abstract level in order to optimize the chaining of operations during scheduling.

In future work it would be interesting to evaluate the effects of balanced scheduling and balanced chaining on the power and thermal properties of FPGA designs with varying sampling periods. It would also be interesting to explore the use of retiming in combination with the proposed delay models during early stages of design.

7. References

[1] G. De Micheli, *Synthesis and Optimization of Digital Circuits*. McGraw Hill, 1994.

[2] P. Paulin and J. Knight, "Force-Directed Scheduling for the Behavioral Synthesis of ASICs," in *IEEE Transactions on Computer-Aided Design*, vol. 8, no. 6, June 1989.

[3] P. Paulin and J. Knight, "Scheduling and Binding Algorithms for High-Level Synthesis," in *Proceedings of the Design and Automation Conference*, Las Vegas, NV, June 1989.

[4] W. Verhaegh, P. Lippens, E. Aarts, J. Korst, A. Werf, and J. Meerbergen, "Efficiency Improvements for Force-Directed Scheduling," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, Nov. 1992.

[5] D. Kerns and S. Eggers, "Balanced Scheduling: Instruction Scheduling when Memory Latency is Uncertain," in *ACM SIGPLAN Notices,* vol. 39, issue 4, Apr. 2004.

[6] G. Mittal, D. Zaretsky, X. Tang, and P. Banerjee, "Automatic Translation of Software Binaries onto FPGAs," in *Proceedings of the 41st Annual Conference on Design Automation*, 2004.

[7] D. Zaretsky, G. Mittal, X. Tang, and P. Banerjee, "Overview of the FREEDOM Compiler for Mapping DSP Software to FPGAs," in *Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, CA, 2004.

[8] A. Srinivasan, G. Huber, and D. LaPotin, "Accurate Area and Delay Estimations from RTL Descriptions," in *IEEE Transactions on VLSI Systems*, vol. 6, no. 1, March 1998.

[9] M. Nemani and F. Najm, "Delay Estimation of VLSI Circuits from a High-Level View," in *Proceedings of the 35th Annual Design Automation Conference*, San Francisco, CA, 1998.

[10] M. Nourani and C. Papachristou, "False Path Exclusion in Delay Analysis of RTL-Based Datapath-Controller Designs," in *Proceedings of the Conference on European Design Automation*, Geneva, Switzerland, Sept. 1996.

[11] M. Xu and F. Kurdahi, "Area and Timing Estimation for Lookup Table Based FPGAs," in *Proceedings of the 1996 European Conference on Design and Test*, Mar. 1996.

[12] A. Nayak, M. Haldar, A. Choudhary, and P. Banerjee, "Accurate Area and Delay Estimators for FPGAs," in *Proceedings of the Conference on Design, Automation and Test in Europe*, Mar. 2002.

[13] T. Jiang, X. Tang, and P. Banerjee, "Macro-models for high level area and power estimation on FPGAs," in *Proceedings of the 14th ACM Great Lakes symposium on VLSI*, Boston, MA, Apr. 2004.

[14] C. E. Leiserson and J. B. Saxe, "Retiming Synchronous Circuitry," in *Algorithmica*, Vol. 6, No. 1, 1991.

[15] S. Malik, E. Sentovich, R. Brayton, and A. Sangiovanni-Vincentelli, "Retiming and Resynthesis: Optimization of Sequential Networks with Combinational Techniques," in *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, Jan. 1991.

[16] H. Zhou, V. Singhal, and A. Aziz, "How Powerful is Retiming?" in *Proceedings of the IEEE/ACM International Workshop on Logic Synthesis*, May 1998.