Loop Shifting and Compaction for the High-Level Synthesis of Designs with Complex Control Flow

Sumit Gupta[†] Nikil Dutt[†] Rajesh Gupta[§] Alexandru Nicolau[†]

Center for Embedded Computer Systems

[†]School of Information and Computer Science
University of California at Irvine
{sumitg, dutt, nicolau}@cecs.uci.edu

[§]Dept. of Computer Science and Engineering
University of California at San Diego
gupta@cs.ucsd.edu

http://www.cecs.uci.edu/∼spark

Abstract

Emerging embedded system applications in multimedia and image processing are characterized by complex control flow consisting of deeply nested conditionals and loops. We present a technique called loop shifting *that incrementally exploits loop level parallelism across iterations by shifting and compacting operations across loop iterations. Our experimental results show that loop shifting is particularly effective for the synthesis of designs with complex control especially when resource utilization is already high and/or under tight resource constraints. In situations when further loop unrolling (or initiating another iteration of the loop body) leads to a sharp increase in the longest combinational path in the circuit and the circuit area, loop shifting is able to achieve up to 20 % reduction in the input-to-output delay in the synthesized circuit. We implemented loop shifting within the SPARK parallelizing high-level synthesis framework and present results for experiments on designs derived from multimedia and image processing applications.*

1 Introduction

The computationally expensive portions of multimedia and image processing applications typically consist of arithmetic operations embedded in deeply nested loops with a complex mix of conditional (if-then-else) constructs. The focus of our work is improving the synthesis results of these codes by exposing and increasing the parallelism available in the algorithmic description [1]. In the past, speculative code motions have been used to significantly improve the performance, area, and resource utilization of the synthesized circuits by moving operations across conditional boundaries [2, 3].

The presence of nested loops in our application codes, however, limits the scope of parallelizing code motion transformations to within one loop iteration. To achieve the next level of performance improvement, we must look beyond loop iterations. In this paper, we present a loop transformation, called *loop shifting*, that moves operations from one iteration of the loop body to its previous iteration. It does this by shifting a set of operations from the beginning of the loop body to the end of the loop body; a copy of these operations is also placed in the loop head or prologue. In contrast to loop pipelining techniques that initiate a new iteration of the loop body at constant time (initiation) intervals, loop shifting shifts a set of operations one at a time, thereby, exposing just as much parallelism as can be exploited by the available resources. Parallelizing transformations can then operate on the shifted operations to further compact the loop body.

Although loop unrolling and pipelining have been proposed previously for high-level synthesis, we found that when the resource utilization is already high – because of either high instruction level parallelism in the design or as a result of loop unrolling – the control and interconnect (multiplexing) costs of further loop unrolling or loop pipelining outweigh the gains achieved in performance. This is because frequently the longest combinational path in the circuit (the critical path length) increases so much that the input to output circuit delay becomes worse (we demonstrate this through experiments in Section 5). In such a situation, we propose applying loop shifting to incrementally move code in a structured way so that operations can be compacted further without excessive increase in controller costs.

We implemented our loop shifting technique in the *SPARK* parallelizing high-level synthesis framework. This framework synthesizes a behavioral description specified in C (with restrictions on pointers, irregular jumps, and function recursion) using a set of aggressive compiler, parallelizing compiler and synthesis techniques and generates a resource bound RTL VHDL description. We demonstrate the utility of loop shifting by presenting scheduling and logic synthesis results for experiments performed on designs derived from the multimedia and image processing domains.

The rest of this paper is organized as follows: we first review previous related work. In Section 3, we present the loop shifting technique and its various aspects. We briefly describe loop unrolling in Section 3.4. We present an algorithm for loop shifting in Section 4, followed by the experimental setup and finally the results in Section 5.

2 Related Work

Loop unrolling and loop pipelining (or software pipelining) have been explored extensively in the software community for exploiting inter-iteration parallelism ([4, 5, 6, 7, 8, 9] to name a few). Loop shifting was first proposed as a part of the *resource-directed loop pipelining* (RDLP) tech-

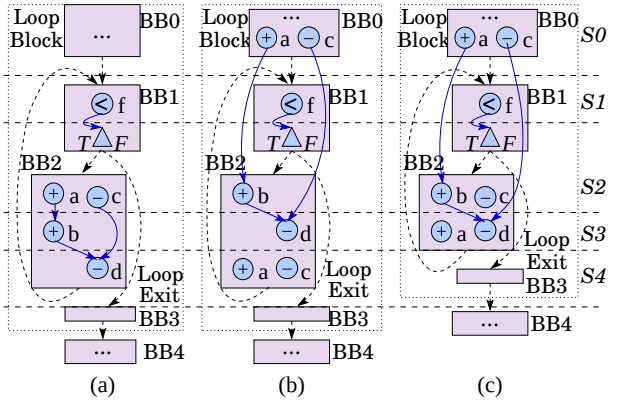

Figure 1. ***Loop Shifting: (a) An example with a loop. (b) Operations*** *a* ***and*** *c* ***are shifted to the end of loop body (basic block*** *BB*2***) and copies are inserted in the loop head (***BB0***). (c) Shorter schedule length after code compaction.***

nique [7]. RDLP first unrolls the loop several times and then attempts loop shifting and compaction.

Early work on loop pipelining in high-level synthesis focused on the innermost loops of DSP applications with no conditional constructs. These works include loop winding [10], rotation scheduling [11], percolation based synthesis [12], and loop folding [13]. More recent work on designs with conditional branches includes work on scheduling operations on the most probable path first [14], extending rotation scheduling [15], speculatively executing operations from future loop iterations [16], and extending loop folding for CDFGs [17].

Although several parallelizing code transformations and loop transformations have been proposed in the parallelizing compiler community, we found that these transformations are not directly useful for high-level synthesis (HLS). The cost functions and optimization criteria for compilers are different from those of HLS. Whereas, compilers often pursue maximum parallelization, we found that in HLS, parallelizing transformations have to be tempered by their effects on the control and area (in terms of interconnect) costs. Indeed, the very nature of transformations that are useful for HLS is different from those that are useful for compilers. In some cases, this means that we actually end up moving operations into the conditional blocks by reverse and conditional speculation [3]. Similarly, in this paper, we show that loop unrolling can lead to worse circuit delays and area; instead an incremental loop transformation technique such as loop shifting is more useful for HLS.

3 Loop Shifting

Loop shifting is a technique whereby an operation op is moved from the beginning of the loop body to the end of the loop body, along the back-edge of the loop. To preserve the correctness of the program, a copy op_c of operation op is placed in the loop head/prologue. Thus, op_c is executed before the first iteration of the loop body and the original operation op is then executed at the end of the loop body. This execution corresponds to the execution of op from the next loop iteration as per the original code.

We demonstrate loop shifting with an example in Figure 1. In this example, basic blocks *BB*1 and *BB*2 form the body

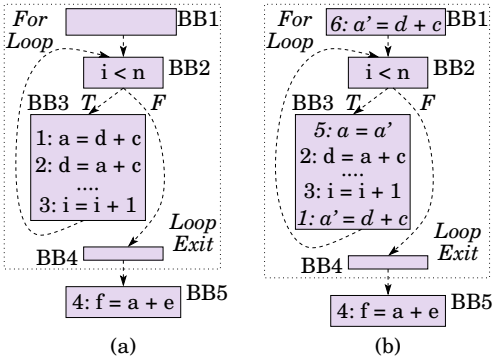

Figure 2. ***(a) An example design. (b) Copy operation,*** $a = a'$ ***is left in place of shifted op*** 1 ***to ensure code correctness.***

of a loop and *BB*0 is the loop head and *BB*3 is the loop exit or tail. Solid arrows indicate data flow and dashed arrows indicate control flow. Consider that we shift operations *a* and *c* from the loop body in the original design in Figure 1(a) to the end of the loop body (*BB*2) and copies of *a* and *c* are inserted in the loop head (*BB*0). The resultant design is shown in Figure 1(b).

We can now compact the code inside the shifted loop body using parallelizing transformations. In the shifted design, it is possible to schedule operation *a* concurrently with operation *d* and *c* concurrently with operation *b*. The resultant, compacted design is shown in Figure 1(c). The state assignments (*S*0 to *S*4) for these three designs are demarcated by dashed lines. Clearly, the design in Figure 1(c), after shifting and compaction, has a shorter schedule length than the original design in Figure 1(a).

Thus, as a result of loop shifting and compaction, the loop body executes in fewer cycles. These fewer cycles multiplied by the loop iteration count give us the reduction in execution cycles of the design. However, loop shifting is useful only when the gains in performance of the loop body is larger than the overhead of the copies of the shifted operations that are placed in the loop head.

3.1 Ensuring the Correctness of Code

Shifting an operation leads to one extra execution of the operation over the number of times it is executed in the original code. This can be understood by the shifted design shown earlier in Figure 1(c). In this design, if the loop executes for 8 iterations, then the shifted operation *a* executes 8 times inside the loop body plus once in the loop head (basic block *BB*0). In contrast, in the original design in Figure 1(a), operation *a* executes only 8 times inside the loop body.

To ensure that executing the shifted operation one extra time does not change the behavior of the program, we write the result of the shifted operation, *op*, to a new variable, *newVar* and in place of *op*, we leave a copy operation from *newVar* to the result variable of the original operation *op*.

We demonstrate this through an example in Figure 2(a). Here, the result of operation 1 in the loop body (in basic block *BB*3) is read by operation 4 after the loop. Consider that we shift operation 1 to the end of the loop body and place a copy as operation 6 in the loop head. Both these operations write to a new variable a' and a copy operation $a = a'$ is left in place of the original operation 1. This en-

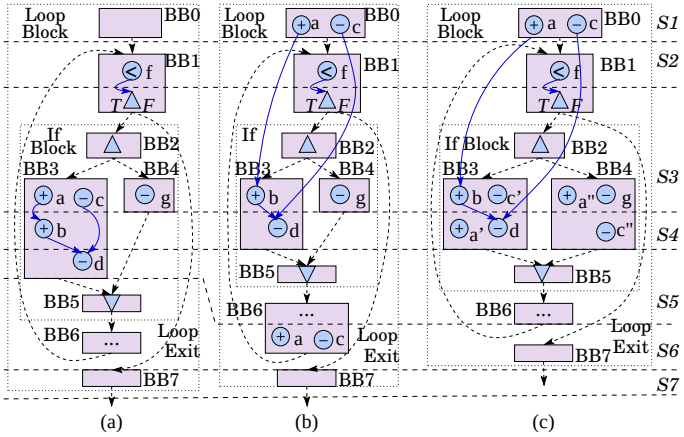

Figure 3. ***(a) A design with a if-then-else conditional block inside a loop. (b) Operations*** *a* ***and*** *c* ***are shifted from the longer conditional,*** *BB*3***. (c) Code compaction by duplicating operations*** *a* ***and*** *c* ***into both conditional branches.***

sures that operation 4 gets the correct value of *a* after loop shifting. The resultant design is shown in Figure 2(b).

We also have to maintain the inter and intra-iteration data dependencies while applying loop shifting since a shifted operation may have data dependencies across loop iterations. In the example in Figure 2(a), operation 1 reads the variable *d* that is written by operation 2. Hence, after shifting operation 1, we have to add a data dependency arc from operation 2 to shifted operation 1.

3.2 Shifting Loops with Conditional Branches

In loops with conditional constructs, operations can be shifted from within a conditional branch. Since the goal of our approach is to minimize the length of the longest path through the design, we shift operations from the branch of the conditional with the *longer schedule length*.

Consider the example in Figure 3(a). This example has an if-then-else conditional block within the body of a loop. Since the true branch (basic block *BB*3) of this if-block has a longer schedule length (of 3) than the false branch (*BB*4), we choose to shift operations from basic block *BB*3.

Hence, consider that we shift operations *a* and *c* from *BB*3, as shown in Figure 3(b). The parallelizing code transformations can now compact the shifted code by duplicating operations *a* and *c* into both branches of the if-block, as operations a' and a'' and c' and c''. The resultant design is shown in Figure 3(c). Note that, we employ a code motion called *conditional speculation* [1] for operation duplication.

3.3 Our Approach to Loop Shifting

We perform loop shifting after scheduling the loop body once. The scheduler may schedule some operations to execute concurrently in the same cycle. We term a set of concurrent operations in a basic block as a *scheduling step*.

In our approach, instead of shifting one operation at a time, we shift an entire scheduling step across loop iterations. This is because shifting only one of several concurrent operations will not eliminate the scheduling step and thus, the schedule length of the basic block (and loop body) will not decrease. In the design in Figure 3(a), we chose to shift the first scheduling step in *BB*3, i.e., both operations *a* and *c* (instead of just one of them).

```
/* Shifts one scheduling step in the loop body */
ShiftLoopBody(LoopNode)
1:  firstBB ← FirstBB(LoopNode→loopBody)
2:  stepToShift ← FindStepToShift(firstBB)
3:  BB(stepToShift) ← BB(stepToShift) - stepToShift
4:  lastBB ← LastBB(LoopNode→loopBody)
5:  lastBB ← lastBB ∪ stepToShift
6:  loopHeadBB ← LastBB(LoopNode→loopHead)
7:  loopHeadBB ← loopHeadBB ∪ Copy(stepToShift)
8:  Reschedule(LoopNode)
```

(a)

```
/* Recursive function that returns a step to shift from */
/* the basic block in the longer conditional branch */
FindStepToShift(currBB)
  Returns: The step to shift
1:  stepToShift ← FirstNonCondStep(currBB)
2:  if (stepToShift = ∅) {
3:      Find nextBB ∈ SUCCS(currBB) with the
            maximum NumSteps(nextBB)
4:      stepToShift ← FindStepToShift(nextBB)
5:  }
6:  return stepToShift
```

(b)

Figure 4. ***Loop Shifting Algorithm: shifts one scheduling step in a loop body.***

3.4 Loop Unrolling

Loop unrolling is a code transformation in which a duplicate of one or more iterations of the loop body is placed at the end of the current loop body. The loop bounds and loop index variable increment are updated as necessary. Loop unrolling is used for exposing parallelism across loop iterations and thus, enable code compaction of the unrolled loop body. However, loop unrolling can lead to code explosion; so, loops are usually unrolled one iteration at a time.

In our synthesis framework, the number of unrolls for each loop is user-directed. Our synthesis tool first unrolls the loop as specified by the designer and then schedules the design. We can thus study the effects of different unrolling factors on hardware costs and circuit performance.

4 Loop Shifting Algorithm

Our loop shifting algorithm is listed in Figure 4(a). This algorithm takes the loop node to be shifted as input and shifts one scheduling step from the beginning of the loop body to its end.

We use an intermediate representation called *Hierarchical Task Graphs* (HTGs) [18, 1] that encapsulates constructs such as loops, if-then-else blocks, et cetera in hierarchical nodes that in turn may have sub-nodes. Using this intermediate representation, we can access the sub-parts (loop head, body, and tail) of a loop by referring to *LoopNode*→*loopHead*, *LoopNode*→*loopBody* and *LoopNode*→*loopTail*. The loop head and loop tail each contain one basic block, whereas the loop body is a hierarchical node that may contain other hierarchical nodes (including if-then-else blocks and other loops). By definition, each HTG node *HtgNode* has a *Start* (or first) basic

Design	# Ifs	# Loops	# BBs	# Ops	# Resources					
					$+-$	$==$	$<<$	[]	/	$*$
*pred*1	4	2	17	123	2	2	2	2	-	-
*pred*2	11	6	45	287	2	2	2	2	-	-
tiler	11	2	35	150	3	2	2	2	1	1

Table 1. ***Characteristics of the designs used in our experiments and the resources allocated for scheduling them.***

block and a *Stop* (or last) basic block that can be obtained by $FirstBB(HtgNode)$ and $LastBB(HtgNode)$ [1].

The loop shifting algorithm starts by looking for a scheduling step to shift. To do this, it calls the function $FindStepToShift$ with the first basic block in the loop as argument. This function, listed in Figure 4(b), calls the function $FirstNonCondStep$ for each basic block *currBB*. This function returns a NULL step if *currBB* is empty (due to past shift operations) or *currBB* only has scheduling steps with conditional Boolean checks (denoted by triangles in our figures). If $FirstNonCondStep$ does not find a scheduling step, the $FindStepToShift$ function recursively traverses the basic blocks in the loop body till it finds a scheduling step in one of them. If a basic block has several successor basic blocks (branches), the algorithm traverses to the branch with the larger number of scheduling steps.

Once the $FindStepToShift$ function returns a scheduling step *stepToShift*, this step is removed from its basic block, and added to the last basic block in the loop body (lines 3 to 5 in Figure 4(a)). A copy of *stepToShift* is also added to the loop head (lines 6 and 7). We then reschedule the loop by calling the function *Reschedule*. Note that, by adding or removing a scheduling step, we mean that the operations in that step are added or removed from a basic block.

In the worst case, this algorithm may end up traversing all the basic blocks in a loop. In practice, it usually traverses not more than 2 to 3 basic blocks. Rescheduling the loop, on the other hand, can be computationally expensive. However, in practice, only the shifted operations have to be repacked in the schedule. In our experiments, we find that the run times of our synthesis tool, on a 1.6 Ghz PC running Linux, range from 1-3 usecs (user seconds) with no loop shifting, 2 to 6 usecs with loop shifting and 5 to 10 usecs with loop unrolling (see next two sections). 60 % of this time is spent in resource binding, since it is formulated as a network flow problem. In contrast, the logic synthesis tool takes between 2 to 8 *hours* for synthesizing these designs.

5 Experimental Setup and Results

We implemented the loop unrolling and shifting transformations, along with the shifting algorithm in the *SPARK* high-level synthesis framework [1]. The SPARK framework takes an input in ANSI-C (with restrictions of no pointers, irregular jumps, and recursive functions) and produces synthesizable RTL VHDL. The framework applies a range of compiler transformations (loop-invariant code motion, CSE, copy propagation, dead code elimination) and parallelizing compiler transformations (speculative code motions, dynamic CSE) [1]. So all the results presented in this section represent improvements over a design *already optimized* by these code transformations.

Transform Applied	*MPEG*-1 *pred*1		*MPEG*-1 *pred*2	
	# States	# cycles	# States	# cycles
No Unrolls	38	899	69	2127
1 Unroll	48(+26.3%)	803(-10.7%)	79(+14.5%)	2031(-4.5%)
3 Unrolls	66(+37.5%)	723(-10%)	97(+22.8%)	1951(-3.9%)
Total Reduc	+73.7 %	-19.6 %	+40.6 %	-8.3 %

Table 2. ***Scheduling results after unrolling the inner loops in*** *pred*1 ***and*** *pred*2***.***

Transformation Applied	*GIMP tiler*	
	# States	# cycles
No Unrolls	32	2534
1 Unroll	52 (+62.5 %)	2284 (-9.9 %)
4 Unrolls	97 (+86.5 %)	1834 (-19.7 %)
Total Reduction	+203.1 %	-27.6 %

Table 3. ***Scheduling Results after unrolling for*** *tiler***.**

In this section, we present results for experiments performed using designs derived from two moderately complex real-life applications: the *pred*1 and *pred*2 functions from the *Prediction* block of the MPEG-1 algorithm and the *tiler* transform from the GIMP image processing tool [19]. For the experiments presented below, we apply loop unrolling and loop shifting to inner loops of the three designs. All three designs have doubly nested loop pairs that form the main kernel of these application codes.

Table 1 lists the characteristics of the various designs in terms of the number of if-then-else blocks, for-loops, non-empty basic blocks and operations in the input description. The resources allocated to schedule these designs are also listed: $+-$ does add and subtract, $==$ is a comparator, $*$ a multiplier, / a divider, [ ] an array address decoder and $<<$ is a shifter. The multiplier ($*$) executes in 2 cycles and the divider (/) in 5 cycles. All other resources are single cycle.

5.1 Synthesis Results for Loop Unrolling

We first present the scheduling results for loop unrolling in Tables 2 and 3. The unroll factors are determined as follows: for a loop with an iteration count of N, we allow unrolling the loop by M times such that $N/(1+M)$ is an integer and less than or equal to 1. The loops that are unrolled in *pred*1, *pred*2 and *tiler* have N equal to 8, 8 and 10 respectively. Hence, for N=8, possible values of M are 1, 3 and 7, and for N=10, M can be 1, 4 or 9.

The scheduling results are in terms of the number of states in the FSM controller and the cycles on the longest path. Longest path for loops is the cycles on the longest path through the loop body multiplied by the number of loop iterations. The first row in the two tables lists the results for the case when no loop unrolling is done, the second row for one loop unroll, and the third row for 3 loop unrolls for the *pred*1 and *pred*2 designs and 4 unrolls for *tiler*. The percentage reductions of each row over the previous row are given in parentheses. The last row gives the total reduction of the third row over the first row.

The results in the second row of Tables 2 and 3 show that with one unroll, we can achieve improvements ranging from 4 % to 10 % in the cycles on the longest path for the three designs. Unrolling the loop further (three times for the *pred*1 and *pred*2 designs and four times for *tiler*) leads to a further improvement of 10, 4 and 19.7 % respectively.

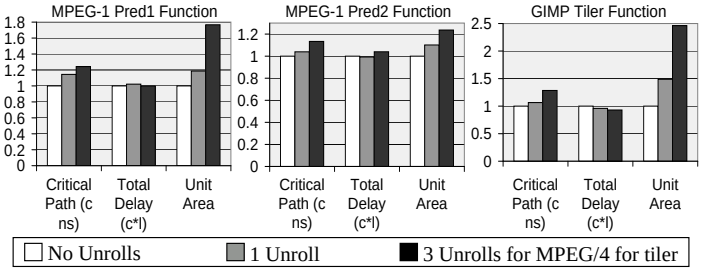

Figure 5. ***Logic synthesis results after loop unrolling.***

However, the results in Tables 2 and 3 also show that loop unrolling leads to a large increase in the size of the FSM controller (number of states). This is because when the loop body is duplicated, the number of control steps in the schedule increases, even though the number of executions of the loop body may reduce.

To study the impact on circuit area and delay, we performed logic synthesis on the RTL VHDL generated after scheduling, binding, and controller generation by the SPARK high-level synthesis tool. We used Synopsys Design Compiler with the TSMC 0.13 micron technology library. The logic synthesis results are presented in the graphs in Figure 5. The metrics mapped in these graphs are the critical path length in nanoseconds (as determined by the static timing analysis tool), the longest input to output delay or latency of the circuit (longest path cycles multiplied by the critical path length), and the unit area of the circuit (in terms of the technology library). All values are normalized by the no unrolling case. The critical path length is longest combinational path through the circuit and hence, dictates clock period or frequency.

The results in these graphs show that when the loops are unrolled, the critical path lengths increase by 10 to 25 %. This increase works against the gains achieved in cycles through the longest path. As a result, the longest input to output delay or latency through the three designs remains almost constant as the loops are unrolled. However, there is a substantial increase in area – from 22 % to up to 150 %.

The increases in critical path length and area are due to the larger controller size and more complex steering logic (multiplexers, de-multiplexers and associated control logic). As the loops are unrolled, the number of operations in the design increases. Hence, a larger number of operations are mapped to the same number of resources. This increases resource utilization, which in turn leads to an increase in the size and complexity of the steering logic.

5.2 Synthesis Results for Loop Shifting

Tables 4 and 5 list the scheduling results for the three designs as the inner loops are shifted, starting from no loop shifting (first row) to three shifts (fourth row). The percentage reductions of each row over the previous row are given in parentheses. The last row gives the total reduction of the fourth row (3 loop shifts) over the first row (no loop shifts).

The results in this table show that as the loops are shifted, the schedule length (cycles on the longest path) can sometimes increase. This happens when a set of concurrent operations is shifted from one branch of an already balanced conditional block. This means that, potentially, after shifting the scheduler is unable to compact the loop body to its size before shifting. However, in such a case, we can usually get back to the original schedule length by shifting once more; this time the scheduling step from the other branch of the conditional gets shifted.

Transform Applied	*MPEG-1 pred1*		*MPEG-1 pred2*	
	# States	# cycles	# States	# cycles
No Shifts	38	899	69	2127
1 Shift	36(-5.3 %)	771(-14%)	67(-2.9 %)	1999(-6 %)
2 Shifts	37(+2.8 %)	779(+1 %)	68(+1.5 %)	2007(+0.4 %)
3 Shifts	36(-2.7 %)	715(-8.2 %)	67(-1.5 %)	1943(-3.2 %)
Total Reduc	-5.3 %	-20.5 %	-2.9 %	-8.7 %

Table 4. ***Scheduling results after loop shifting.***

Transform Applied	*GIMP tiler*	
	# States	# cycles
No Shifts	32	2534
1 Shift	30 (-6.3 %)	2334 (-7.9 %)
2 Shifts	30 (0 %)	2244 (-3.9 %)
3 Shifts	29 (-3.3 %)	2054 (-8.5 %)
Total Reduc	-9.4 %	-18.9 %

Table 5. ***Results after shifting the inner loops. Further shifting does not improve scheduling results.***

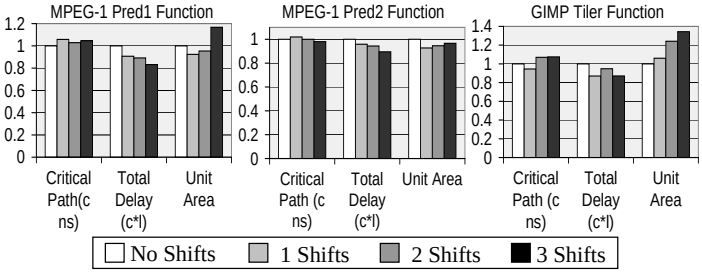

Figure 6. ***Logic synthesis results after loop shifting.***

If two consecutive shifts produce worse results, this indicates that we should stop shifting. The worse results mean that it is not possible to compact the loop body with any more shifted operations. We are currently developing more deterministic ways to determine the number of loop shifts. For now, we can experiment with different loop shifts due to low run times of our synthesis tool for fairly large designs.

From the results in Tables 4 and 5, we can see that the best scheduling results are achieved for all the designs after shifting the loop 3 times. The total reductions (last row) range from 8 to 20 % in the cycles on the longest path and 2 to 9 % in the states in the FSM controller.

These scheduling results translate over to the critical path length and area results obtained after logic synthesis. These are presented in Figure 6. The bars in these graphs correspond to no shifts, 1 shift, 2 shift and 3 shifts. These logic synthesis results show that we can achieve 5-20 % improvements in the delay through the circuit by employing loop shifting, while incurring marginal increases in circuit area compared to loop unrolling. Recall that the base case (no shifts) represents a design that is already optimized by a range of parallelizing compiler transformations [1].

5.3 Results with Higher Resource Allocation

We ran our experiments again with a higher resource allocation for the MPEG designs[1]. We used 4 adders and 3

[1]Due to a lack of space, we left out results for *tiler* in this section

Transform Applied	*MPEG*-1 *pred*1		*MPEG*-1 *pred*2	
	# States	# cycles	# States	# cycles
No Unrolls	36	771	68	1935
1 Unroll	42(+16.7%)	611(-20.8%)	72(+5.9%)	1775(-8.3%)
3 Unrolls	56(+33.3%)	563(-7.9%)	86(+19.4%)	1727(-2.7%)
Total Reduc	+55.6 %	-27 %	+26.5 %	-10.7 %

Table 6. ***Higher resource allocation: Scheduling results after unrolling the inner loops in*** *pred*1 ***and*** *pred*2***.***

Transform Applied	*MPEG*-1 *pred*1		*MPEG*-1 *pred*2	
	# States	# cycles	# States	# cycles
No Shifts	36	771	68	1935
3 Shifts	36(0 %)	659(-14.5%)	66(-2.9%)	1823(-5.8%)
5 Shifts	36(0 %)	603(-8.5%)	66(0 %)	1767(-3.1%)
7 Shifts	37(+2.8%)	555(-8 %)	67(+1.5%)	1719(-2.7%)
Total Reduc	+2.8 %	-28 %	-1.5 %	-11.2 %

Table 7. ***Higher resource allocation: Scheduling results after shifting loops in*** *pred*1 ***and*** *pred*2***.***

array decoders instead of 2 each, with other resources being the same as before (increasing the other resources did not affect scheduling results). The scheduling results for loop unrolling and shifting are listed in Tables 6 and 7 respectively. With a higher resource allocation, the improvements are larger for both loop unrolling and loop shifting. Also, loop unrolling leads to a smaller increase in controller size because the unrolled operations get compacted more easily due to the higher resource allocation. Also, we are able to do more loop shifting: 7 shifts versus the earlier 3.

The logic synthesis results for these experiments are presented in Figure 7. There results again demonstrate that loop unrolling leads to a large increase circuit area with only modest improvements in circuit delay. In contrast, loop shifting again leads to 5 to 25 % improvement in circuit delay with fairly constant circuit area.

6 Conclusions and Future Directions

We presented a loop transformation technique called *loop shifting* that incrementally exposes parallelism across loop iterations by shifting operations from one iteration of the loop body to the previous one. Experimental results on designs derived from multimedia and image applications show that loop shifting results in up to 20 % lower delays with increases in area between 0-20 %. These represent improvements over designs already optimized by our synthesis framework using a range of parallelizing code motions and code transformations. In contrast, the control and multiplexing overheads of loop unrolling undo the gains achieved in schedule lengths. The length of the critical path (longest combinational path) and thus, circuit delay increases. Circuit area increases by up to 150 %. In future work, we want to develop cost models to guide our loop transformation heuristics that estimate the impact of the loop transformations on circuit control and interconnect costs.

References

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

[2] K. Wakabayashi and H. Tanaka. Global scheduling independent of control dependencies based on condition vectors. In *Design Automation Conference*, 1992.

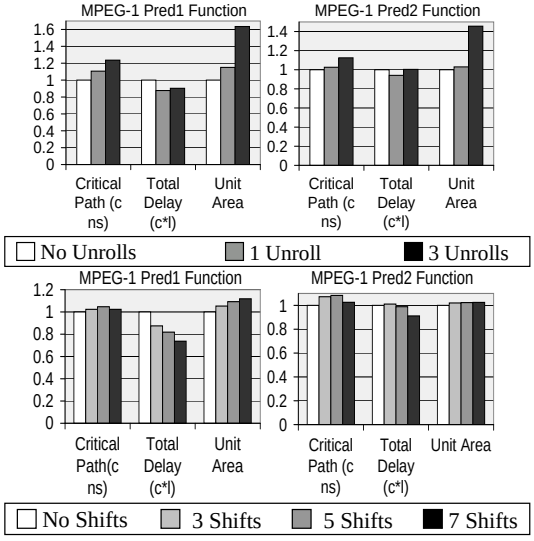

Figure 7. ***Higher resource allocation: Logic synthesis results after loop unrolling and loop shifting.***

[3] S. Gupta, et al. Conditional speculation and its effects on performance and area for high-level synthesis. In *Intl. Symp. on System Synthesis*, 2001.

[4] B. R. Rau and C. D. Glaeser. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientifi c computing. In *Annual Workshop on Microprogramming*, 1981.

[5] A. Aiken, A. Nicolau. Perfect Pipelining: A new loop parallelization technique. *Euro Symp. on Programming*, 1988.

[6] M. Lam. Software pipelining: An effective scheduling technique for VLIW machines. *PLDI*, 1988.

[7] S. Novack and A. Nicolau. An effi cient, global resource-directed approach to exploiting instruction-level parallelism. In *Conference on PACT*, 1996.

[8] A. Aiken, A. Nicolau, and S. Novack. Resource-constrained software pipelining. *IEEE Transactions on Parallel and Distributed Systems*, 6(12), December 1995.

[9] R. Jones and V. Allan. Software pipelining: A comparison and improvement. In *Proceedings of the Micro-23*, 1990.

[10] E. Girczyc. Loop winding - a data fbw approach to functional pipelining. *Intl. Symp. of Circuits and Systems*, 1987.

[11] L.-F. Chao, A. S. LaPaugh, and E. H.-M. Sha. Rotation scheduling: A loop pipelining algorithm. *DAC*, 1993.

[12] R. Potasman, J. Lis, A. Nicolau, and D. Gajski. Percolation based synthesis. In *Design Automation Conference*, 1990.

[13] G. Goossens, J. Vandewlle, and H. De Man. Loop optimization in register-transfer scheduling for DSP-systems. In *Design automation conference*, 1989.

[14] U. Holtmann and R. Ernst. Combining MBP-speculative computation and loop pipelining in high-level synthesis. In *European Design and Test Conference*, 1995.

[15] T. Z. Yu, E. H.-M. Sha, N. Passos, and R. D.-C. Ju. Algorithms and hardware support for branch anticipation,. In *Great Lakes Symposium on VLSI*, 1997.

[16] G. Lakshminarayana, A. Raghunathan, and N.K. Jha. Incorporating speculative execution into scheduling of control-fbw intensive behavioral descriptions. *DAC*, 1998.

[17] I. Radivojevic and F. Brewer. Analysis of conditional resource sharing using a guard-based control representation. In *International Conference on Computer Design*, 1995.

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

[19] GNU Image Manipulation Program. `http://www.gimp.org`.