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

Sumit Gupta[‡] Nikil Dutt[‡] Rajesh Gupta[§] Alex Nicolau[‡]

CECS
Technical Report #03-14
April 2003

Center for Embedded Computer Systems

[‡]Dept. 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. Effective hardware generation using high-level synthesis tools for these applications requires the ability to move code across condition and loop boundaries and exploit the algorithmic parallelism. Traditional loop transformations such as loop unrolling and loop pipelining have been shown to be effective only in the presence of substantive resource allocation. In case of high level synthesis, it has been shown earlier that straightforward exposition of maximum parallelism does not yield the highest performance designs, due to the control and multiplexing overheads. In this paper, we present a technique that implicitly and incrementally exploits loop level parallelism across iterations by shifting and compacting operations across loop iterations. We demonstrate the effectiveness of this technique, even under tight resource constraints, in terms of circuit performance, controller size and total size of the design. We have implemented loop shifting within a parallelizing high-level synthesis system, Spark. *We achieve improvements of up to 20 % in input-to-output delay in the synthesized circuit for experiments on designs derived from two moderately complex industrial-strength applications, MPEG-1 and the GIMP image processing tool.*

Contents

List of Figures

1 Introduction

The computationally expensive portions of multimedia and image processing applications typically consist of arithmetic operations embedded in deeply nested loops. Furthermore, these codes often contain a considerable amount of conditional (if-then-else) constructs that guide how the data is operated upon. The focus of our work is improving the synthesis results of these codes by exposing and increasing the parallelism available in the algorithmic description.

In the past, speculative code motions have been used to expose parallelism by moving operations across basic block[1] boundaries [1, 2, 3, 4, 5, 6]. These speculative code motions achieve significant improvements in performance, area and resource utilization of the synthesized circuits.

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 one such loop transformation, called *loop shifting*, that moves operations from one iteration of the loop body to the previous iteration of the loop body. 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.

Loop transformations that unroll and pipeline loops have long been recognized in the software community as key to exploiting much larger amounts of parallelism in program codes than is possible by looking at just a single iteration of the loop body [7, 8, 9, 10]. In contrast, the use of loop transformations for high-level synthesis have been few and the work has focused mostly on data flow graphs [11, 12, 13, 14]. Furthermore, these works have not analyzed the hardware-performance trade-offs of these transformations.

Straightforward application of compiler transformations does not work for high-level synthesis. This is because the increases in control, interconnect (multiplexing) and area costs are unacceptably large – particularly for designs with complex nested control flow – and are often accompanied by a corresponding increase in the critical path lengths in the synthesized circuits. Indeed, it has been shown earlier that exposing maximum parallelism does not yield the best results, and parallelizing transformations must be applied judiciously – sometimes even moving operations a non-parallelizing way (e.g., reverse [4] and conditional speculation in [5]) in order to gain improvement in overall synthesis results. Along the same lines, loop shifting – unlike loop unrolling – attempts

[1] A *basic block* is a sequence of statements in the input description that have no control flow between them.

incremental code movements in a structured way that leads to greater opportunities for compaction of operations and resource sharing without excessive increase in controller costs.

We have implemented our loop shifting technique in a C-based high-level synthesis framework. This framework synthesizes a behavioral description specified in C 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 two industrial strength designs derived from the multimedia and image processing domains.

The contributions of this paper include: (a) a novel loop transformation – loop shifting – that exposes parallelism incrementally between loop iterations, (b) an analysis of the control, interconnect and area costs of loop unrolling vis-a-vis loop shifting, and (c) an algorithm for loop shifting in designs with nested conditionals and loops.

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 in Section 4 demonstrate loop shifting in designs with conditional branches. Next, we present our approach to loop shifting, followed by an algorithm for loop shifting in Section 6. We then briefly overview loop unrolling in Section 7, followed by the experimental setup and results in Section 8. We conclude the paper with a discussion.

2 Related Work

Loop unrolling and loop pipelining (or software pipelining) have been shown to be effective techniques for exploiting parallelism across loop iterations in the parallelizing compiler community [7, 8, 9, 10, 15, 16]. Modulo scheduling and its variants [7, 9] create a schedule of one iteration of the loop body such that repetition of this iteration after a regular initiation period does not violate any resource constraints and inter- and intra-iteration dependencies. The Resource-constrained software pipelining [15] and Perfect pipelining [8] techniques iteratively unroll and schedule the loop body until a repeating pattern of operations emerges. A new pipelined loop body is then created using this repeating pattern. Loop shifting was first proposed as a part of the *resource-directed loop pipelining* (RDLP) technique [10]. 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. Loop winding [11] partitions the data flow graph (DFG) of the loop body into pieces that then form repeating pipeline stages. Rotation scheduling [14] moves or "rotates" operations in a DFG from one iteration of the loop body to the next. Percolation based synthesis [13] applies the perfect pipelining approach to high-level synthesis. Cathedral-II [12] applies loop folding to overlap successive iterations of a loop body in a data flow graph.

Holtmann and Ernst [17] apply loop pipelining to designs with conditional branches. They schedule operations on the most probable path through the loop body by deferring operations on other paths. Compensation code is executed for an incorrect prediction. Yu et al. [18] extend rotation scheduling for control-data flow graphs (CDFGs); however, a *branch anticipation* controller is required to store and propagate branch control signals across loop iterations. Lakshminarayana et al. [3] describe how operations from successive loop iterations are speculatively executed. Radivojevic and Brewer [19] extend the loop folding technique for CDFGs. However, these last two approaches do not discuss how mis-predicted speculative execution is corrected.

Most previous works discuss the effectiveness of their loop pipelining techniques only in terms of schedule lengths. These approaches do not consider the hardware overhead for speculative execution of operations from conditional branches and future loop iterations. We find that for designs with complex control flow, the control, interconnect (multiplexing) and area overheads can adversely affect or even *undo* the gains achieved in schedule lengths.

Also, equally important is the fact that no consideration is given for the overheads required to compensate for mis-predicted speculative execution of operations. This can be done by either executing compensation code or by committing the results of speculated operations only after their original conditions have been evaluated. We adopt the second approach in our scheduler (see Section 3.1).

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 of the operation, op_c, is placed in the loop head/prologue. This ensures that on the first iteration of the loop body, the operation op_c is executed as part of the loop head. The original operation op is then executed at the end of the loop body; however, this execution corresponds to the execution of op in 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 of a loop, *BB*0 the loop head and *BB*3 is the loop exit. Solid arrows indicate data flow and dashed arrows indicate control flow. Consider that we shift operation *a* from the loop body in the original design in Figure 1(a) to the end of the loop body (*BB*2) and a copy of *a* is 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*; 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;

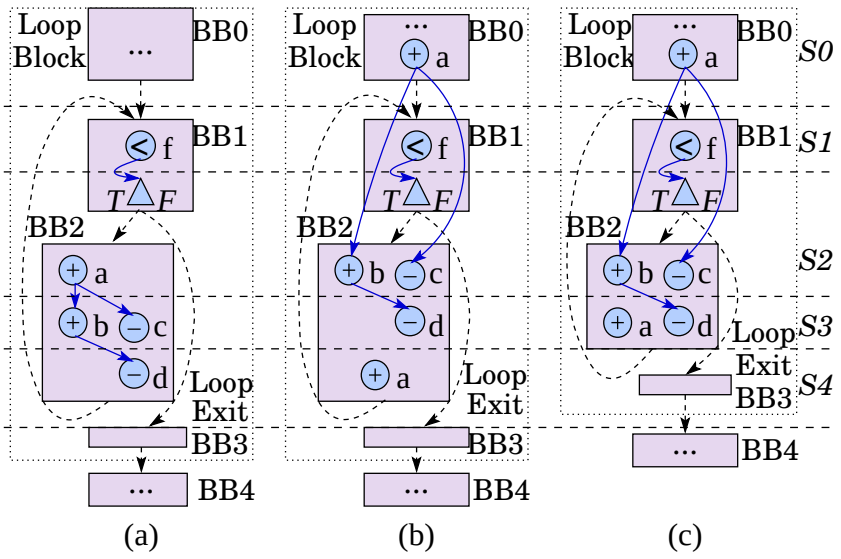

Figure 1. ***(a) An example with a loop. (b) Operation*** *a* ***is shifted to the end of loop body (basic block*** *BB*2***) and a copy is inserted in the loop head (basic block*** *BB*0***). (c) Shorter schedule length after code compaction.***

clearly, the design in Figure 1(c), after shifting and compaction, has a shorter schedule length than the original design in Figure 1(a).

This example demonstrates that loop shifting can lead to improved code compaction and parallelization of the loop body. However, since copies of the shifted operations are placed in the loop head, there is an overhead of this technique. Hence, loop shifting can be applied if it enables a shorter schedule length through the loop body and if the loop body executes enough times that the gains in performance of the loop body are larger than the overhead of duplicated operation in the loop head. As we will see in the results section, this is always true for the class of applications we target (multimedia and image processing).

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*.

This is demonstrated through an example in Figure 2(a). In this example, the result of operation 1 in the loop

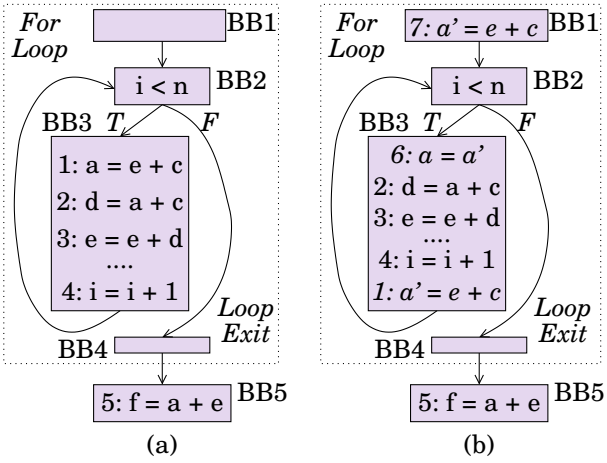

Figure 2. ***(a) An example design. (b) Copy operation,*** $a = a'$ ***is left in place of the shifted operation*** 1**.**

body (basic block *BB*3) is read by an operation after the loop body, namely, operation 5. Consider that we shift operation 1 to the end of the loop body and place a copy as operation 7 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 ensures that operation 5 gets the correct value of a in the design after loop shifting. The resultant design is shown in Figure 2(b).

A shifted operation may, also, have inter-iteration data dependencies, i.e., data dependencies across loop iterations. In the example in Figure 2(a), operation 1 reads the variable e that is written by operation 3. Hence, after shifting operation 1, we have to update the data flow graph and add a data dependency arc from operation 3 to operation 1. This is because the updated data flow graph is used by the parallelizing techniques to make decisions on code motions. Hence, we also have to maintain the inter and intra-iteration data dependencies when loop shifting is applied.

4 Shifting Loops with Conditional Branches

In loops with conditional constructs, operations may have to 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 a 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) that the false branch (*BB*4), we choose to shift an operation from basic block *BB*3.

So consider that we shift operation a from *BB*3, as shown in Figure 3(b). The parallelizing code transformations can now compact the code by duplicating operation a into both branches of the if block, as operations a' and a''.

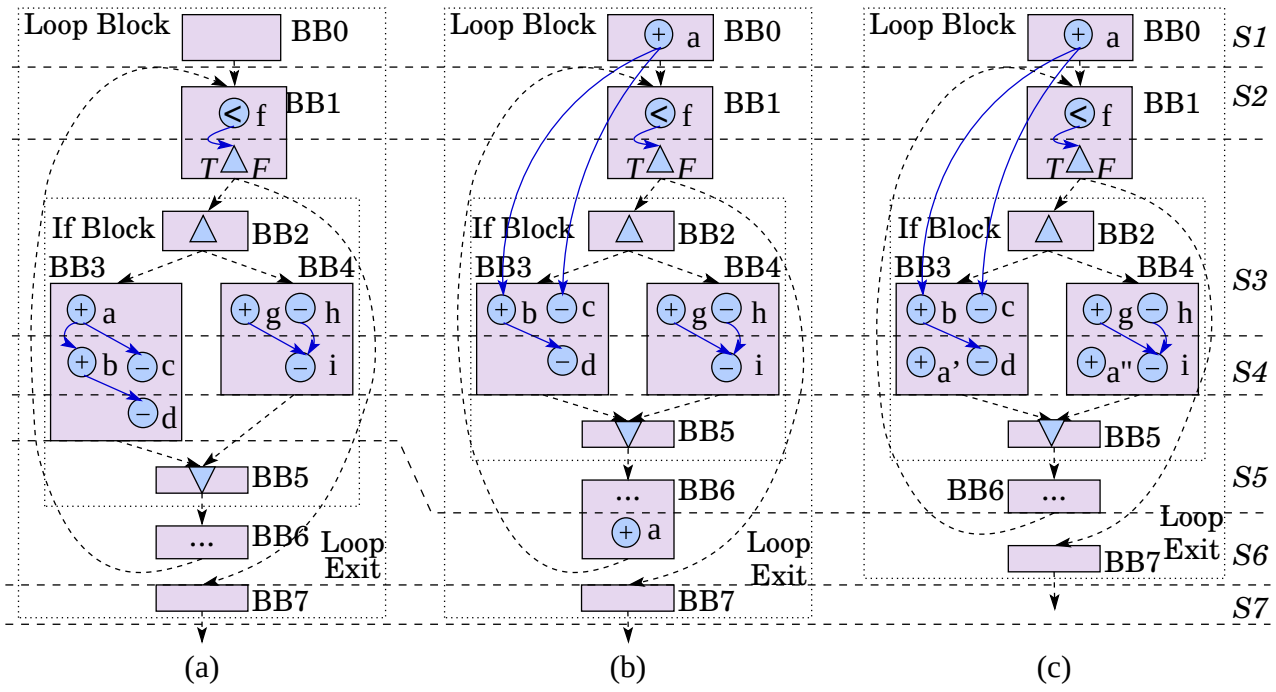

Figure 3. ***(a) A design with a if-then-else conditional block inside a loop. (b) Operation*** *a* ***is shifted from the longer conditional,*** *BB*3***. (c) Code compaction by duplicating operation*** *a* ***into both conditional branches.***

The resultant design is shown in Figure 3(c). Operation duplication into the branches of a conditional is done by a code motion transformation called *conditional speculation* [5].

Note that, when shifting operations out of conditional branches, we have to store the result of the shifted operation in a new variable. The result is committed to the original result variable of the operation only within the conditional branch. Hence, we insert a copy operation in the same manner as demonstrated earlier in Section 3.1, while discussing techniques to ensure correctness of code.

5 Our Approach to Loop Shifting

Instead of shifting one operation at a time, we shift a set of concurrent operations together. We begin by scheduling the loop using a list scheduling heuristic that employs parallelizing compiler transformations [6]. The scheduled design has operations within basic blocks that have been scheduled to execute concurrently (in the same cycle). We term a set of concurrent operations in a basic block as a *scheduling step*. For example, in the design in Figure 3(a), operation *a* executes in the first scheduling step of *BB*3, operations *b* and *c* execute concurrently in the second scheduling step and so on. Similarly, operations *g* and *h* execute concurrently in the first scheduling step of *BB*4.

The operations in a scheduling step execute in the same clock cycle. Our state assignment algorithm assigns the

same state to scheduling steps in mutually exclusive control paths that execute in the same clock cycle. Hence, as shown in Figure 3(a), the first scheduling steps of *BB*3 and *BB*4 are both assigned the state *S*3. The scheduling step that actually executes depends on the conditional check based on the evaluation of the comparison operation f. The schedule length through a conditional branch is, thus, determined by the number of scheduling steps in the basic blocks that comprise that branch. Again, in the example in Figure 3(a), basic block *BB*3 has 3 scheduling steps and *BB*4 has 2. Hence, we choose to shift an operation from the longer conditional branch (*BB*3), as described in the previous section.

In our approach, however, we shift entire scheduling steps across loop iterations instead of single operations. 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 will not decrease. In the design in Figure 3(a), we chose to shift the first scheduling step in *BB*3; in this case, the step had only operation *a*.

6 Loop Shifting Algorithm

Our loop shifting algorithm is given in Figure 4. This algorithm takes as input the loop node to be shifted. We use an intermediate representation called *Hierarchical Task Graphs* (HTGs) [20, 6]. Constructs such as loops, if-then-else blocks, et cetera are encapsulated in hierarchical nodes that in turn may have sub-nodes. For example, a loop node consists of three parts: a loop head, a loop body and a loop tail or exit. 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). An example of the HTG representation of a loop is given in Figure 5(a). In this figure, basic block *BB*1 is the loop head, *BB*2 and *BB*3 comprise the loop body and *BB*4 is the loop tail.

The sub-parts of a loop can be accessed by referring to $LoopNode \rightarrow loopHead$, $LoopNode \rightarrow loopBody$ and $LoopNode \rightarrow loopTail$. By definition, each HTG node, *HtgNode*, has a *Start* (or first) basic block and a *Stop* (or last) basic block; these are obtained by FirstBB(*HtgNode*) and LastBB(*HtgNode*) [6].

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 the argument. This function, also listed in Figure 4, recursively traverses the basic blocks in the loop body till it finds a scheduling step in one of them (basic blocks may be empty because of past shift operations). If a basic block has several successor basic blocks, the algorithm traverses to the basic block 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. A copy of $stepToShift$ is also added to the loop head. We then reschedule the loop by calling the function *Reschedule*. Note that, by adding or removing a scheduling

/* Shifts one scheduling step in the loop body */

ShiftLoopBody(LoopNode)

1: $firstBB$ = FirstBB($LoopNode \rightarrow loopBody$)

2: $stepToShift$ = FindStepToShift($firstBB$)

3: BB($stepToShift$) $\leftarrow$ BB($stepToShift$) $\setminus$ $stepToShift$

4: $lastBB$ = LastBB($LoopNode \rightarrow loopBody$)

5: $lastBB \leftarrow lastBB \cup stepToShift$

6: $loopHeadBB$ = LastBB($LoopNode \rightarrow loopHead$)

7: $loopHeadBB \leftarrow loopHeadBB \cup$ Copy($stepToShift$)

8: Reschedule($LoopNode$)

/* 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: $stepToShift$ = FirstStep($currBB$)

2: **if** ($stepToShift = \emptyset$) {

3: Find $nextBB \in SUCCS(currBB)$ with the maximum NumSteps($nextBB$)

4: $stepToShift$ = FindStepToShift($nextBB$)

5: }

6: **return** $stepToShift$

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

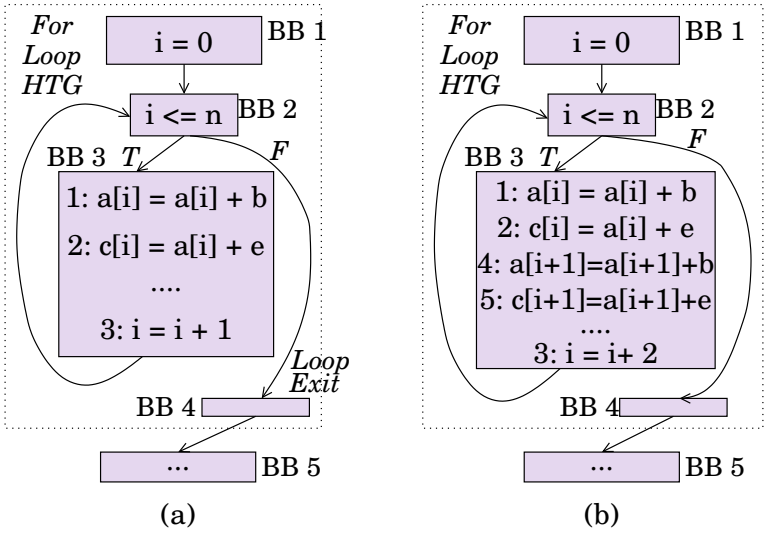

Figure 5. ***(a) The HTG of an example with a loop and some operations. (b) The loop is unrolled once***

step, we mean that the operations in that step are added or removed from a basic block. Also, note that, loop shifting is performed implicitly without unrolling the loop.

In the worst case, this algorithm may end up traversing all the basic blocks in a loop. In reality, it usually traverses not more 2-3 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, 5 to 16 usecs with loop shifting and 5 to 26 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 16 *hours* for synthesizing these designs.

7 Loop Unrolling

Loop unrolling is the process of placing a duplicate of one or more iterations of the loop body 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 code compaction across loop iterations. However, loop unrolling can lead to code explosion; so, loops are usually unrolled one iteration at a time.

Consider the example in Figure 5(a). The resulting design after the loop is unrolled once is shown in Figure 5(b). This unrolled loop exposes the inherent parallelism among the operations in the loop body – the operations 1 and 4 can be executed concurrently, followed by the concurrent execution of operations 2 and 5 (not shown in this figure). Without loop unrolling, the two iterations of the loop body would have executed sequentially.

In the our synthesis framework, the amount of loop unrolling for each loop is currently user-directed. The

Design	#Ifs	#Lps	#BBs	#Ops	# Resources					
					$+-$	$==$	$<<$	[]	/	$*$
*pred*1	4	2	17	123	3	2	4	2	-	-
*pred*2	11	6	45	287	5	2	3	2	-	-
tiler	11	2	35	150	4	4	3	2	1	1

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

tool first unrolls the loop as specified by the designer and then schedules the design. We can, thus, study the trade-offs between hardware and performance of different amounts of loop unrolling. Full loop unrolling leads to an exponential increase in scheduling times (as is the case for commercial high-level synthesis tools) due to the increase in the number of operations in the design graph.

8 Experimental Setup and Results

We implemented the loop unrolling and shifting transformations, along with the shifting algorithm in a high-level synthesis tool, called *Spark*. *Spark* takes an input in ANSI-C and produces synthesizable RTL VHDL [6]. In this section, we present results for experiments performed using two large and moderately complex real-life applications. These designs consist of the *pred*1 and *pred*2 functions from the *Prediction* block of the MPEG-1 algorithm and *tiler* transform from the GIMP image processing tool [21]. For the experiments presented below, we apply loop unrolling and loop shifting to the "j", "p" and "c" loops in the *pred*1, *pred*2 and *tiler* designs respectively. These loops are the inner loops of 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. All these designs have doubly nested loops. We also list the the type and quantity of each resource allocated to schedule these designs. The resources are; $+-$ 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 4 cycles. All other resources are single cycle.

8.1 Scheduling Results for Loop Unrolling

Table 2 lists the scheduling results after loop unrolling for the 3 designs. These results are in terms of the number of states in the FSM controller and the cycles on the longest path. Longest path length for loops is calculated by multiplying the cycles on the longest path through the loop body by the number of loop iterations. The first row lists the results for the case when no loop unrolling is done on the designs, the second row for when the loop is

Transformation Applied	*MPEG*-1 *pred*1		*MPEG*-1 *pred*2		*GIMP tiler*	
	# States	# cycles	# States	# cycles	# States	# cycles
No Unrolls	34	833	63	2060	29	2334
1 Unroll	40(+17.6%)	641(-23.1%)	67(+6.3%)	1804(-12.4%)	47 (+62.1 %)	2084 (-10.7 %)
3/4[2]Unrolls	56(+40%)	609(-5 %)	78(+16.4%)	1724(-4.4%)	81 (+72.3 %)	1534 (-26.4 %)
Total Reduction	+64.7 %	-26.9 %	+23.8 %	-16.3 %	+179.3 %	-34.3 %

Table 2. ***Results after Unrolling the "j" loop in*** *pred*1***, the "p" loop in*** *pred*2 ***and the "c" loop in*** *tiler***.**

unrolled once, and the third row for when the loop is unrolled 3 times for the *pred*1 and *pred*2 designs and 4 times for the *tiler* design. 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.

Note that, in our approach, the number of times a loop can be unrolled without adding a loop condition check inside the loop is determined by the loop iteration count in the original code. The loops that are unrolled in *pred*1, *pred*2 and *tiler* have an iteration count of 8, 8 and 10 respectively. For an iteration count of N, a loop can be unrolled M times such that N/(1+M) is an integer and less than or equal to 1. Hence, for N=8, possible values of M are 1, 3 and 7. For N=10, possible values of M are 1, 4 and 9.

The results in the second row of Table 2 show that with one unroll, we can achieve improvements ranging from 12 % to 23 % 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 leads to only a further improvement of 4 to 5 %. This indicates that unrolling the loop just once exposes most of the parallelism available between the different loop iterations for these designs. In contrast, for the *tiler* design, we can get a 26 % reduction in the cycles on the longest path by unrolling the loop four times.

The results in Table 2 also show that loop unrolling leads to a large increase in the size of the FSM controller (number of states). This is because when a duplicate of the loop body is added to the original loop body, the number of control steps in the schedule increases, even though the number of times that the loop body executes reduces.

The last row in Table 2 lists the total reduction in the number of states and cycles after 3 unrolls over no loop unrolls. These results lead us to believe that loop unrolling can lead to improvements ranging from 16 to 34 % in the schedule length (cycles on the longest path). However, as we will see in the next section, the increase in the complexity of the controller and the multiplexing often undoes these gains in schedule length.

[2]Loops are unrolled 3 times for MPEG-1 pred1 and pred2 functions and 4 times for the GIMP tiler function. See Section 8.1.

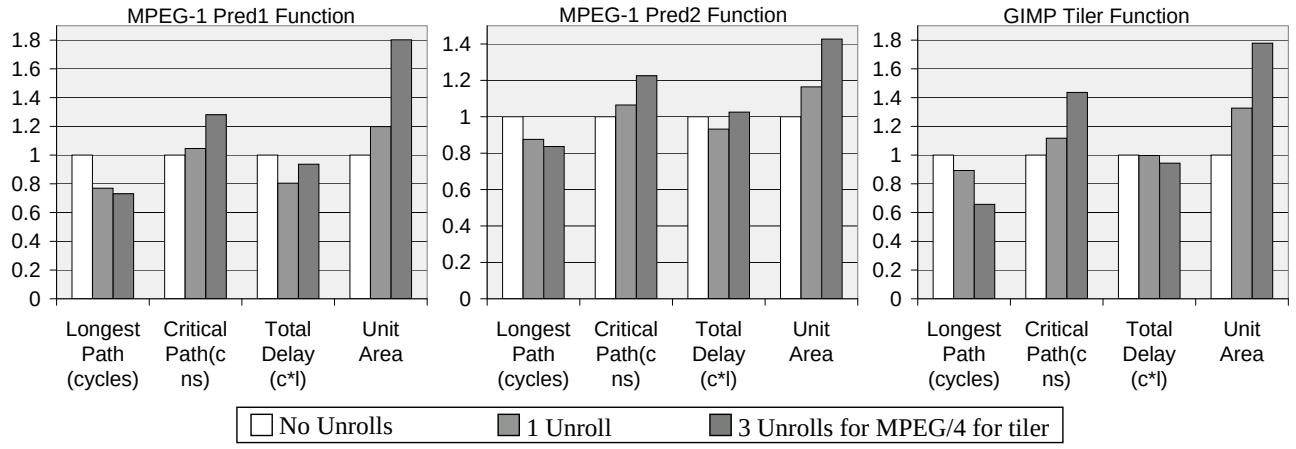

Figure 6. ***Loop Unrolling: logic synthesis results for the MPEG*** *Pred*1 ***and*** *Pred*2 ***functions and the GIMP*** *tiler* ***function.***

To study the impact of the larger controllers, we synthesized the RTL VHDL produced after scheduling and binding by our high-level synthesis tool, using the Synopsys Design Compiler logic synthesis tool. These logic synthesis results are presented in the graphs in Figure 6. The metrics mapped in these graphs are the cycles on the longest path (from the scheduling results above), the critical path length in nanoseconds (as determined by the static timing analysis tool), the total input to output delay 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). We have used the LSI-10K library for technology mapping. All values are normalized by the no unrolling case.

The results in these graphs show that when the loops are unrolled once, on the one hand, cycles on the longest path decrease by 12 to 23 %, and on the other hand, the critical path lengths increase by about 10 %. Hence, the total (longest) input to output delay through *pred*1 decreases by 20 %, through *pred*2 decreases by 7 % and through *tiler* remains almost constant. However, there is a substantial corresponding increase in the area – between 20 to 30 %. Unrolling the loop any further makes the results worse for all 3 designs: area increases from 40 to 80 % of the baseline case (no loop unrolling).

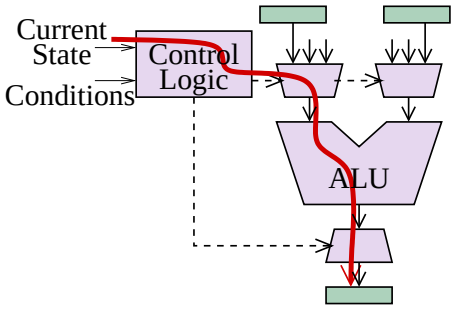

Figure 7. ***Typical critical paths in control-intensive designs pass through multiplexers and associated control logic.***

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). A typical critical path in the synthesized designs is shown in Figure 7. It originates in the control logic that generates the select signals for the multiplexers connected

to the functional units, passes through the multiplexers, the functional unit, the de-multiplexer and terminates in an output register. As the loops are unrolled, the number of operations in the design increase. Hence, a larger number of operations are mapped to the same number of resources. This increased resource utilization in turn leads to an increase in the size of the steering logic shown in Figure 7.

8.2 Synthesis Results for Loop Shifting

Table 3 lists the scheduling results after loop shifting for the 3 designs. The first row lists the results for the case when no loop shifting is done on the designs, the second row for when the loop is shifted once, the third row for 2 shifts and so on till 5 shifts. The percentage reductions of each row over the previous row are given in parentheses. The last row gives the total reduction of the fifth row (5 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.

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 Table 3, we can see that the best scheduling results are achieved for all the designs after shifting the loop 5 times (highlighted in bold font). The total reductions (last row) in cycles on the longest path after shifting range from 8.9 % to 22 %. Also, the size of the controller remains fairly constant after loop shifting.

These scheduling results are reflected in the critical path length and area results obtained after logic synthesis. These are given in Figure 8. The bars in these graphs correspond to no shifts, 3 shifts and then 5 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 only about half the increases in the controller and multiplexer costs over loop unrolling. Area does increase for the *tiler* design, however, this is because in this design we have allocated one multiplier that is used by a large number of designs. Allocating more than one multiplier makes the area of this design too large.

In Figure 9, we plotted the cycles on the longest path and the total delay for no loop unrolling, unrolling once and unrolling 3 times for *pred*1 and *pred*2 and 4 times for *tiler*. The same two metrics have been plotted for shifting the loop from zero to five times. From this graph, we can better visualize that the reductions in cycles

Transformation Applied	*MPEG*-1 *pred*1		*MPEG*-1 *pred*2		*GIMP tiler*	
	# States	# cycles	# States	# cycles	# States	# cycles
No Shifts	34	833	63	2060	29	2334
1 Shift	33(-2.9 %)	769(-7.7 %)	62(-1.6 %)	1996(-3.1 %)	28 (-3.5 %)	2234 (-4.3 %)
2 Shifts	33(0 %)	769(0 %)	62(0 %)	1996(0 %)	28 (0 %)	2144 (-4 %)
3 Shifts	32(-3 %)	705(-8.3 %)	61(-1.6 %)	1932(-3.2 %)	27 (-3.6 %)	1954 (-8.9 %)
4 Shifts	33(+3.1 %)	713(+1.1 %)	62(+1.6 %)	1940(+0.4 %)	28 (0 %)	1874 (-4.1 %)
5 Shifts	***32(-3 %)***	***649(-9 %)***	***61(-1.6 %)***	***1876(-3.3 %)***	***28 (0 %)***	***1874 (0 %)***
Total Reduction	-5.9 %	-22.1 %	-3.2 %	-8.9 %	+3.5 %	-19.7 %

Table 3. ***Results after Shifting the "j" loop in*** *pred*1***, the "p" loop in*** *pred*2 ***and the "c" loop in*** *tiler*. ***Further shifting does not improve scheduling results.***

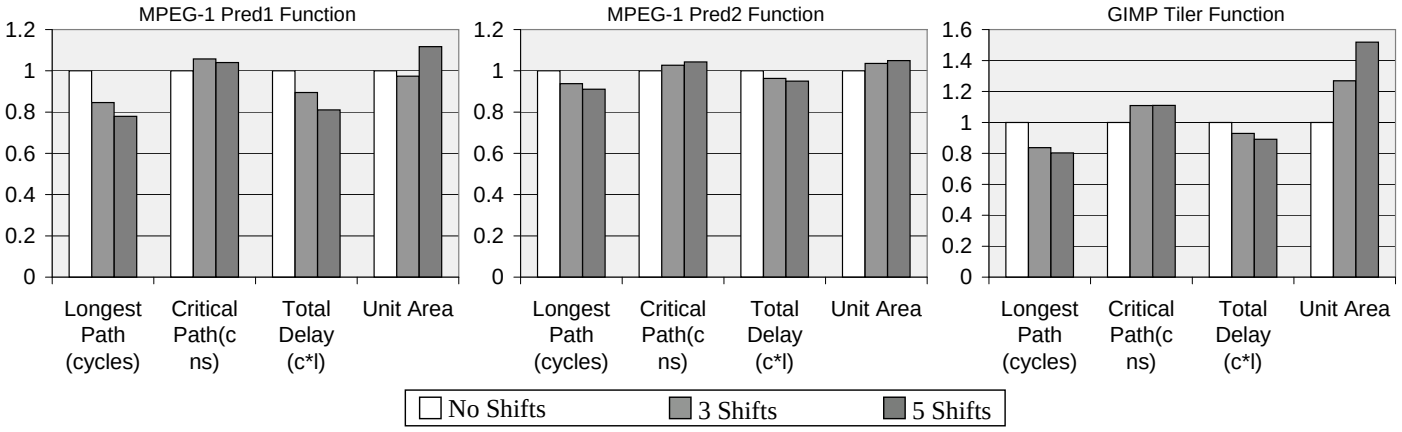

Figure 8. ***Loop Shifting: Logic synthesis results for the MPEG*** *Pred*1 ***and*** *Pred*2 ***functions and the GIMP*** *tiler* ***function.***

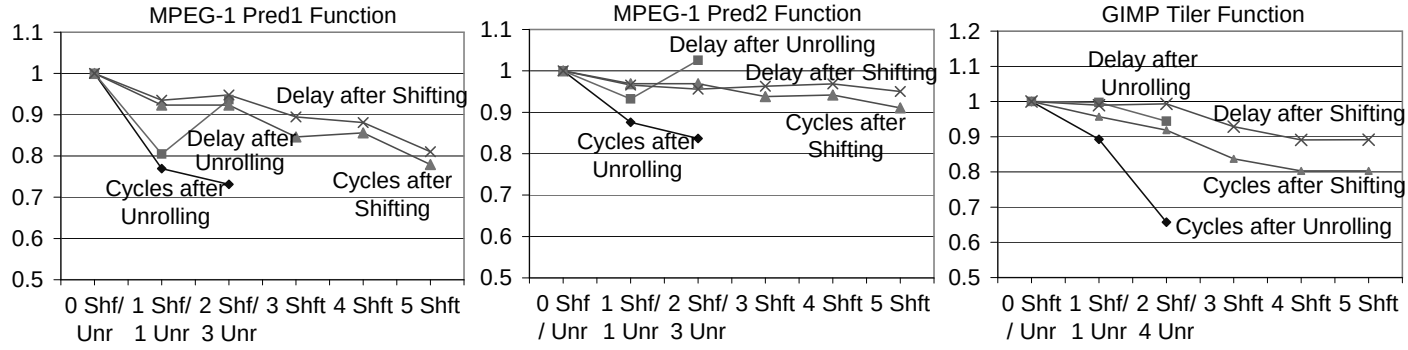

Figure 9. ***Comparison between loop unrolling and loop shifting in terms of cycles on longest path and total circuit delay.***

Transformation Applied	*MPEG*-1 *pred*1		*MPEG*-1 *pred*2		*GIMP tiler*	
	# States	# cycles	# States	# cycles	# States	# cycles
No Unrolls	34	833	63	2060	29	2334
1 Unroll	40(+17.6 %)	641(-23.1 %)	67(+6.3 %)	1804(-12.4 %)	47 (+62.1 %)	2084 (-10.7 %)
1 Unroll + 3 Shifts	38(-5 %)	577(-10 %)	66(-1.5 %)	1772(-1.8 %)	43 (-8.5 %)	1724 (-17.3 %)
Total Reduction	+11.8 %	-30.7 %	+4.8 %	-14 %	+48.3 %	-26.1 %

Table 4. ***Results after Unrolling and Shifting the "j" loop in*** *pred*1***, the "p" loop in*** *pred*2 ***and the "c" loop in*** *tiler*.

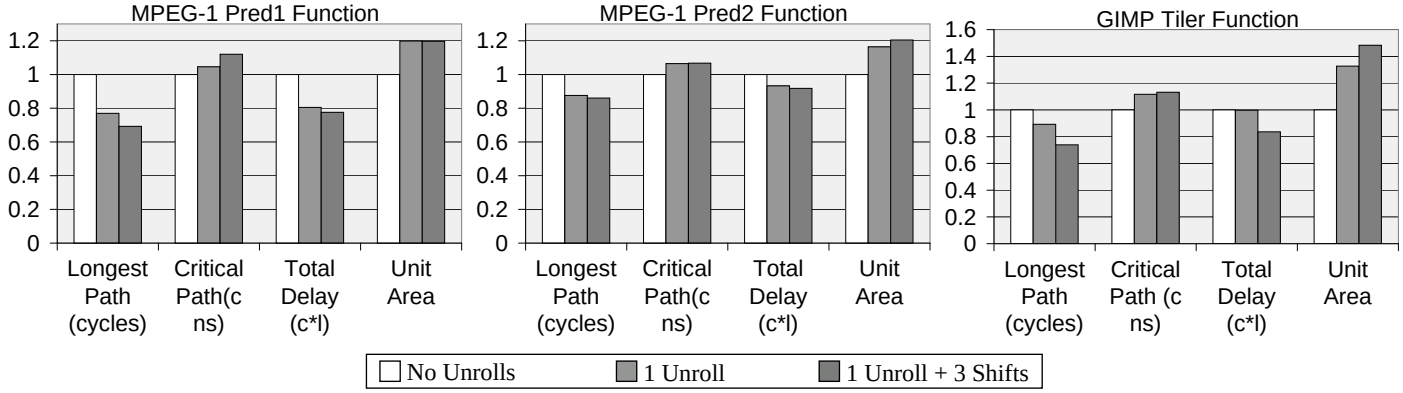

Figure 10. ***Loop Unrolling and Shifting: Logic synthesis results after no unrolls, 1 unroll and then 1 unroll and 3 shifts.***

from unrolling are larger than those obtained by loop shifting. However, it is also evident that when we look at the total delay of the circuit, loop shifting outperforms loop unrolling. Hence, although we can achieve better schedule lengths with loop unrolling, the lowest overall delays for the final synthesized circuit are achieved by loop shifting.

8.3 Combining Loop Unrolling and Shifting

Our results indicate that partially unrolling a loop can sometimes be beneficial. Hence, we experimented with performing loop shifting after partial loop unrolling in a bid to extract even more parallelism. This is done by first unrolling the loop, scheduling the design, performing loop shifting and then re-scheduling the loop.

The scheduling results for these experiments are presented in Table 4 for the three designs. The first row lists the results with no loop unrolling and shifting, the second row for when the loop is unrolled once and the third row for when the loop is unrolled once and then shifted 3 times. From these results, we observe that loop shifting can reduce the cycles on the longest path by 2 to 17 % after loop unrolling. Furthermore, the number of states in the controller also reduces by 1 to 8 %. Loop shifting thus helps in better compaction of the loop body after unrolling.

The logic synthesis results for these experiments are given in Figure 10. The results in these graphs support

the scheduling results we see in Table 4. We see that loop shifting can be used in conjunction with loop unrolling to obtain an additional 5 to 10 % improvement in total delay. The area overhead is small when compared to only loop unrolling. For the *tiler* design, we observe 17 % worse area results with an equal improvement in delay.

9 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. We implemented the loop shifting algorithm in a C-to-VHDL parallelizing high-level synthesis framework; this algorithm and the synthesis framework are particularly suited for the synthesis of designs with complex control flow. We find that when resources are tightly constrained, the control and multiplexing overheads of loop unrolling undo the gains achieved in schedule lengths. Also, the increases in area are unacceptably large. In comparison, we are always able to achieve better synthesis results with loop shifting – up to 20 % lower delays with an increase of about 10 % in area. We also demonstrated how loop shifting can be performed in conjunction with loop unrolling. We presented scheduling *and* logic synthesis results for experiments on industrial-strength designs drawn from multimedia and image processing applications. In future work, we want to develop cost models to estimate the impact of the loop transformations on control and interconnect (and thus, critical path length and area) of the final synthesized netlist. These cost models can then be used to guide the loop transformation heuristics.

Acknowledgments

This project is funded by the Semiconductor Research Corporation under Task I.D. 781.001.

References

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

[2] I. Radivojevic and F. Brewer. A new symbolic technique for control-dependent scheduling. *IEEE Transactions on CAD*, January 1996.

[3] G. Lakshminarayana, A. Raghunathan, and N.K. Jha. Incorporating speculative execution into scheduling of control-flow intensive behavioral descriptions. In *Design Automation Conference*, 1998.

[4] S. Gupta, N. Savoiu, S. Kim, N.D. Dutt, R.K. Gupta, and A. Nicolau. Speculation techniques for high level synthesis of control intensive designs. In *Design Automation Conference*, 2001.

[5] S. Gupta, N. Savoiu, N.D. Dutt, R.K. Gupta, and A. Nicolau. Conditional speculation and its effects on performance and area for high-level synthesis. In *International Symposium on System Synthesis*, 2001.

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

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

[8] A. Aiken and A. Nicolau. Perfect Pipelining: A new loop parallelization technique. In *European Symposium on Programming*, 1988.

[9] M. Lam. Software pipelining: An effective scheduling technique for VLIW machines. In *ACM SIGPLAN Conference Programming Languages Design Implementation*, 1988.

[10] S. Novack and A. Nicolau. An efficient, global resource-directed approach to exploiting instruction-level parallelism. In *Conference on Parallel Architectures and Compilation Techniques*, 1996.

[11] E. Girczyc. Loop winding - a data flow approach to functional pipelining. In *International Symposium of Circuits and Systems*, 1987.

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

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

[14] L.-F. Chao, A. S. LaPaugh, and E. H.-M. Sha. Rotation scheduling: A loop pipelining algorithm. In *Design Automation Conference*, 1993.

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

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

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

[18] 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.

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

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

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