Dynamic Common Sub-Expression Elimination during Scheduling in High-Level Synthesis

Sumit Gupta Mehrdad Reshadi Nick Savoiu
Nikil Dutt Rajesh Gupta Alex Nicolau

Center for Embedded Computer Systems
University of California at Irvine
http://www.cecs.uci.edu/∼spark
{sumitg, reshadi, savoiu, dutt, rgupta, nicolau}@cecs.uci.edu

ABSTRACT

We introduce a new approach, "Dynamic Common Sub-expression Elimination (CSE)", that dynamically eliminates common sub- expressions based on new opportunities created during scheduling of control-intensive designs. Classical CSE techniques fail to eliminate several common sub-expressions in control-intensive designs due to the presence of a complex mix of control and data-flow. Aggressive speculative code motions employed to schedule control-intensive designs often re-order, speculate and duplicate operations, hence changing the control flow between the operations with common sub-expressions. This leads to new opportunities for applying CSE dynamically. We have implemented dynamic CSE in a high-level synthesis framework called *Spark* and present results for experiments performed using various combinations of CSE and dynamic CSE. The benchmarks used consist of four functional blocks derived from two moderately complex industrial-strength applications, namely, MPEG-1 and the GIMP image processing tool. Our dynamic CSE techniques result in improvements of up to 22 % in the controller size and up to 31 % in performance; easily surpassing the improvements obtained by the traditional CSE approach. We also observe an unexpected (and significant) reduction in the number of registers using our approach.

Categories and Subject Descriptors: B.5.1 [Register-Transfer-Level Implementation] Design Aids
General Terms: Design, Performance
Keywords: High-level synthesis, Common Sub-Expression Elimination, Dynamic CSE, Parallelizing Transformations

1. INTRODUCTION

The quality of synthesis results for most high level synthesis approaches is strongly affected by the choice of control flow (through conditions and loops) in the input description. This has led to a need for high-level and compiler transformations that overcome the effects of syntactic variance or programming style on the quality of generated circuits. To address this need, a set of speculative code motion transformations have been developed that enable movement of operations through, beyond, and into conditionals with the objective of maximizing performance [1, 2].

ISSS'02, October 2–4, 2002, Kyoto, Japan.

These code motions have been shown to be effective in improving both the scheduling and the synthesis results of high-level synthesis, particularly for control-intensive designs. Such code motions re-order, speculate and sometimes duplicate operations, creating opportunities for dynamically eliminating common sub-expressions during scheduling.

Operations with common sub-expressions are often present in designs with moderately complex control flow. However, these operations are typically within conditional branches and hence, do not execute under all conditions. This mix of control-data flow limits the effectiveness of common sub-expression elimination (CSE), when applied as a pass before scheduling. However, the movement and duplication of operations caused by speculative code motions frequently creates new opportunities for applying transformations such as CSE and copy propagation. This has led us to develop a new technique called *Dynamic CSE* that exploits these opportunities dynamically as they arise.

Dynamic CSE operates during scheduling by eliminating common sub-expressions between the operation that has been scheduled and the other operations that are ready to be scheduled. The heuristic takes advantage of the possible new position or duplication of the scheduled operation. Any additional copy operations generated during scheduling are also dynamically propagated as explained in Section 4.1. Applying CSE as a pass *after* scheduling is not as effective as performing dynamic CSE during scheduling, since the latter eliminates some operations from the design graph, and other operations may get scheduled in lieu of these eliminated operations.

We have implemented dynamic CSE and dynamic copy propagation along with various other code transformations such as code motions, in a high-level synthesis framework called *Spark* that takes a behavioral description in ANSI-C as input and generates synthesizable register-transfer level VHDL. We use this system to evaluate the effectiveness of dynamic CSE on some large benchmarks. The results obtained from these experiments demonstrate that dynamic CSE is an effective technique that not only reduces execution cycles and controller size, but also reduces the number of registers and the area of the final synthesized design.

The rest of this paper is organized as follows: the next section discusses previous related work and gives an overview of CSE. Section 3 reviews the speculative code motions in our synthesis system. Next, we present dynamic CSE and demonstrate how it can exploit the new CSE opportunities created by these code motions. We also show how this technique has been extended to dynamic copy propagation. Section 6 presents our experimental setup and the results and Section 7 concludes the paper.

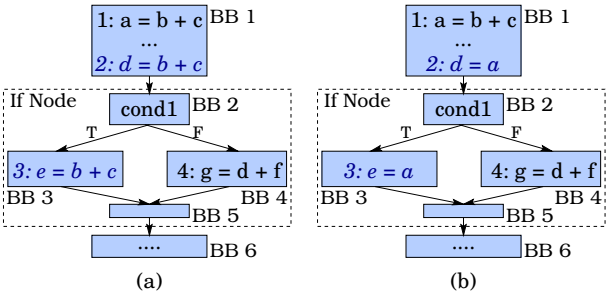

Figure 1: CSE: (a) a sample control-data flow graph (b) the common sub-expression $b+c$ in operations 2 and 3 has been replaced with the variable a from operation 1.

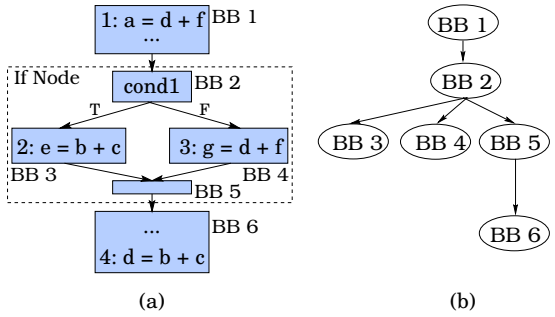

Figure 2: Basic Block Dominance: (a) Operation 4 cannot be replaced with variable from operation 2 (b) Basic block dominator tree for the example

2. PREVIOUS WORK

CSE is a standard transformation that is implemented in most software compilers [3, 4]. In the domain of high-level synthesis, CSE has been used for throughput improvement [5], for optimizing multiple constant multiplications [6, 7] and as an algebraic transformation for operation cost minimization [8, 9]. [10] and [11] present the converse of CSE, namely, common subexpression replication, whereby a redundant operation is inserted to aid scheduling. A compiler transformation called partial redundancy elimination (PRE) [12] inserts copies of operations present in only one conditional branch into the other conditional branch, so as to eliminate common sub-expressions in subsequent operations. The authors in [8, 13] propose doing CSE at the source-level to reduce the effects of the factorization of expressions and control flow on the results of CSE.

Mutation scheduling [14] also performs local optimizations such as CSE during scheduling in an opportunistic, context-sensitive manner; whereas dynamic CSE is a guided technique that systematically and globally applies CSE during scheduling.

2.1 Classical CSE

Common sub-expression elimination (CSE) is a well-known transformation that attempts to detect repeating sub-expressions in a piece of code, stores them in a variable and reuses the variable wherever the sub-expression occurs subsequently [3]. Hence, as shown by the example in Figure 1, the common sub-expression $b+c$ in operations 2 and 3 can be replaced with the result of operation 1.

Whether a common sub-expression between two operations can be eliminated depends on the control flow between the locations or basic blocks of the two operations. One common approach to capture the relationship between basic blocks in a control flow graph is using *dominator trees* [3, 15]. These trees can be constructed using

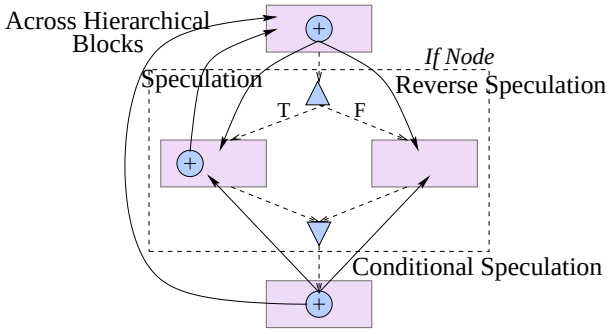

Figure 3: Various speculative code motions: operations may be speculated, reverse speculated, conditionally speculated or moved across entire conditional blocks.

the following definition: a node d in a control flow graph (CFG) is said to *dominate* another node n, if every path from the initial node of the flow graph to n goes through d.

Consider the example in shown in Figure 2(a) and its corresponding dominator tree in Figure 2(b). In this example, basic block BB_2 dominates basic blocks BB_3, BB_4 and BB_5 and is itself dominated by BB_1. BB_5 in turn dominates BB_6.

In order to preserve the control-flow semantics of a CFG, the common sub-expression in an operation op_2 can only be replaced with the result of another operation op_1, if op_1 resides in a basic block BB_1 that dominates the basic block BB_2 in which op_2 resides. Hence, in the example in Figure 2(a), BB_3 does *not* dominate BB_6 as per the dominator tree shown in Figure 2(b). So, the common sub-expression in operation 4 cannot be replaced with the result of operation 2. On the other hand, operation 3 can be eliminated using the result of operation 1.

Dominator trees have been extensively used previously for data flow analysis and transformations such as loop-invariant code motion and CSE [3, 15]. They have recently been extended to incorporate the notion of sets of basic blocks dominating over other basic blocks (see Section 5) [16]. In our work as well, we use the dominator information of the basic blocks that contain the operations, to apply CSE. However, as shown next, speculative code motions can change the basic blocks that contain such operations.

3. SPECULATIVE CODE MOTIONS

To alleviate the problem of poor synthesis results in the presence of complex control flow in designs, a set of code motion transformations have been developed that re-order operations to minimize the effects of syntactic variance in the input description. These beyond-basic-block code motion transformations are usually speculative in nature and attempt to extract the inherent parallelism in designs and increase resource utilization.

Generally, speculation refers to the unconditional execution of operations that were originally supposed to have executed conditionally. However, frequently there are situations when there is a need to move operations *into* conditionals [2, 17]. This may be done by *reverse speculation*, where operations before conditionals are moved into *subsequent* conditional blocks and executed conditionally, or this may be done by *conditional speculation*, wherein an operation from after the conditional block is duplicated *up* into *preceding* conditional branches and executed conditionally. The various speculative code motions are shown in Figure 3. Also, shown is the movement of operations across entire hierarchical blocks, such as if-then-else blocks or loops.

To illustrate how these speculative code motions can create new opportunities for applying CSE, consider the example in Figure

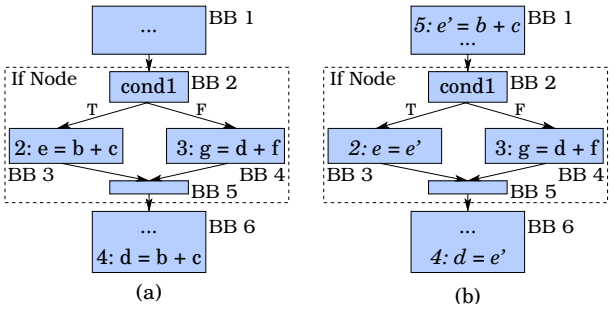

Figure 4: Dynamic CSE: (a) A sample control-data flow graph (b) Speculative execution of operation 2 as operation 5 in basic block BB_1 allows dynamic CSE to replace the common sub-expression in operation 4.

4(a). In this example, classical CSE cannot eliminate the common sub-expression in operation 4 with operation 2 since operation 4's basic block BB_6 is not dominated by operation 2's basic block BB_3. Consider now that the scheduling heuristic decides to schedule operation 2 in BB_1 and execute it speculatively as operation 5 as shown in Figure 4(b). Now, the basic block BB_1 containing this speculated operation 5, dominates operation 4's basic block BB_6. Hence, operation 4 can be eliminated and replaced by the result of operation 5, as shown in Figure 4(b).

Since CSE is traditionally applied as a pass, usually before scheduling, it can miss these kinds of opportunities. This motivated us to develop a technique by which CSE can be applied dynamically while scheduling a design.

4. DYNAMIC CSE

Dynamic CSE is a technique that operates after an operation has been scheduled. Conceptually, it examines the list of remaining ready-to-be-scheduled operations and determines which of these have a common sub-expression with the currently scheduled operation; this common sub-expression can now be eliminated due to the code motion of the currently scheduled operation. We use the term "dynamic" to differentiate from the phase ordered application of CSE before scheduling.

This transformation can be incorporated into a scheduling heuristic; we illustrate a list scheduling heuristic incorporating dynamic CSE in Figure 5. This heuristic takes as input an unscheduled control-data flow graph (CDFG) of the design and produces a resource constrained schedule. The heuristic schedules the design by traversing the CDFG in a top-down manner by considering one basic block at a time. In our implementation, control paths are followed such that at the fork node of a if-then-else conditional block, the true branch is scheduled first and then the false branch. However, unscheduled operations in other basic blocks are also considered for scheduling into the current basic block using the speculative code motions discussed above.

The heuristic starts scheduling on each resource at each control step in a basic block by collecting a list of available operations $\mathcal{A}$. *Available operations* are operations whose data dependencies are satisfied and that can be scheduled on the given resource at the current scheduling step by using the various available code motions [18]. Available operations may be collected from all unscheduled basic blocks in the CDFG. Next, the cost of scheduling each of these operations is calculated; currently, this cost function favors operations on the longest data dependency chain (critical path). However, in the future, this cost function can be made to include estimates of the control costs of a code transformation. Finally, the heuristic selects the operation op with the lowest cost and schedules it using the appropriate code motions.

Algorithm 1: Dynamic CSE during Scheduling
Inputs: Unscheduled *CDFG* of design, Resource List *R*
Output: Scheduled *CDFG* of design
1: Scheduling step $step = 0$
2: **while** ($step \neq$ last step of CDFG) **do**
3: **foreach** (resource *res* in Resource List *R*) **do**
4: Get List of Available Operations $\mathcal{A}$
5: Calculate cost of all operations in $\mathcal{A}$
6: Pick Operation op with lowest cost in $\mathcal{A}$
7: Move op and schedule on *res* in $step$
8: Get list of operations *cseOpsList* from $\mathcal{A}$ that have common sub-expressions with op
9: **foreach** (operation *cseOp* in *cseOpsList*) **do**
10: **if** (BB(op) dominates BB(*cseOp*) **then**
11: ApplyCSE(*cseOp*, op)
12: **endforeach**
13: **endforeach**
14: $step = step + 1$
13: **endwhile**

Figure 5: Incorporating Dynamic CSE in a list scheduling heuristic. Available operations are determined based on data dependencies and the ability of an operation to be moved to the scheduling step under consideration. $BB(op)$ **gives the basic block that the** op **is located in.**

As shown in line 8 of this heuristic, from the remaining operations in the available list, dynamic CSE *then* determines the list of operations, *cseOpsList*, that have a common sub-expression with the scheduled operation op. Then, for each operation *cseOp* in *cseOpsList*, if the basic block of *cseOp* is dominated by the basic block of op *after* scheduling, then it replaces the common sub-expression in *cseOp* with the result from op (by calling *ApplyCSE*).

We illustrate this heuristic using the earlier example from Figure 4(a). In this example, consider that while scheduling basic block BB_1, the scheduling heuristic determines that available operations are operations 2, 3 and 4. Of these operations, the heuristic schedules operation 2 in BB_1. Then, the dynamic CSE heuristic examines the remaining operations in the available list, namely operations 3 and 4, and detects and replaces the common sub-expression $(b+c)$ in operation 4 with the result, e', of the scheduled operation 5, since $BB(op_5)$ dominates $BB(op_4)$.

This example also demonstrates that applying CSE as a pass *after* scheduling is not as effective as dynamic CSE, because in this example, the resource freed up by eliminating the common sub-expression in operation 4, can now potentially be used to schedule another operation by the scheduler. Performing CSE after scheduling would be too late to effect any decisions by the scheduler.

4.1 Dynamic Copy Propagation

The concept of dynamic CSE can also be applied to *copy propagation*. After code motions such as speculation and transformations such as CSE, there are usually several copy operations left behind. Copy operations read the result of one variable and write them to another variable. For example in Figure 4(b), operations 2 and 4 copy variable e' to variables e and d respectively.

These variable copy operations can be propagated forward to operations that read their result. Again, traditionally *copy propagation* is done as a compiler pass before and after scheduling to eliminate unnecessary use of variables. However, we have found that it is essential to propagate the copies created by speculative code motions and dynamic CSE, *dynamically* during scheduling itself, since this enables opportunities to apply CSE on subsequent operations that read these variable copies. After copy propagation,

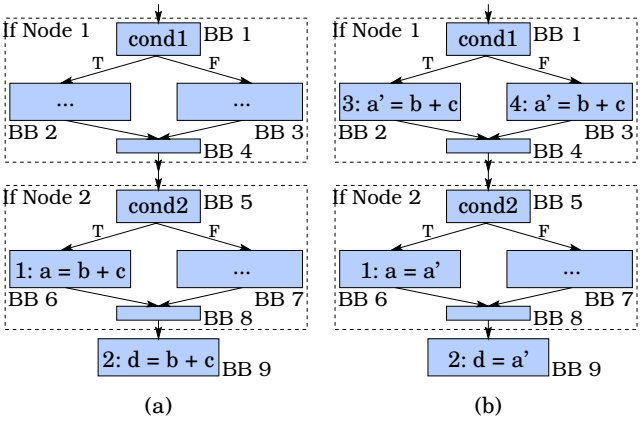

Figure 6: Dynamic CSE after conditional speculation: (a) A sample CDFG (b) Operation 1 has been conditionally speculated into BB_2 and BB_3. This allows dynamic CSE to be performed for operation 2 in BB_9.

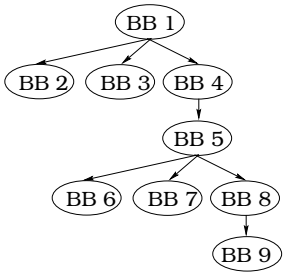

Figure 7: Dominator tree for the example in Figure 6

these dependent operations can directly use the result of the operation that creates the variable in the first place, rather than its copy operation. A dead code elimination pass after scheduling can then remove unused copies.

5. CONDITIONAL SPECULATION AND DYNAMIC CSE

Another code motion that has a significant impact on the number of opportunities available for CSE is conditional speculation [2]. *Conditional speculation* duplicates operations up into the true and false branches of a if-then-else conditional block. This is demonstrated by the example in Figure 6(a). Consider that the scheduling heuristic decides to conditionally speculate operation 1 into the branches of the if-then-else conditional block $IfNode_1$. Hence, as shown in Figure 6(b), the operation is duplicated up as operations 3 and 4 in basic blocks BB_2 and BB_3 respectively.

If we look at the original description in Figure 6(a) again, we note that operation 2 in BB_9 has a common sub-expression with operation 1 in BB_6. But since BB_9, is not dominated by BB_6 (see dominator tree in Figure 7), this common sub-expression cannot be eliminated by classical CSE. However, after conditional speculation, operations with this common sub-expression exist in all control paths leading up to BB_9. Hence, we can apply dynamic CSE now and operation 2 uses the result, a', of operations 3 and 4 as shown in Figure 6(b).

This leads to the notion of dominance by sets of basic blocks [16]. A set of basic blocks can dominate another basic block, if all control paths to the latter basic block come from at least one of the basic blocks in the set. Hence, in Figure 6(b), basic blocks BB_2 and BB_3 together dominate basic block BB_9, hence, enabling dynamic CSE of operation 2. In this manner, we use this property of domination by sets of basic blocks while performing dynamic CSE along with code motions such as reverse and conditional speculation that duplicate operations in the control-data flow graph.

Another case, in which dynamic CSE is applied in conjunction with conditional speculation, arises when an operation is duplicated into a basic block in which another operation with the same expression already exists. In this case, the operation being duplicated is instead replaced with a copy operation using the result of the already present operation with the same expression. In this way, we are again able to reduce the number of operations with common sub-expressions in the final scheduled design.

6. EXPERIMENTAL SETUP AND RESULTS

Dynamic CSE and copy propagation have been implemented in our high-level synthesis research framework called *Spark* [17]. This synthesis system takes a behavioral description in ANSI-C as input and generates synthesizable register-transfer level VHDL. This enables the system to evaluate the effects of several coarse and fine-grain optimizations on logic synthesis results. Code motion techniques such as Trailblazing [19] are used to enable the parallelizing, speculative code motions. These code motions are supported by standard compiler transformations such as CSE, copy and constant propagation and dead code elimination.

We have chosen two large and moderately complex real-life applications, representative of the multimedia and image processing domains, to perform experiments using various combinations of CSE and dynamic CSE. We present results for two functions from the *Prediction* block of the MPEG-1 algorithm [20] and for two functions derived the GIMP image processing tool [21]. The MPEG functions used are the *pred*0_1 and *pred*2 functions and GIMP functions are the functions in the "tiler" transform and the *calc_undistorted_coords* from the "Polarize" transform[1]. The two functions (scale and tile) in the tiler transform have been inlined into one function, that we call "tile".

6.1 High-Level Synthesis Results

The synthesis results for these benchmarks are presented in Tables 1 and 2. These results are in terms of the number of states in the finite state machine controller, the cycles on the longest path (i.e. execution cycles) and the number of registers in the output VHDL. For loops, the longest path length of the loop body is multiplied by the number of loop iterations. The resources are indicated in the tables; *ALU* 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. The number of non-empty basic blocks and operations in the functions are also given.

The first row in both these tables presents synthesis results for when CSE is not applied, when only CSE is applied as a pass before scheduling (2nd row), when only dynamic CSE is applied (3rd row) and finally, both CSE and dynamic CSE are applied (4th row). In all these experiments, dynamic copy propagation is done whenever possible (even when dynamic CSE is not applied) and the baseline is with all parallelizing code motions enabled but no CSE applied. The percentage reductions of each row over the *first* row (no CSE baseline case) are also given in parentheses.

The second row in both Table 1 and Table 2 demonstrates that when CSE alone is applied, improvements ranging from 3 to 25 %

[1]Note that these floating point functions have been arbitrarily converted to integer functions for the purpose of our experiments. This does not effect the nature of the control flow, but only the way the data is handled.

MPEG Pred. Block; Resources = 3*ALU*, 2[ ], 3 <<, 2 ==,1*; BBs = non-empty Basic Blocks						
Transformation Applied	*pred*2 (217 Ops, 45 BBs)			*pred*0_1 (101 Ops, 26 BBs)		
	# States	Long Path	# Regs	# States	Long Path	# Regs
No CSE	52	2260	26	55	1051	18
with CSE	50(-3.8%)	2188(-3.2%)	26(0%)	53(-3.6%)	987(-6.1%)	17(-5.6%)
with Dyn CSE	45(-13.5%)	1676(-25.8%)	17(-34.6%)	48(-12.7%)	731(-30.4%)	8(-55.6%)
with CSE & Dyn CSE	43(-17.3%)	1676(-25.8%)	15(-42.3%)	46(-16.4%)	731(-30.4%)	10(-44.4%)

Table 1: Scheduling results after applying CSE and Dynamic CSE for functions from the MPEG-1 Prediction block

Transformation Applied	tile with inlined scale(145 Ops,35 BBs) Resources=3+, 2−, 1/, 2∗, 2 <<, 2 ==, 1[ ]			polar: *calc_undist_coords*(252 Ops,78 BBs) Resources=2+, 3−, 1/, 2∗, 2 <<, 2 ==		
	# States	Long Path	# Regs	# States	Long Path	# Regs
No CSE	41	3131	34	47	47	27
with CSE	34(-17.1%)	2331(-25.6%)	22(-35.3%)	47(0%)	47(0%)	25(-7.4%)
with Dyn CSE	32(-22%)	2131(-31.9%)	17(-50%)	45(-4.3%)	45(-4.3%)	22(-22.7%)
with CSE & Dyn CSE	32(-22%)	2131(-31.9%)	18(-47.1%)	45(-4.3%)	45(-4.3%)	22(-22.7%)

Table 2: Scheduling results after applying CSE and Dynamic CSE for two functions from the GIMP image processing tool

in the number of states and longest path cycles are obtained. In itself these improvements are good.

Also, contrary to common belief, the results in these tables show that applying CSE leads to a *reduction* in the number of registers required. This decrease can be attributed to three factors: (a) the reduced schedule lengths imply shorter variable lifetimes, especially for variables whose results are required for future loop iterations, (b) elimination of an operation by CSE means that the variables read by the operation, can potentially be killed earlier and hence, have shorter lifetimes, and (c) the speculative nature of the code motions means that there are operations with the same subexpression being executed anyway; by eliminating some of these operations and then performing copy propagation, CSE is able to increase reuse of the result from only one of the operations, instead of storing the results of several operations.

When dynamic CSE is applied, the improvements are even more dramatic, especially for the MPEG functions. As the third row in Table 1 shows, the number of states and longest path cycles reduce as much as 13 % and 30 % respectively, whereas register usage can go down by as much as 55 % for the *pred*0_1 function. Note that since both these functions have doubly nested loops, any reduction in schedule length of the loop body leads to large overall reductions.

When both CSE and dynamic CSE are applied to these functions (4th row of Table 1), the improvements in performance and controller size are the same as when only dynamic CSE is applied. And, although the *pred*2 function requires fewer registers, the *pred*0_1 function actually requires two extra registers with this combination – but this is due to the fact that our resource binding methodology attempts to minimize interconnect and hence, area – sometimes at the expense of registers [2]. This is validated later in Section 6.2, when we look at the area after logic synthesis of these functions. The total improvements in performance and controller size after applying CSE and dynamic CSE for the functions from the MPEG benchmark range from 16 to 30 % and number of registers reduce by as much as 44 % (last row of table).

The results for the functions from the GIMP tool in Table 2 also show improvements when dynamic CSE is used (3rd row of table) versus applying CSE alone (2nd row). Whereas for the *tiler* function, both CSE and dynamic CSE leads to improvements, for the *polarize* function, only dynamic CSE leads to improvements in performance and controller size. The improvements for the *tiler* function are in the range of 22 to 31 %, and those of the *polarize* function are in the 4 % range. Note that, the *calc_undistorted_coords* function from the *Polarize* transform is called in two places, both times in a doubly nested loop, since the transform is applied for each pixel in an image. Hence, the improvements shown in Table 2 will multiply by the number of iterations of both the nested loops.

Overall the results from these tables demonstrate that dynamic CSE is always able to out perform doing CSE alone as a pass before scheduling. We also verified that runnint CSE as a pass both before scheduling and after scheduling did not lead to any improvements over running CSE only before scheduling.

6.2 Logic Synthesis Results

In this section, we present results for our experiements on the effects of CSE and dynamic CSE on the final net list generated by logic synthesis. We synthesized the VHDL generated by the Spark system for the MPEG functions using the logic synthesis tool, *Design Compiler* from Synopsys. The logic synthesis results are presented in Figure 8. In these graphs, three metrics are mapped: the critical path length, the unit area and the maximum delay through the design. The critical path length is the length of the longest combinational path in the netlist as determined by static timing analysis. The unit area is in terms of the synthesis library used (the LSI-10K library) and the maximum delay is the product of the longest path length (in cycles) and the critical path length (in nanoseconds) and signifies the maximum input to output latency of the design. The values of each metric are normalized by the case when no CSE or dynamic CSE is applied.

The results in the graphs in Figure 8 again demonstrate the usefulness of dynamic CSE. For both the MPEG functions, applying CSE alone leads to only marginal improvements in the synthesis metrics when compared to applying dynamic CSE. When dynamic CSE is applied with or without classical CSE, shorter delays and significantly lower area (up to 35 % less) are obtained, with critical path lengths remaining fairly constant or decreasing marginally.

These decreases in the area and critical path length can be attributed to two factors. Firstly, the elimination of some operations due to dynamic CSE means that fewer operations are mapped to the functional units. This leads to reduced interconnect (multiplexors and demultiplexors). Also, since dynamic CSE leads to lower register usage, this too contributes to the reduced area of the final circuit.

7. CONCLUSIONS AND FUTURE WORK

In this paper, we have presented a new approach for applying the classical compiler transformation, common sub-expression elimination (CSE) called dynamic CSE that is able to eliminate common

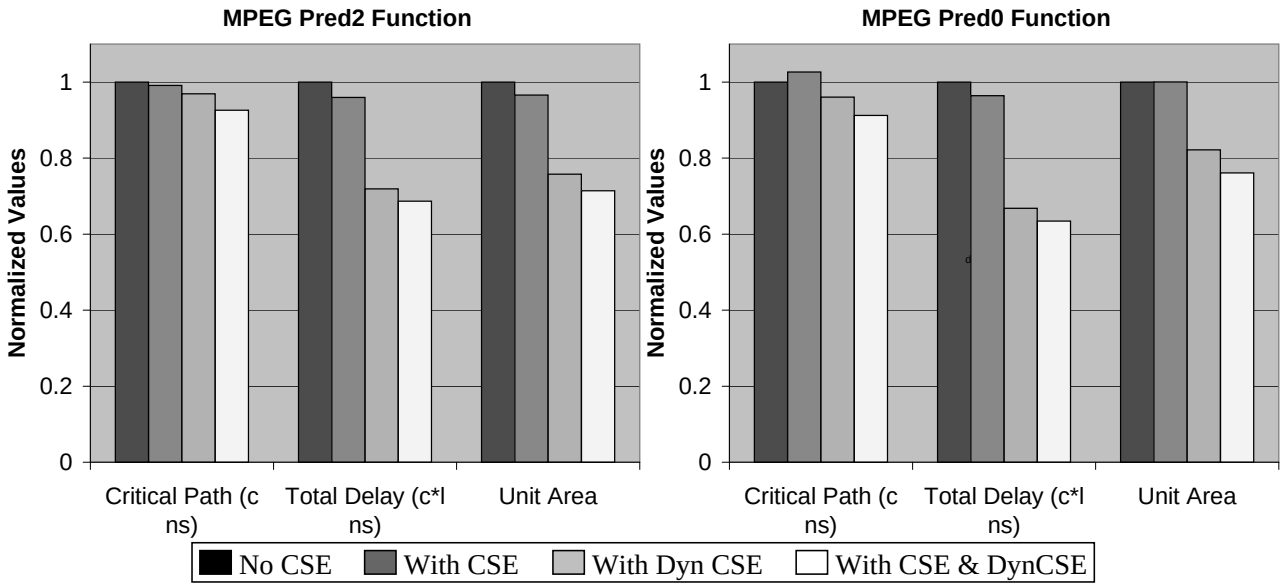

Figure 8: Effects of CSE and dynamic CSE on logic synthesis results for the MPEG *Pred*2 and *Pred*0_1 functions

sub-expressions that are missed by classical CSE due to the presence of control-flow. Dynamic CSE exploits the operation movement caused by scheduling to apply CSE based on the new position of operations that have been scheduled. This operation movement occurs due to speculative code motions that re-order, speculate and sometimes duplicate operations to achieve higher performance and maximize parallel execution.

We have demonstrated the effectiveness of dynamic CSE over classical CSE on two moderately complex and control-intensive benchmarks derived from real-life multimedia and image processing applications. We obtain significant improvements in performance, controller size and area of the design implementation. Furthermore, we observe a non-intuitive and significant reduction in register usage due to shorter schedule lengths and elimination of duplicate operations. These results encourage us to look at other transformations that may be applied dynamically and may be able to take advantage of the transformations applied during scheduling.

8. REFERENCES

[1] L.C.V. dos Santos and J.A.G. Jess. A reordering technique for efficient code motion. In *Design Automation Conf.*, 1999.

[2] 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 *Intl. Symp. on System Synthesis*, 2001.

[3] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles and Techniques and Tools*. Addison-Wesley, 1986.

[4] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[5] Z. Iqbal, M. Potkonjak, S. Dey, and A. Parker. Critical path optimization using retiming and algebraic speed-up. In *Design Automation Conference*, 1993.

[6] M. Potkonjak, M.B. Srivastava, and A. Chandrakasan. Multiple constant multiplications: Efficient and versatile framework and algorithms for exploring common subexpression elimination. *IEEE Trans. on CAD*, Mar 1996.

[7] R. Pasko, P. Schaumont, V. Derudder, S. Vernalde, and D. Durackova. A new algorithm for elimination of common subexpressions. *IEEE Trans. on CAD*, Jan 1999.

[8] M.Janssen, F.Catthoor, and H.De Man. A specification invariant technique for operation cost minimisation in flow-graphs. In *Intl. Symp. on High-level Synthesis*, 1994.

[9] M.Miranda, F.Catthoor, M. Janssen, and H.De Man. High-level address optimisation and synthesis techniques for data-transfer intensive applications.

[10] D.A. Lobo and B.M. Pangrle. Redundant operator creation: A scheduling optimization technique. In *Design Automation Conference*, 1991.

[11] M. Potkonjak and J. Rabaey. Maximally fast and arbitrarily fast implementation of linear computations. In *International Conference on CAD*, 1992.

[12] R. Kennedy, S. Chan, S.-M. Liu, R. Io, P. Tu, and F. Chow. Partial redundancy elimination in SSA form. *ACM Trans. Progrm. Languages and Systems*, May 1999.

[13] S. Gupta, M. Miranda, F. Catthoor, and R. Gupta. Analysis of high-level address code transformations for programmable processors. In *Design, Automation and Test in Europe*, 2000.

[14] S. Novack and A. Nicolau. Mutation scheduling: A unified approach to compiling for fine-grain parallelism. In *Languages and Compilers for Parallel Computing*, 1994.

[15] V.C. Sreedhar, G. R. Gao, and Y.-F. Lee. Incremental computation of dominator trees. *ACM Trans. Progrm. Languages and Systems*, March 1997.

[16] V.C. Sreedhar, G. R. Gao, and Y.-F. Lee. A new framework for exhaustive and incremental data flow analysis using DJ graphs. *ACM SIGPLAN Conf. on PLDI*, 1996.

[17] 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.

[18] K. Ebcioglu and A. Nicolau. A global resource-constrained parallelization technique. In *3rd International Conference on Supercomputing*, 1989.

[19] A. Nicolau and S. Novack. Trailblazing: A hierarchical approach to percolation scheduling. In *International Conference on Parallel Processing*, 1993.

[20] Spark Synthesis Benchmarks FTP site. ftp://ftp.ics.uci.edu/pub/spark/benchmarks.

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