SPARK : A High-Level Synthesis Framework For Applying Parallelizing Compiler Transformations*

Sumit Gupta Nikil Dutt Rajesh Gupta Alex Nicolau

Center for Embedded Computer Systems
University of California at Irvine
http://www.cecs.uci.edu/~spark
E-mail: {sumitg,dutt,rgupta,nicolau}@cecs.uci.edu

Abstract

This paper presents a modular and extensible high-level synthesis research system, called SPARK*, that takes a behavioral description in ANSI-C as input and produces synthesizable register-transfer level VHDL. SPARK uses parallelizing compiler technology developed previously to enhance instruction-level parallelism and re-instruments it for high-level synthesis by incorporating ideas of mutual exclusivity of operations, resource sharing and hardware cost models. In this paper, we present the design flow through the SPARK system, a set of transformations that include speculative code motions and dynamic transformations and show how these transformations and other optimizing synthesis and compiler techniques are employed by a scheduling heuristic. Experiments are performed on two moderately complex industrial applications, namely, MPEG-1 and the GIMP image processing tool. The results show that the various code transformations lead to up to 70 % improvements in performance without any increase in the overall area and critical path of the final synthesized design.*

1 Introduction

Recent years have seen the widespread acceptance and use of language-level modeling of digital designs. Increasingly, the typical design process starts with design entry in a hardware description language at the register-transfer level (RTL), followed by logic synthesis. This and the increased use of high level languages for behavioral modeling has led to a renewed interest in high level synthesis from behavioral descriptions,both in the industry and in academia [1, 2, 3, 4].

However, current synthesis efforts have several limitations: synthesizability is guaranteed on a small, constrained sub-set of the input language and the language level optimizations are few and their effects on final circuit area and speed are not well understood. Also, for designs with moderately complex control flow, the quality of synthesis results is poor due to the presence of conditionals and loops.

These factors have led to a need for high-level and compiler transformations that improve the quality of synthesis results in the presence of control flow. To develop these transformations, we have developed a high-level synthesis framework, called *SPARK*. SPARK has been designed to facilitate experimentation of the application of both coarse-grain and fine-grain code optimizations and view the effects of these transformations on the resultant VHDL code. The SPARK framework provides a toolbox of code transformations and supporting compiler transformations. The toolbox approach enables the designer to apply heuristics to drive selection and control of individual transformations under realistic cost models for high-level synthesis. The synthesis framework is a complete high-level synthesis system that provides a path from an unrestricted input behavioral description down to synthesizable RTL VHDL code.

Using the SPARK framework, we have developed a set of speculative code motion transformations that enable movement of operations through, beyond, and into conditionals with the objective of maximizing performance [5, 6]. We have also developed dynamic transformations, such as dynamic CSE and dynamic copy propagation, that operate during scheduling to take advantage of change in the relative control flow between operations caused by the speculative code motions employed during scheduling [7].

Whereas we have presented the speculative code motions and the dynamic CSE transformation earlier [5, 6, 7], in this paper we present the SPARK system in which these transformations have been implemented. The contributions of this paper include: (a) an overview of the SPARK system and the design flow through the system, (b) the internal representation model used for capturing the control-intensive designs targeted by our system, (c) the *percolation* and *trailblazing* code motion techniques and the dynamic variable renaming technique implemented in the system, and (d) a scheduling heuristic that incorporates the previously presented code motion and dynamic CSE techniques.

The rest of this paper is organized as follows: in the next section, we review previous work. Section 3 presents the SPARK framework followed by the internal representation model, the code motion techniques and dynamic variable renaming techniques. We then present the speculative code motions, followed by CSE and dynamic CSE. Section 9 presents the list scheduling heuristic followed by synthesis results that demonstrate the effectiveness of these transformations. We conclude the paper with a discussion.

2 Related Work

High-level synthesis (HLS) has been a subject for research for almost two decades now [8]. Early work presented scheduling heuristics for purely data flow designs. Recent work has presented speculative code motions for mixed control-data flow designs and demonstrated their ef-

*This work is supported by SRC grant 781.001

Proceedings of the 16th International Conference on VLSI Design (VLSI'03)
1063-9667/03 $17.00 © 2003 IEEE

IEEE COMPUTER SOCIETY

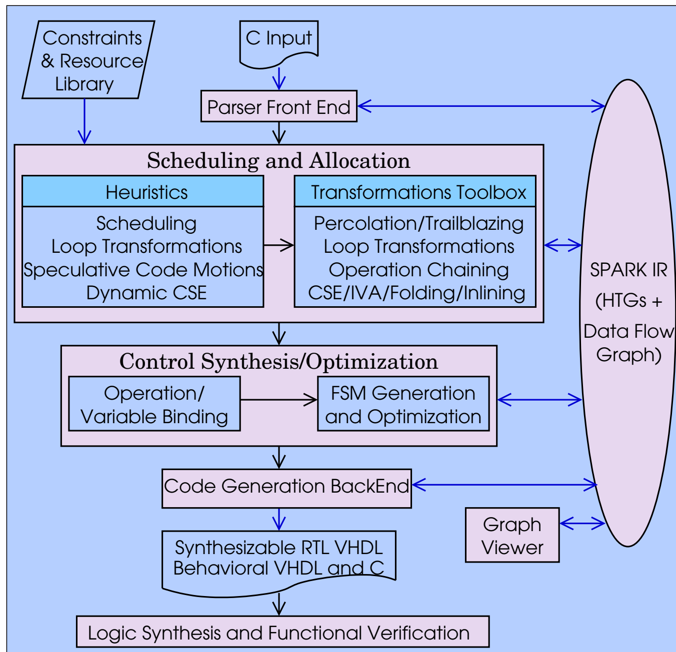

Figure 1. ***The SPARK High-Level Synthesis System***

fects on schedule lengths [5]. CVLS [1] uses condition vectors to improve resource sharing among mutually exclusive operations. Radivojevic et al [2] present an exact symbolic formulation which generates an ensemble schedule of valid, scheduled traces. The "Waveschedule" approach [3] minimizes the expected number of cycles by using speculative execution. Santos et al [4] and Rim et al [9] support generalized code motions for scheduling in HLS. Similar code transformation techniques that have been presented in for software (parallelizing) compilers [10] need to be re-instrumented for synthesis to use hardware cost models for operations and resources.

An important limitation of earlier work is the restrictions on the input description that can be synthesized. Also, several systems do not provide a complete design flow from architectural description to final synthesized netlist and present only scheduling results for small, synthetic benchmarks. The SPARK system has been designed to overcome these limitations as explained over the next few sections.

3 The SPARK High Level Synthesis System

The *SPARK* synthesis framework is a modular and extensible high-level synthesis system that provides a number of code transformation techniques. SPARK has been designed to aid in experimenting with new transformations and heuristics that enhance the quality of synthesis results. Figure 1 provides an overview of the SPARK system. The input language for design descriptions is ANSI-C, currently with the restrictions of no pointers and no function recursion. This input description is parsed into a hierarchical intermediate representation described in Section 4.

The core of the synthesis system has a *transformations toolbox* that consists of a set of information gathering passes, basic code motion techniques and several compiler transformations. Passes from the toolbox are called by a set of heuristics that guide how the code refinement takes place. Since the heuristics and the underlying transformations that they use are completely independent, heuristics can be easily tuned by calling different passes in the toolbox.

As shown in Figure 1, the transformations toolbox contains a data dependency extraction pass, parallelizing code motion techniques [11, 12], dynamic renaming of variables, the basic operations of loop (or software) pipelining and some supporting compiler passes such as copy and constant propagation and dead code elimination [13]. The various passes and transformations can be controlled by the designer using scripts, hence, allowing experimentation with different transformations and heuristics.

After scheduling, the system then control synthesis and optimization. *Control synthesis* generates a finite state machine controller and also does resource binding [5]. The back-end of the SPARK system then generates synthesizable RTL VHDL and hence, the SPARK system integrates into the standard synthesis design flow. In the next few sections, we examine the SPARK system in more detail, starting with the internal intermediate representation it uses.

4 HTG: A Model for Control Intensive Designs

The *SPARK* system stores the behavioral description in an intermediate representation (IR) that retains all the information given in the input description. This is critical for enabling source-level transformations, making global decisions about code motion and enabling the visualization of intermediate results to improve user-interaction.

The intermediate representation used in SPARK consists of basic blocks encapsulated in *Hierarchical Task Graphs* (HTGs) [12, 14]. An HTG is a directed acyclic graph that has three types of nodes: *simple* nodes (non-hierarchical nodes), *compound* nodes (nodes that have sub-nodes), and *loop* nodes. Operations that execute concurrently are aggregated together in simple nodes called *statements*. Statements that have no control flow between them are aggregated together into basic blocks. Basic blocks are encapsulated into compound HTG nodes to form hierarchical structures such as if-then-else blocks, switch-case blocks, loop nodes or a series of HTG nodes. Expressions are stored as abstract syntax trees [13] and each operation expression is initially encapsulated in a statement node of its own.

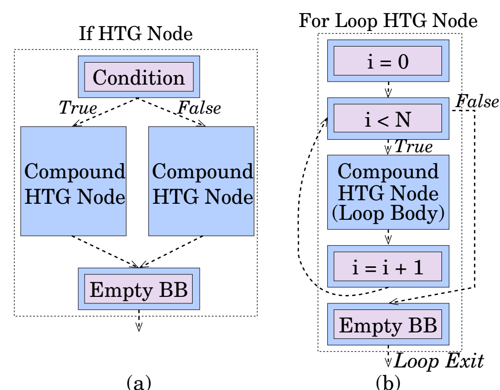

Figure 2. ***The hierarchical task graph (HTG) representation of (a) an if-block, (b) a For-Loop.***

1063-9667/03 $17.00 © 2003 IEEE
IEEE COMPUTER SOCIETY

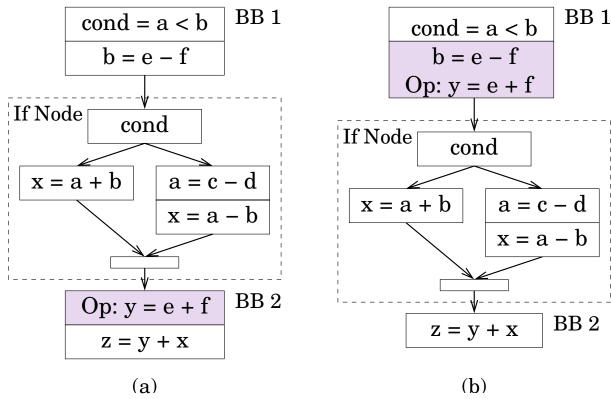

Figure 3. ***Trailblazing: Operation** $op1$ **is moved from basic block** BB_2 **to basic block** BB_1 **across the if-then-else HTG node without visiting each basic block inside the node.***

Figure 2(a) illustrates the HTG for a if-then-else conditional block (the dashed edges indicate control flow). It consists of a basic block each for the condition and for the join and compound HTG nodes for the true and false branches. Similarly, the conceptual representation of a *For-loop HTG* is shown in Figure 2(b). A for-loop HTG consists of basic blocks for the initialization, the conditional check and the loop index increment (optional) and a compound HTG node for the loop body.

An important feature of HTGs is that they are *strongly connected components* [14]; i.e., they have a single entry and a single exit point. This property enables HTGs to be used to encapsulate complex loops and irregular regions of code, to regularize code motion techniques and reduce the amount of patch-up code inserted as explained next.

5 Code Motion Techniques in SPARK

The code motion techniques implemented in the toolbox of the SPARK system are percolation scheduling and trailblazing. *Percolation Scheduling* (PS) was developed as a technique to target code to parallel architectures such as VLIWs and vector processors [11]. Percolation scheduling compiles programs into parallel code by systematically applying semantic preserving transformations. These transformations have been proven to be complete with respect to the set of all possible local, dependency-preserving transformations on program trees.

However, to move an operation from a node A to node B, percolation requires a visit to each node on every control path from A to B. The incremental nature of these linear operation moves cause code explosion by unnecessarily duplicating operations and inserting copy operations. Trailblazing was proposed to circumvent these problems.

Trailblazing is a code motion technique that exploits the hierarchical structuring of the input description's operations and global information in HTGs to make non-incremental operation moves without visiting every operation that is bypassed [12]. At the lowest level, trailblazing is able to perform the same fine-grained transformations as percolation. However, at a higher level, trailblazing is able to move operations across large blocks of code.

While an operation is being moved using trailblazing,

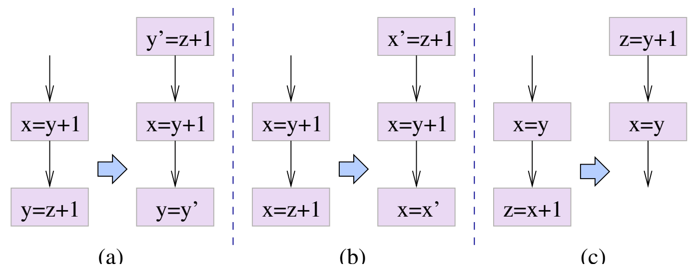

Figure 4. ***Moving one operation across another operation while eliminating (a) an anti dependency (b) an output dependency and (c) a flow dependency.***

the algorithm moves the operation across HTG nodes that it comes across if the moving operation has no dependencies with the HTG node. Hence, in the example in Figure 3, the operation $Op : y = e + f$ can be moved from basic block BB_2 to BB_1, across the if-then-else HTG node, since it has no data dependencies with any of the operations in this if-node. The resultant code is shown in Figure 3(b). To perform the same code motion, percolation would have duplicated Op into both the branches of the if-block and visited each node in the if-block before finally unifying the copies at the conditional check.

6 Dynamic Renaming

There are four types of data dependencies [15]: flow (variable read after write), anti (write after read), output (write after write) and input (read after read). Several previous high-level synthesis works maintain only flow dependencies in control-data flow graphs (CDFGs). The variable names from the original source are discarded. However, this hinders the visualization of intermediate results of transformations applied to the input description. Hence, the SPARK system retains the complete information about variables used in the input description in data dependency graphs that maintain all the data dependency types.

However, non-flow dependencies that prevent code motions can often be resolved by *dynamic renaming* and *combining* [16]. Figures 4(a) to (c) demonstrate how one operation can be moved past another one while dynamically eliminating data dependencies. In Figure 4(a), an anti dependency can be resolved during scheduling by moving only the right hand side of the operation $y = z + 1$. The result is written to a new destination variable y' and the original operation is replaced by the copy operation, $y = y'$. Similarly, in Figure 4(b), an output dependency between two operations that write to the same variable x, can be resolved in a similar manner by creating a new destination variable x'. These copy operations introduced by dynamic renaming, can also be circumvented by a technique known as *combining*. Combining replaces the copy in the operation being moved by the variable being copied. This is demonstrated in Figure 4(c), where the operation $z = x + 1$ is moved past the copy operation $x = y$. The variable x is replaced with the variable y in the moving operation.

Dynamic renaming and combining can lead to considerable easing of the constraints imposed by data dependencies and enable the set of speculative code motions discussed next to be more effective.

IEEE COMPUTER SOCIETY

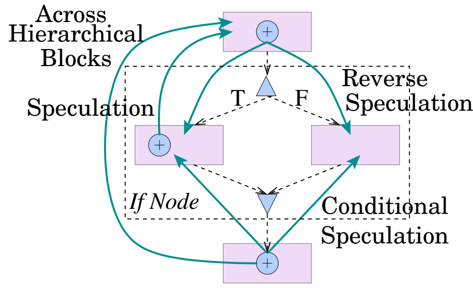

Figure 5. ***The various speculative code motions***

7 Speculative Code Motions

An overview of the various speculative code motions is shown in Figure 5. Operations may be moved out of conditionals and executed *speculatively*, or operations before conditionals may be moved into *subsequent* conditional blocks and executed conditionally by *reverse speculation*, or an operation from after the conditional block may be duplicated *up* into *preceding* conditional branches and executed conditionally by *conditional speculation* [5, 6]. Operations can also be moved across entire hierarchical blocks, such as if-then-else blocks or loops. Reverse speculation can be coupled with *early condition execution* that evaluates conditional checks as soon as possible. Since these code motions re-order, speculate and duplicate operations, they often create new opportunities for dynamically applying transformations such as common sub-expression elimination during scheduling as discussed in the next section.

8 Dynamic 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 [13]. Hence, for the example in Figure 6(a), the common sub-expression $b+c$ in operation 2 can be replaced with the result of operation 1, as shown in Figure 6(b).

Now consider that for the example in Figure 6(a), the scheduling heuristic decides to schedule operation 3 in BB_1 and execute it speculatively as operation 5, as shown in Figure 6(b). The result of the speculated operation 5 can then be used to replace the common sub-expression in operation 4 as shown in Figure 6(b). Hence, to exploit these new opportunities created by speculative code motions, CSE has to be applied during scheduling rather than the traditional

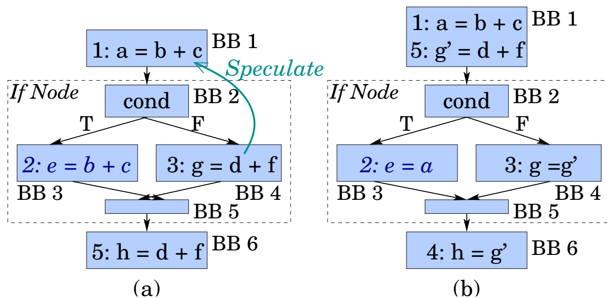

Figure 6. ***(a) A sample HTG (b) CSE replaces*** $b+c$ ***in operation*** 2 ***with the variable*** a ***from op.*** 1***. After operation*** 3 ***is executed speculatively as operation*** 5 ***in*** BB_1***, dynamic CSE eliminates the common sub-expression in op.*** 4

Algorithm 1: Scheduling Heuristic with Dynamic CSE
Inputs: Unscheduled HTG of design, Resource List R
Output: Scheduled HTG of design
1: Scheduling step $step = 0$
2: **while** ($step \neq$ last step of HTG) **do**
3: **foreach** (resource res in Resource List R) **do**
4: Get List of Available Operations $\mathcal{A}$
5: Pick Operation op with lowest cost in $\mathcal{A}$
6: Move op and schedule on res in $step$
7: ApplyDynamicCSE(op, $\mathcal{A}$)
8: **endforeach**
9: $step = step + 1$
10: **endwhile** (a)

Algorithm 2: Get List of Available Operations
Inputs: Resource res, Scheduling $step$, $AllowedCMs$
Output: Available Operations List $\mathcal{A}$
1: Candidates $\mathcal{A}$ = all unscheduled ops U in HTG that can be scheduled on resource res
2: **foreach** (op in $\mathcal{A}$) **do**
3: **if** (data dependencies of op cannot be satisfied) OR
4: (op cannot be moved to $step$ using $AllowedCMs$)
5: Remove op from $\mathcal{A}$
6: Calculate cost of operation op
7: **endforeach** (b)

Figure 7. ***(a) Priority-based List Scheduling Heuristic (b) Determining the list of Available operations.***

approach of applying it as a pass before scheduling. We call this new approach of applying CSE while scheduling an operation, *Dynamic CSE*. Conceptually, dynamic CSE finds and eliminates operations in the list of remaining ready-to-be-scheduled operations that have a common sub-expression with the currently scheduled operation. Applying CSE as a pass *after* scheduling is not as effective as dynamic CSE, since the resource freed up by eliminating an operation during scheduling can potentially be used to schedule another operation by the scheduler.

Dynamic CSE has been shown to significantly improve synthesis results when applied with speculative code motions, particularly code motions such as reverse and conditional speculation that duplicate operations in the HTG [7].

9 Priority-based List Scheduling Heuristic

In this section, we present a scheduling heuristic that schedules the HTG of the design using the speculative code motions and the dynamic CSE transformation. This *Priority-based Global List Scheduling* heuristic is presented in Figure 7(a). The inputs to this heuristic are the unscheduled HTG of the design and the list of resource constraints. Additionally, the designer may specify a list of allowed code motions, *AllowedCMs* (i.e. speculation, conditional speculation et cetera), whether dynamic variable renaming is allowed, and the code motion technique (percolation or trailblazing) for moving the operations. The heuristic starts by assigning a priority to each operation in the input description based on the length of the dependency chain of operations that depend on it [6].

Scheduling is done one control or scheduling step at a

1063-9667/03 $17.00 © 2003 **IEEE**

IEEE COMPUTER SOCIETY

Transformation Applied	**pred2: 217 Ops, 45 BBs** 3 ALU,2[ ],3<<,2==,1*		**pred0:101 Ops,26 BBs** 3 ALU,2[ ],3<<,2==,1*		**polar:252 Ops,78 BBs** 2+,3-,1/,2*,2<<,2==		**tiler: 145 Ops,35 BBs** 3+,2-,1/,2*,2<<,2==,1[ ]	
	# States	Long Path	# States	Long Path	# States	Long Path	# States	Long Path
Across hier blocks	98	4877	104	2268	50	50	69	6531
+speculation	76(-22%)	4045(-17%)	79(-24%)	1883(-17%)	46(-8%)	46(-8%)	59(-15%)	5431(-17%)
+early cond exec	67(-12%)	3469(-14%)	70(-13%)	1627(-14%)	43(-6.5%)	43(-6.5%)	58(-2%)	5331(-2%)
+cond speculation	48(-28%)	2260(-35%)	51(-27%)	1051(-35%)	41(-4.7%)	41(-4.7%)	34(-41%)	3031(-43%)
all CMs+CSE only	46(-4.2%)	2188(-3.2%)	49(-3.9%)	987(-6.1%)	41(-0%)	41(-0%)	27(-21%)	2231(-26%)
+ Dyn.CSE only	41(-15%)	1676(-26%)	44(-14%)	731(-30%)	37(-9.8%)	37(-9.8%)	26(-24%)	2131(-30%)
+ CSE+Dyn.CSE	39(-19%)	1676(-26%)	42(-18%)	731(-30%)	37(-9.8%)	37(-9.8%)	26(-24%)	2131(-30%)
Total Reduction	**-60.2 %**	**-65.6 %**	**-59.6 %**	**-67.8 %**	**-43.3 %**	**-43.3 %**	**-62.3 %**	**-67.4 %**

Table 1. ***Scheduling results for the various code motions and application of CSE and Dynamic CSE for the functions from the MPEG-1 Prediction block and the GIMP Image Processing Tool***

time while traversing the basic blocks in the design's HTG. Within a basic block, each scheduling step corresponds to a statement HTG node (see Section 4). At each scheduling step in the basic block, for each resource in the resource list, a list of *available* operations is collected,

Available operations is a list of operations that can be scheduled on the given resource at the current scheduling step. Pseudo-code for collecting the list of available operations is given in Figure 7(b). Initially, all unscheduled operations in the HTG that can be scheduled on the current resource type are added to the available operations list. Subsequently, operations whose data dependencies are not satisfied and cannot be satisfied by dynamic variable renaming, and operations that cannot be moved in the HTG to schedule them onto the current scheduling step using the allowed code motions, are removed from the available list. The remaining operations are assigned a cost based on the length of the dependency chain leading up to the operation [6].

The scheduling heuristic then picks the operation with the *lowest* cost from the available operations list as shown in line 5 of Figure 7(a). The code motion technique (trailblazing) is then instructed to schedule this operation at the current scheduling step. This is repeated for all resources in each scheduling step in the HTG.

Once the chosen operation has been scheduled, the dynamic CSE heuristic finds and eliminates common subexpressions in the operations in the available list, if the new position of the scheduled operation *op* permits. This heuristic is discussed in detail in [7].

10 Experimental Setup and Results

We have implemented the scheduling heuristic presented in the previous section in the SPARK framework. In this section, we present results of experiments performed using two large and moderately complex real-life applications. These designs consist of the *pred*0_1 and *pred*2 functions from the *Prediction* block of the MPEG-1 algorithm [18] and the *calc_undistorted_coords* from the "polarize" transform and tile function (with scale inlined) from the "tiler" transform from the GIMP image processing tool [19]. The run time of SPARK for these designs is less than 5 user seconds on a 1.6 Ghz PC running Linux.

The synthesis results for the 4 functions from these benchmarks are presented in Table 1. These results are in terms of the number of states in the FSM controller and the cycles on the longest path. Due to shortage of space, we are unable to present results for the average case of execution cycles. The resources are indicated in the tables; all resources are single cycle except for the multiplier (2 cycles) and the divider (4 cycles).

The first four rows in Table 1 present results with by incrementally allowing code motions: in the first row, code motions are allowed only within basic blocks and across hierarchical blocks. The second row allows speculation, the third row has reverse speculation and early condition execution enabled as well and the fourth row has the conditional speculation also enabled. The percentage reductions of each row over the previous row are given in parentheses.

The 5th to 7th rows in Table 1 present synthesis results for when only CSE is applied as a pass before scheduling (5th row), when only dynamic CSE is applied during scheduling (6th row) and finally, both CSE and dynamic CSE are applied (7th row). The percentage reductions of each row over the *fourth* row (all code motions enabled but no CSE applied) are also given in parentheses.

The results in this table demonstrate that the various code transformations lead to a significant improvement in the performance and controller size of all the designs. Some transformations are more effective than others; however, it is clear that CSE alone is not nearly as effective as dynamic CSE – for all the designs, dynamic CSE is able to eliminate many more operations than CSE alone.

The total reductions obtained when all the code motions are enabled and CSE and dynamic CSE are applied are summarized in the last row of Table 1. These improvements range from 43 to 67 %. These improvements further add up over the whole design, since several of these design blocks are called from within loops.

10.1 Logic Synthesis Results

We synthesized the RTL VHDL from the various designs using the Synopsys *Design Compiler* logic synthesis tool. The LSI-10K synthesis library was used for technology mapping. The results for the various transformations for the MPEG functions are presented in the graphs in Figures 8 and 9, in terms of three metrics: the critical path length (in ns), the unit area and the maximum delay through the design (longest path cycles $*$ critical path length).

The results in Figure 8 correspond to the first four rows

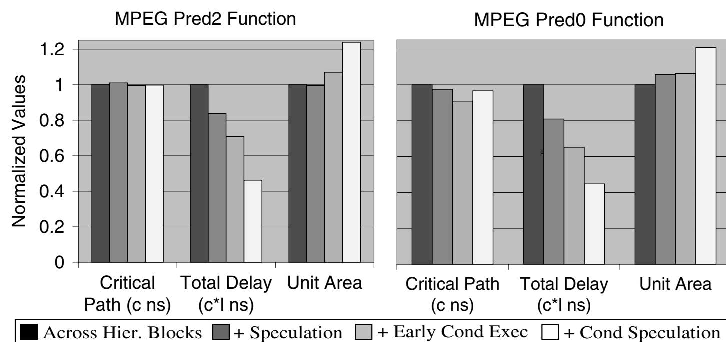

Figure 8. ***Effects of the various code motions on logic synthesis results for the MPEG** $Pred2$ **and** $Pred0_1$ **functions***

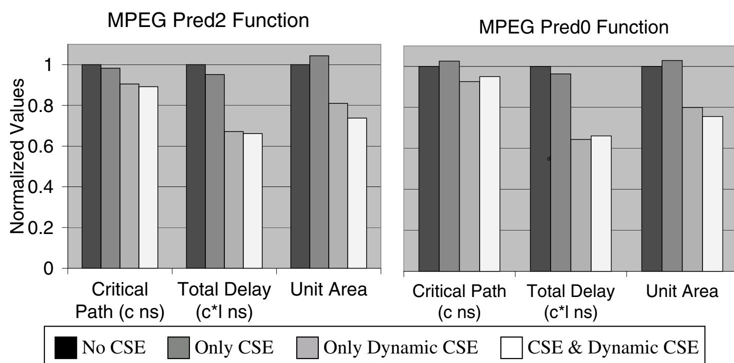

Figure 9. ***Effects of CSE and dynamic CSE on logic synthesis results for the MPEG** $Pred2$ **and** $Pred0_1$ **functions***

from Table 1 with the various code motions enabled. Similarly, Figure 9 presents results for CSE and dynamic CSE and the bars correspond to the 4th to 7th rows in Table 1. The values are normalized with respect to the first bar.

The results in Figure 8 demonstrate that the improvements in performance seen earlier in Table 1 translate over to the synthesis results of the final netlist. Hence, the total delay through the circuit reduces by up to 50 % as the code motions are enabled with negligible impact on critical path length. However, these aggressive code motions do lead to an increase in area by about 20 % when all the code motions are enabled. This area increase is due to the additional steering and control logic caused by increased resource sharing.

However, the results in Figure 9 demonstrate that dynamic CSE leads to lower area due to fewer operations and the corresponding reductions in interconnect. This ends up counterbalancing the increases in area seen due to the code motions. The results from these graphs demonstrate that enabling all the code motions along with CSE and dynamic CSE, leads to overall reductions in the total delay through the circuit by nearly 70 % when compared to only within basic block code motions, while the design area remains almost constant (and sometimes even decreases).

11 Conclusions and Future Directions

In this paper, we presented the SPARK high-level synthesis framework that provides an integrated flow from architectural specification in behavioral C to logic synthesis of the output RTL VHDL. This framework provides a platform for applying a range of coarse-grain and fine-grain code optimizations aimed at improved synthesis results. We presented the design flow through the system along with a description of the various passes in the system. We outlined a list scheduling heuristic that incorporates transformations such as speculative code motions and dynamic transformations such as dynamic CSE. Scheduling results after applying the code transformations on moderately complex real-life benchmarks show significant improvements in performance and reduction in controller size, while maintaining design area and critical path lengths almost constant. In on-going work, we are exploring a set of loop transformations.

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 et al. Incorporating speculative execution into scheduling of control-flow intensive behavioral descriptions. In *DAC*, 1998.

[4] L.C.V. dos Santos and J.A.G. Jess. A reordering technique for efficient code motion. In *DAC.*, 1999.

[5] S. Gupta et al. Conditional speculation and its effects on performance and area for HLS. In *ISSS*, 2001.

[6] S. Gupta et al. Speculation techniques for high level synthesis of control intensive designs. In *DAC*, 2001.

[7] S. Gupta et al. Dynamic common sub-expression elimination during scheduling in HLS. In *ISSS*, 2002

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

[9] M. Rim et al. Global scheduling with code-motions for high-level synthesis applications. *IEEE Trans. on VLSI Systems*, Sept. 1995.

[10] J. Fisher. Trace scheduling: A technique for global microcode compaction. *Trans. on Computers*, July 1981.

[11] A. Nicolau. A development environment for scientific parallel programs. CS-TR 86-722, Cornell Univ., 1985.

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

[13] A. Aho et al. *Compilers: Principles and Techniques and Tools*. Addison-Wesley, 1986.

[14] M. Girkar and C.D. Polychronopoulos. Automatic extraction of functional parallelism from ordinary programs. *IEEE Trans. on PDS*, Mar. 1992.

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

[16] S.-M. Moon and K. Ebcioglu. An efficient resource-constrained global scheduling technique for superscalar and VLIW processors. *Intl. Symp. on Microarch.*, 1992.

[17] V.C. Sreedhar et al. A new framework for exhaustive and incremental data flow analysis using DJ graphs. *SIGPLAN Conf. on PLDI*, 1996.

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

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

IEEE COMPUTER SOCIETY