Coordinated Parallelizing Compiler Optimizations and High-Level Synthesis

SUMIT GUPTA
Tallwood Venture Capital
RAJESH KUMAR GUPTA
University of California, San Deigo
NIKIL D. DUTT
University of California, Irvine
and
ALEXANDRU NICOLAU
University of California, Irvine

We present a high-level synthesis methodology that applies a coordinated set of coarse-grain and fine-grain parallelizing transformations. The transformations are applied both during a presynthesis phase and during scheduling, with the objective of optimizing the results of synthesis and reducing the impact of control flow constructs on the quality of results. We first apply a set of source level presynthesis transformations that include common sub-expression elimination (CSE), copy propagation, dead code elimination and loop-invariant code motion, along with more coarse-level code restructuring transformations such as loop unrolling. We then explore scheduling techniques that use a set of aggressive speculative code motions to maximally parallelize the design by re-ordering, speculating and sometimes even duplicating operations in the design. In particular, we present a new technique called "Dynamic CSE" that dynamically coordinates CSE and code motions such as speculation and conditional speculation during scheduling. We implemented our parallelizing high-level synthesis in the *SPARK* framework. This framework takes a behavioral description in ANSI-C as input and generates synthesizable register-transfer level VHDL. Our results from computationally expensive portions of three moderately complex design targets, namely, MPEG-1, MPEG-2 and the GIMP image processing tool, validate the utility of our approach to the behavioral synthesis of designs with complex control flows.

Categories and Subject Descriptors: B.5.1 [**Register-Transfer-Level Implementation**]: Design Aids

This project was funded by the Semiconductor Research Corporation (SRC) under Task I.D. 781.001.
Author's addresses: S. Gupta, Tallwood Venture Capital, 635 Waverley St., Palo Alto, CA 94301; email: sumit@tallwoodvc.com; R. Gupta, Department of Computer Science and Engineering, University of California, San Diego, AP&M 3111, 9500 Gilman Drive, La Jolla, CA 92093-0114; email: rgupta@ucsd.edu; N. D. Dutt and A. Nicolau, Center for Embedded Computer Systems, 444 CS, University of California, Irvine, CA 92697-3425; email: {dutt,nicolau}@ics.uci.edu.

General Terms: Algorithms, Design, Performance

Additional Key Words and Phrases: Code motions, common subexpression elimination, embedded systems, dynamic CSE, high-level synthesis, parallelizing transformations, presynthesis

1. INTRODUCTION

Driven by the increasing size and complexity of digital designs, there has been a renewed interest in high-level synthesis of digital circuits from behavioral descriptions both in the industry and in academia [Wakabayashi 1999; Get2Chip; Forte; Celoxica; dos Santos 1998; Haynal 2000; Lakshminarayana et al. 1999; Gupta et al. 2004]. A key change that has taken place since high-level synthesis was first explored two decades ago is the widespread acceptance and use of register-transfer level (RTL) language modeling of digital designs. In fact, recent years have seen the use of variants of programming languages such as "C" and "C++" for behavioral level modeling. High-level synthesis and verification tools are essential for enabling widespread industrial adoption of these system-level programming paradigms.

However, there are several challenges that limit the utility and wider acceptance of high-level synthesis. There is a loss of control on the size and quality of the synthesized result. High-level languages allow for additional freedom in the way a behavior is described compared to register-transfer level descriptions. Thus, the style of high-level programming—in particular, the overall control flow and choice of control flow constructs—often has an unpredictable impact on the final circuit. Thus, we need techniques and tools that achieve the best compiler optimizations and synthesis results irrespective of the programming style used in the high-level descriptions.

Our approach is a *parallelizing* high-level synthesis methodology that is outlined in Figure 1. It includes a presynthesis phase that makes available a number of transformations to restructure a design description. These include transformations to reduce the number of operations executed such as common subexpression elimination (CSE), copy propagation, dead code elimination and loop-invariant code motion [Aho et al. 1986]. Also, we use coarse-level loop transformation techniques such as loop unrolling to increase the scope for applying parallelizing optimizations in the scheduling phase that follows.

The scheduling phase employs an innovative set of speculative, beyond-basic-block code motions that reduce the impact of the choice of control flow (conditionals and loops) on the quality of synthesis results. These code motions enable movement of operations through, beyond, and into conditionals with the objective of maximizing performance. Since these speculative code motions often re-order, speculate and duplicate operations, they create new opportunities to apply additional transformations "dynamically" during scheduling such as dynamic common sub-expression elimination. These compiler transformations are integrated with the standard high-level synthesis techniques such as resource sharing, scheduling on multi-cycle operations and operation chaining. Once a design has been scheduled, in the next step of the methodology in Figure 1, we use a resource binding and control generation pass, followed by a back-end code

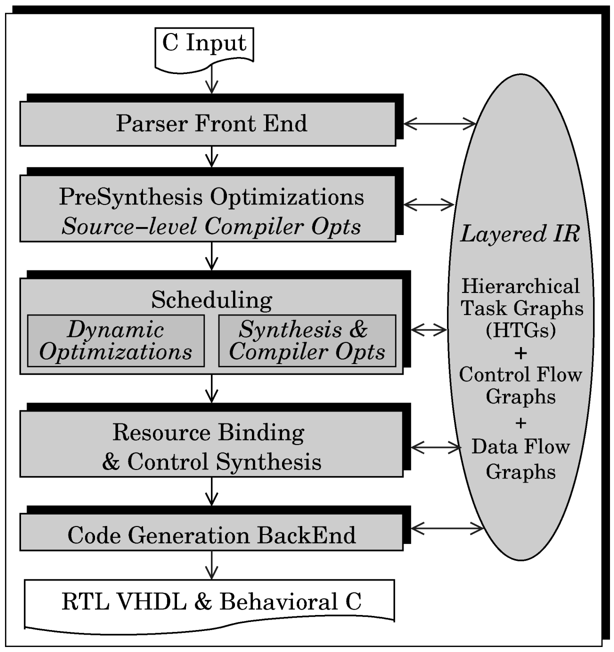

Fig. 1. An overview of the proposed high-level synthesis flow incorporating compiler transformations during the presynthesis source-to-source transformation phase and the scheduling phase.

generator that can interface with standard logic synthesis tools to generate the gate level net-list.

Given the maturity of high-level synthesis techniques and equally antique compiler techniques, it is natural for the reader to be skeptical about the novelty of the contributions in this work. Several compiler techniques have been tried before for high-level synthesis, albeit with mixed success. This is partly because direct application of traditional compiler techniques does not necessarily optimize hardware synthesis results.

In contrast to high-level synthesis tools, compilers often pursue maximum parallelization by applying parallelizing transformations. For instance, percolation provably exposes maximal parallelism by moving operations across and out of conditional branches [Nicolau 1985]. While this is a useful result, in high-level synthesis, such code transformations have to be selected and guided based on their effects on the control, interconnect and area costs. Indeed, we show that the chief strength of our heuristics is the ability to select the code transformations so as to improve the overall synthesis results. In some cases, this means that we actually end up moving operations into the conditional blocks.

The rest of this article is organized as follows: we first review previous related work. In Section 3, we describe our high-level synthesis methodology, followed by the description of the representation model we use for designs with complex control. In Section 5, we briefly describe the presynthesis transformations. Next, we present the speculative code motion transformations and dynamic CSE and dynamic copy propagation. In Section 7, we present a priority-based list scheduling heuristic that incorporates these transformations, followed by experimental results and a discussion.

2. RELATED WORK

High-level synthesis techniques have been investigated for two decades now. Over the years, several books have discussed the advances in high-level synthesis techniques [Gajski et al. 1992; Camposano and Wolf 1991; De Micheli 1994]. Traditionally the focus of high-level synthesis works has been on data flow designs. Researchers have proposed a range of optimizations such as algebraic transformations [Walker and Thomas 1989], retiming [Potkonjak and Rabaey 1994], use of complex library components [Peymandoust and Micheli 2001], and throughput improvement using bit-level chaining of resources [Park and Choi 2001]. Over the last decade, several groups have started looking at applications with a mix of control and data flow. Speculative execution of operations [Wakabayashi 1999; Radivojevic and Brewer 1996; Lakshminarayana et al. 1999; Rim et al. 1995; dos Santos 1998; Gupta et al. 2001b] and other speculative code motions [Gupta et al. 2001a, 2003a] are particularly effective in improving the schedule length and circuit delay through these designs. Presynthesis transformations for these mixed control-data flow designs have focused on altering the control flow or extracting the maximal set of mutually exclusive operations [Li and Gupta 1997].

On the other hand, compiler transformations such as CSE and copy propagation predate high-level synthesis and are standard in most software compilers [Aho et al. 1986; Muchnick 1997]. These transformations are applied as passes on the input program code and as cleanup at the end of scheduling before code generation. Compiler transformations were developed for improving code efficiency. Their use in digital circuit synthesis has been limited. For instance, CSE has been used for throughput improvement [Iqbal et al. 1993], for optimizing multiple constant multiplications [Potkonjak et al. 1996; Pasko et al. 1999] and as an algebraic transformation for operation cost minimization [Janssen et al. 1994; Miranda et al. 1998].

A converse of CSE, namely, *common sub-expression replication* has been proposed to aid scheduling by adding redundant operations [Lobo and Pangrle 1991]. *Partial redundancy elimination* (PRE) [Kennedy et al. 1999] 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. Janssen et al. [1994] and Gupta et al. [2000] 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* [Novack and Nicolau 1994] performs local optimizations such as CSE during scheduling in an opportunistic, context-sensitive manner.

A range of parallelizing code transformation techniques have been previously developed for high-level language software compilers (especially parallelizing compilers) [Fisher 1981; Ebcioglu and Nicolau 1989; Nicolau and Novack 1993; Novack and Nicolau 1996]. Although the basic transformations (e.g., dead code elimination, copy propagation) can be used in synthesis as well, other transformations need to be re-instrumented for synthesis by incorporating ideas of mutual exclusivity of operations, resource sharing and hardware cost models. Cost models of operations and resources in compilers and synthesis tools are particularly very different. In circuit synthesis, code transformations that lead to increased resource utilization, also lead to higher hardware costs in terms of steering logic and associated control circuits. Some of these costs can mitigated by interconnect aware resource binding techniques [Gupta et al. 2001a].

3. ROLE OF PARALLELIZING COMPILER TRANSFORMATIONS IN HIGH-LEVEL SYNTHESIS

As mentioned in the previous section, recent high-level synthesis approaches have employed beyond-basic-block code motions such as speculation—derived from the compiler domain—to increase resource utilization. In previous work, we presented a comprehensive and innovative set of speculative code motions that go beyond the traditional compiler code motions. We demonstrated their usefulness in reducing the impact of the choice of control flow in the input description on the quality of synthesis results [Gupta et al. 2001a, 2001b 2003a].

In this article, we propose a *parallelizing* high-level synthesis methodology that incorporates these and several other techniques derived from the compiler domain, particularly, from parallelizing compilers. However, we propose using these compiler techniques not only during the traditional scheduling phase of high-level synthesis, but also, during a *presynthesis* phase in which coarse-grain transformations are applied to the input description before performing high-level synthesis. We built the *Spark* high-level synthesis framework to implement this methodology. An overview of the *Spark* framework is shown in Figure 2. *Spark* takes a behavioral description in ANSI-C as input and additional inputs in the form of a hardware resource library, resource and timing constraints and user directives for the various heuristics and transformations.

There are a few restrictions on the input C: we do not support pointers (arrays are supported), unstructured jumps (gotos) and function recursion. Each function in the input description is mapped to a (concurrent) hardware block. If one function calls another function, then the called function is instantiated as a component in the calling function. Currently, we have not implemented support for *continue* and *break* statements, although it is always possible to convert a program with continues and breaks into a program without them [Girkar and Polychronopoulos 1992]. *Switch* statements are reduced to a series of if-then-else statements. We support all types of loops; this is explained in more detail in Section 8.1.

The transformations in the *presynthesis phase* include (a) coarse-level code restructuring by function inlining and loop transformations (loop unrolling, loop fusion et cetera), (b) transformations that remove unnecessary and redundant operations such as common subexpression elimination (CSE), copy propagation, and dead code elimination (c) transformations such as loop-invariant code motion, induction variable analysis (IVA) and operation strength reduction,

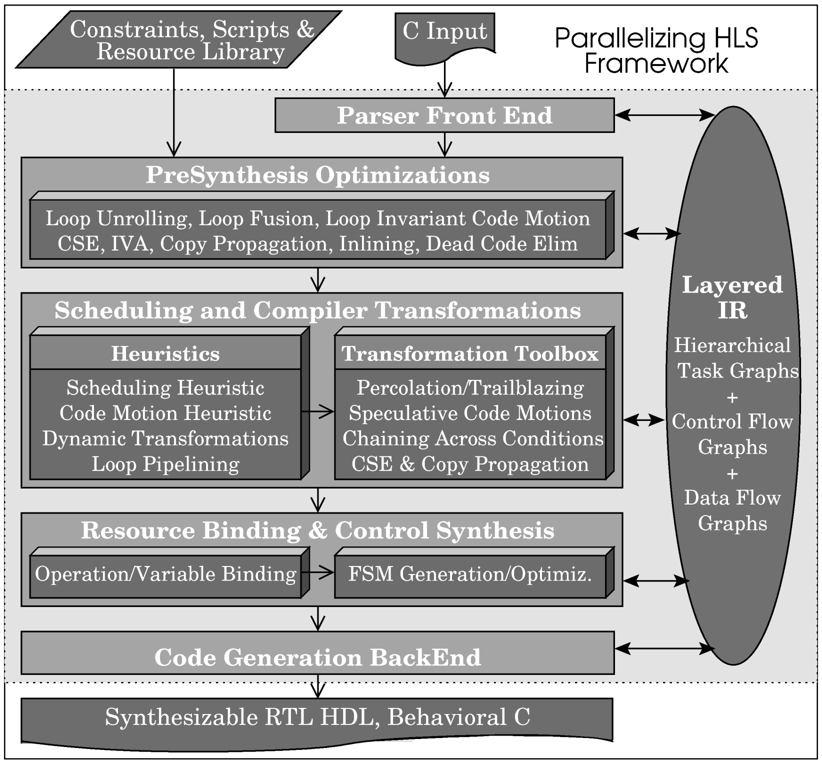
Constraints, Scripts & Resource Library
C Input
Parallelizing HLS Framework
Parser Front End
PreSynthesis Optimizations
Loop Unrolling, Loop Fusion, Loop Invariant Code Motion
CSE, IVA, Copy Propagation, Inlining, Dead Code Elim
Scheduling and Compiler Transformations
Heuristics
Scheduling Heuristic
Code Motion Heuristic
Dynamic Transformations
Loop Pipelining
Transformation Toolbox
Percolation/Trailblazing
Speculative Code Motions
Chaining Across Conditions
CSE & Copy Propagation
Resource Binding & Control Synthesis
Operation/Variable Binding
FSM Generation/Optimiz.
Code Generation BackEnd
Synthesizable RTL HDL, Behavioral C
Layered IR
Hierarchical Task Graphs
+
Control Flow Graphs
+
Data Flow Graphs

Fig. 2. An overview of the *spark* high-level synthesis framework.

that reduce the number of operations within loops and replace expensive operations (multiplications and divisions) with simpler operations (shifts, additions and subtractions).

The presynthesis phase is followed by the scheduling and allocation phase (see Figure 2). In our current approach, we assume the designer has done the module selection and resource allocation and given us a hardware resource library that describes the type and number of each resource. Thereafter, the scheduler in *Spark* does resource constrained scheduling. The scheduler is organized into two parts: the heuristics that perform scheduling and a toolbox of synthesis and compiler transformations. This allows the heuristics to employ the various transformations as and when required, thus enabling a modular approach that allows the easy development of new heuristics.

The synthesis transformations in the scheduler toolbox include chaining operations across conditional blocks [Gupta et al. 2002a], scheduling on multicycle operations, resource sharing et cetera [De Micheli 1994]. Besides, the traditional high-level synthesis transformations, the scheduling phase also employs several compiler transformations applied “dynamically” during scheduling. These dynamic transformations are applied either to aid scheduling, such as speculative code motions, or to exploit the new opportunities created by scheduling decisions, such as dynamic CSE and dynamic copy propagation [Gupta et al. 2002b]. Scheduling in *Spark* is done by a priority-based global list scheduling heuristic. This heuristic employs the transformations from the toolbox and code motion techniques such as *Trailblazing* that efficiently move operations in designs with a mix of data and control flow [Gupta et al. 2003b; Gupta et al. 2004].

The scheduling phase is followed by a resource binding and control generation phase. Our resource binding approach aims to minimize the interconnect between functional units and registers [Gupta et al. 2001a]. The control generation pass generates a finite state machine (FSM) controller that implements the schedule. Finally, a back-end code generation pass generates register-transfer level (RTL) VHDL. This RTL VHDL is synthesizable by commercial logic synthesis tools, hence, completing the design flow path from architectural design to final design netlist. Additionally, we also implemented back-end code generation passes that generate ANSI-C and behavioral VHDL. These behavioral output codes represent the scheduled and optimized design. The output "C" can be used in conjunction with the input "C" to perform functional verification and also to enable better user visualization of how the transformations applied by *Spark* affect the design.

Several of the transformations from the presynthesis phase and the scheduling phase implemented in the *Spark* framework are discussed in the following sections. However, to enable the various coarse and fine-grain transformations employed by *Spark*, we require an intermediate representation that maintains the structural information about the design, as explained in the next section.

4. MODELING DESIGNS WITH COMPLEX CONTROL FLOW

In the past, control-data flow graphs (CDFGs) [Gajski et al. 1992; Orailoglu and Gajski 1986] have been primary model for capturing design descriptions for high-level synthesis. CDFGs work very well for traditional scheduling and binding techniques. However, in order to enable the range of optimizations explored by our work—particularly, source-to-source optimizations and other coarse grain transformations—the *Spark* framework uses an intermediate representation that maintains the hierarchical structuring of the design such as if-then-else blocks and for and while loops. This intermediate representation consists of basic blocks encapsulated in *Hierarchical Task Graphs* (HTGs) [Girkar and Polychronopoulos 1992; Gupta et al. 2004].

HTGs retain coarse, high level information about program structure and are maintained *in addition to* control flow graphs (CFGs) that maintain the control dependencies between basic blocks and data flow graphs (DFGs) that maintain the data dependencies between operations [Gajski et al. 1992]. Thus, whereas CFGs (and CDFGs) are efficient for traversing the basic blocks in a design, HTGs enable higher order manipulation—for example, they enable coarse-grain code restructuring (such as that done by loop transformations [Novack and Nicolau 1996]) and also provide an efficient way to move operations across large pieces of code [Nicolau and Novack 1993].

Of course, several other representation models have been proposed earlier for high-level synthesis [McFarland 1978; Brayton et al. 1988; Chaiyakul et al. 1992; Ku and Micheli 1990; Kountouris and Wolinski 1999; Rim et al. 1995; Bergamaschi 1999]. However, we found HTGs to be a convenient representation for designs with considerable control constructs and the most natural choice for our parallelizing transformations [Nicolau and Novack 1993; Novack and Nicolau 1996].

Note that, *basic blocks* are an aggregation of a sequence of statements or operations from the input description with no conditionals or loops. Also, whereas the input "C" description consists only of operations that execute sequentially, the high-level synthesis scheduler can schedule operations to execute concurrently. We aggregate operations that execute concurrently into *scheduling steps* within basic blocks. These scheduling steps correspond to control steps in high-level synthesis [Gajski et al. 1992] and to VLIW instructions in compilers [Girkar and Polychronopoulos 1992].

4.1 HTGs: A Hierarchical Intermediate Representation for Control-Intensive Designs

We define a hierarchical task graph as follows:

Definition 4.1. A *hierarchical task graph* is a hierarchy of directed acyclic graphs $G_{HTG}(V_{HTG}, E_{HTG})$, where the vertices $V_{HTG} = \{htg_i \mid i = 1, 2, \ldots, n_{htgs}\}$ can be one of three types (we use the terms nodes and vertices interchangeably):

(1) *Single nodes* represent nodes that have no sub-nodes and are used to encapsulate basic blocks. Basic blocks are a sequential aggregation of operations that have no control flow (branches) between them.
(2) *Compound nodes* are recursively defined as HTGs, that is, they contain other HTG nodes. They are used to represent structures like if-then-else blocks, switch-case blocks or a series of HTGs.
(3) *Loop nodes* are used to represent the various types of loops (*for, while-do, do-while*). Loop nodes consist of a loop head and a loop tail that are single nodes and a loop body that is a compound node.

The edge set E_{HTG} in G_{HTG} represents the flow of control between HTG nodes. An edge (htg_i, htg_j) in E_{HTG}, where $htg_i, htg_j \in V_{HTG}$, signifies that htg_j executes after htg_i has finished execution. Each node htg_i in V_{HTG} has two distinguished nodes, htg_{Start}(i) and htg_{Stop}(i), belonging to V_{HTG} such that there exists a path from $htg_{Start}(i)$ to every node in htg_i and a path from every node in htg_i to $htg_{Stop}(i)$.

The htg_{Start} and htg_{Stop} nodes for all compound and loop HTG nodes are always single nodes. The htg_{Start} and htg_{Stop} nodes of a loop HTG node are the loop head and loop tail respectively and those of a single node are the node itself. For the rest of this article, we will denote the top-level HTG corresponding to a design as the *Design HTG*, G_{HTG}. The design HTG is constructed by creating a compound node corresponding to each control construct in the design. Detailed notes on HTG construction are presented in Girkar and Polychronopoulos

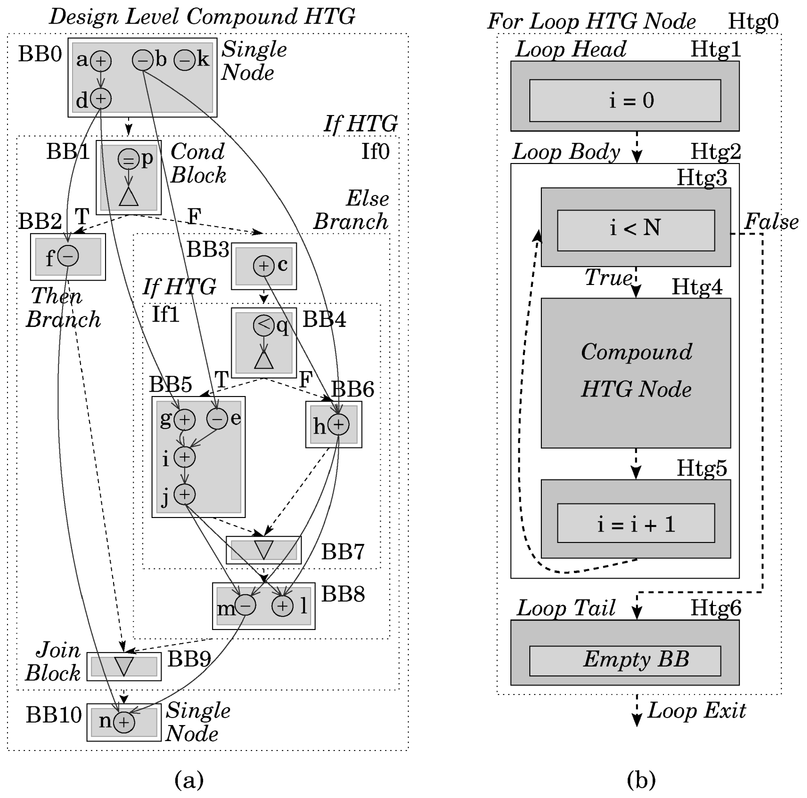

Fig. 3. (a) The hierarchical task graph (HTG) representation of the "waka" benchmark. Flow data dependencies are also shown. (b) The HTG representation of a For-Loop.

[1992]. Note that, we denote the control and data flow graphs for the design by G_{CFG} and G_{DFG} respectively. Together with the design HTG G_{HTG}, these form the design graph $\mathcal{DG}$ [Gupta et al. 2003b].

Figure 3(a) illustrates the HTG for the synthetic benchmark "waka" [Wakabayashi and Tanaka 1992]; the data flow graph has also been superimposed on this HTG representation. Basic blocks are also shown by shaded boxes within the HTG nodes (bb_0 to bb_{10}) and operations are denoted by circular nodes with the operator sign within (operations a to n). Dashed lines denote control flow between HTG nodes and solid lines denote data flow between operations. A fork in the control flow (i.e., a Boolean condition check) is denoted by a triangle ($\triangle$) and a merge by an inverted triangle ($\bigtriangledown$). This design contains an If-HTG node, whose false/else branch contains another If-HTG node. As shown in this figure, an if-then-else HTG consists of a single node for the conditional check, compound HTGs for the true and false branches and a single node with an empty basic block for the *Join* node. The htg_{Start} node for an If-HTG is the conditional check single node and the htg_{Stop} node is the Join node.

The HTG representation for a For-loop HTG is illustrated in Figure 3(b). The For-loop HTG, htg_0, consists of three subnodes: (a) *Loop head* (htg_1): Consists of a single node with an optional initialization basic block: (b) *Loop body* (htg_2):

A compound HTG node containing a single HTG node (htg_3) for the conditional check basic block and a compound HTG node (htg_4) for the main body of the loop, and an optional single node (htg_5) for the loop index increment basic block; and (c) *Loop tail/exit* (htg_6): A single node with an empty basic block. There is a backward control flow edge from the end of the loop body to the conditional check single node. Maintaining the loop hierarchy allows us to treat the back edges as implicit self-loops on composite nodes [Girkar and Polychronopoulos 1992]. Therefore, at any hierarchy level, the HTG is a directed acyclic graph.

Code motion techniques such as *Trailblazing* [Nicolau and Novack 1993] can take advantage of the hierarchical structuring in HTGs to move operations across large pieces of code. For example, when the htg_{Stop} node of a HTG node is encountered while moving an operation, Trailblazing can move the operation directly to the htg_{Start} node of the HTG node without visiting each node in the HTG—provided the operation does not have any data dependencies with the operations in the HTG node [Nicolau and Novack 1993; Gupta et al. 2003b].

For clarity, in the rest of this article, we make several simplifications in the figures used for the examples. We omit the single HTG node that encapsulates basic blocks. Control flow edges in HTG representations are shown to originate from basic blocks and terminate at basic blocks (i.e., these represent the edges from the control flow graph).

5. PRESYNTHESIS OPTIMIZATIONS

We implemented a number of basic compiler transformations such as common subexpression elimination (CSE) and loop-invariant code motion, copy and constant propagation, and dead code elimination in the *Spark* framework. We use these basic compiler transformations during the presynthesis stage to cleanup the code, that is, remove redundant and unnecessary operations. Although these transformations have been used extensively in compilers [Aho et al. 1986; Muchnick 1997; Bacon et al. 1994], high-level synthesis tools have to take into account the additional control and area (in terms of interconnect) costs of these transformations. Also, transformations such as CSE were originally proposed as operation level transformations. However, recent work has shown that these optimizations are more effective when applied at the source level with a global view of the code structure [Gupta et al. 2000].

We found that of the various presynthesis transformations, common subexpression elimination (CSE) and loop-invariant code motion (LICM) are particularly effective in improving the quality of high-level synthesis results for multimedia applications. These applications are characterized by array accesses within nested loops. The index variable calculation for these array accesses often contains complex arithmetic expressions that can be optimized significantly by CSE and LICM. These claims are validated by our results in Section 9. Before proceeding, we give a brief overview of CSE and LICM.

Common subexpression elimination (CSE) is a well-known transformation that attempts to detect repeating subexpressions in a piece of code, stores them in a variable and reuses the variable wherever the subexpression occurs subsequently [Aho et al. 1986]. *Dominator trees* are commonly used to determine if CSE can be applied to operations [Aho et al. 1986; Sreedhar et al. 1997].

Definition 5.1. A basic block bb_1 in a control flow graph (CFG) is said to *dominate* another node bb_2, if every path from the initial node of the control flow graph to bb_2 goes through bb_1.

In order to preserve the control-flow semantics of a CFG, the common subexpression 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.

Loop-invariant operations are computations within a loop body that produce the same results each time the loop is executed. These computations can be moved outside the loop body, without changing the results of the code. In this way, these computations will execute only once *before* the loop, instead of for *each* iteration of the loop body. An operation *op* is said to be loop-invariant if: (a) its operands are constant, or (b) all operations that write to the operands of operation *op* are outside the loop, or (c) all the operations that write to the operands of the operation *op* are themselves loop invariant [Aho et al. 1986; Muchnick 1997].

6. TRANSFORMATIONS EMPLOYED DURING SCHEDULING

One of the aims of the transformations applied in the presynthesis phase is to increase the applicability and scope of the parallelizing transformations employed by scheduling. In this section, we discuss some of the compiler transformations employed during scheduling. We start off with a review of a set of speculative code motions that we presented earlier and then demonstrate how these code motions can enable new opportunities for applying compiler transformations such as CSE and copy propagation—dynamically—during scheduling. Besides these transformations, the scheduler in *Spark* also employs synthesis techniques such as chaining operations across conditional boundaries [Gupta et al. 2002a] and multicycle operation scheduling.

6.1 Speculative Code Motions

To alleviate the problem of poor synthesis results in the presence of complex control flow in designs, we developed a set of code motion transformations that re-order operations to minimize the effects of the choice of control flow (conditionals and loops) 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 [Gupta et al. 2001a, 2001b]. 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*

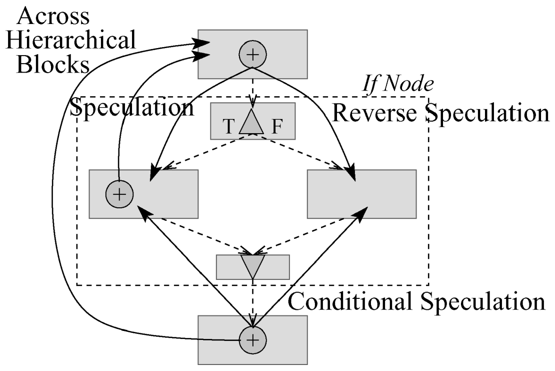

Fig. 4. Various speculative code motions: operations may be speculated, reverse speculated, conditionally speculated or moved across entire conditional blocks.

speculation, wherein an operation from after the conditional block is duplicated *up* into *preceding* conditional branches and executed conditionally. Reverse speculation can be coupled with another novel transformation, namely, *early condition execution*. This transformation evaluates conditional checks as soon as possible. This removes the control dependency on the operations in the conditional branches and thereby, makes them available for scheduling.

The movement of operations due to the various speculative code motions is shown in Figure 4 using solid arrows. Also, shown is the movement of operations across entire hierarchical blocks, such as if-then-else blocks or loops.

6.2 Dynamic Common Subexpression Elimination

We illustrate how the speculative code motions can create new opportunities for applying CSE with the example in Figure 5(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 5(b). Now, the basic block BB_1 containing this speculated operation 5, dominates operation 4's basic block BB_6. Hence, the computation in operation 4 in Figure 5(b) can be replaced by the result of operation 5, as shown in Figure 5(c).

Since CSE is traditionally applied as a pass, usually before scheduling, it can miss these new opportunities created *during* scheduling. This motivated us to develop a technique by which CSE can be applied in the manner shown in the example above, that is, dynamically while the design is being scheduled.

Dynamic CSE is a technique that operates after an operation has been moved and scheduled on a new basic block [Gupta et al. 2002b]. 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 be eliminated if the new basic block containing the newly scheduled operation dominates the basic block of the operation with the

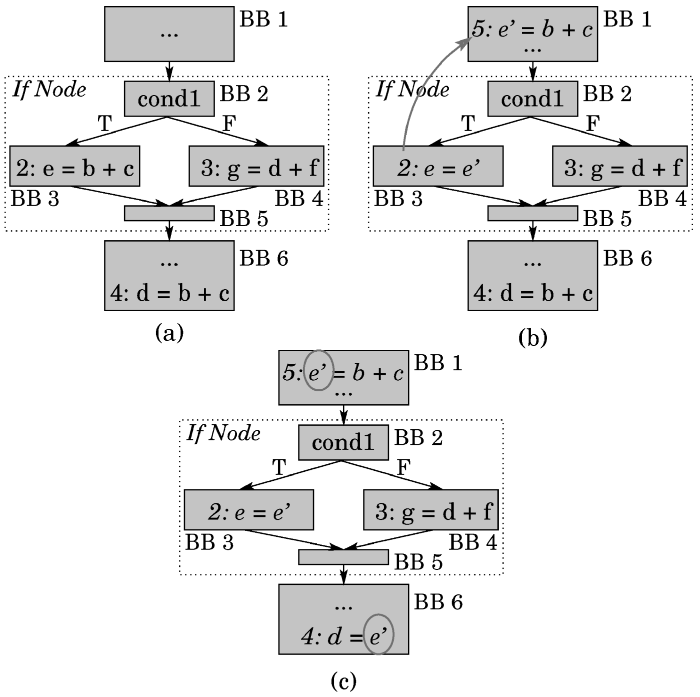

Fig. 5. Dynamic CSE: (a) HTG representation of an example, (b) Speculative execution of operation 2 as operation 5 in BB_1, (c) This allows dynamic CSE to replace the common subexpression in operation 4.

common subexpression. We use the term "dynamic" to differentiate from the phase ordered application of CSE before scheduling.

We can also see from the example in Figure 5 that applying CSE as a pass *after* scheduling is ineffective compared to dynamic CSE. This is because the resource freed up by eliminating operation 4, can potentially be used to schedule another operation in basic block BB_6, by the scheduler. On the other hand, performing CSE after scheduling is too late to effect any decisions by the scheduler.

6.2.1 *Conditional Speculation and Dynamic CSE.* Besides speculation, another code motion that has a significant impact on the number of opportunities available for CSE is conditional speculation [Gupta et al. 2001a]. *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). In this example, the common subexpression that operation 2 in BB_9 has with operation 1 in BB_6 cannot be eliminated since BB_9 is not dominated by BB_6. Consider now 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), operation 1 is duplicated up as operations 3 and 4 in basic blocks BB_2 and BB_3 respectively. These two operations now have a common sub-expression with operation 2 that exists in all control paths leading

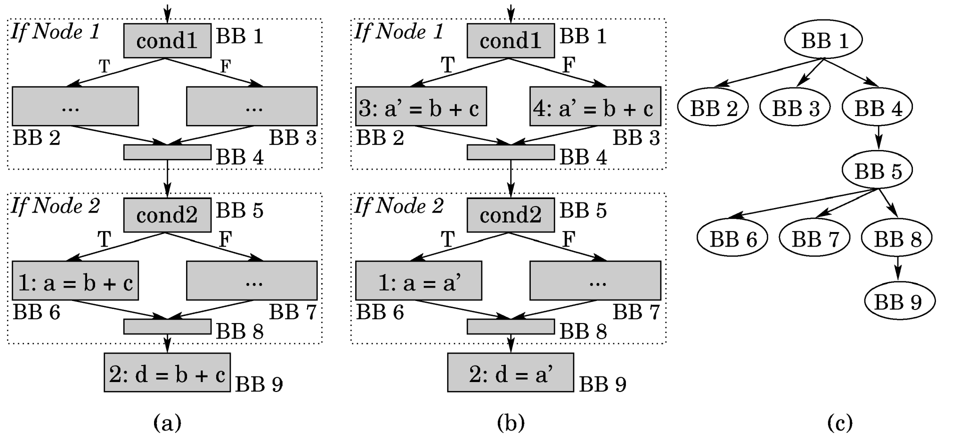

Fig. 6. (a) A sample HTG (b) Operation 1 is conditionally speculated into BB_2 and BB_3. This allows dynamic CSE to be performed for operation 2 in BB_9. (c) Dominator tree for this example.

up to BB_9. Hence, dynamic CSE can now replace the computation in operation 2 with 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 [Sreedhar et al. 1996]. 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. This enables dynamic CSE of operation 2. In this manner, we use this property of group domination while performing dynamic CSE along with code motions such as reverse and conditional speculation that duplicate operations into multiple basic blocks.

6.2.2 *Dynamic Copy Propagation.* The concept of dynamic CSE can also be applied to *copy propagation*. After applying 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 6(b), operations 1 and 2, copy variable a' to variables a 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 during scheduling itself, since this enables opportunities to apply CSE on subsequent operations that read these variable copies. A dead code elimination pass after scheduling can then remove unused copies.

7. PRIORITY-BASED GLOBAL LIST SCHEDULING HEURISTIC

We now present the scheduling heuristic that guides the application of the various scheduling transformations in the *Spark* toolbox. For the purpose of evaluating the various code motion transformations, we chose a *Priority-based*

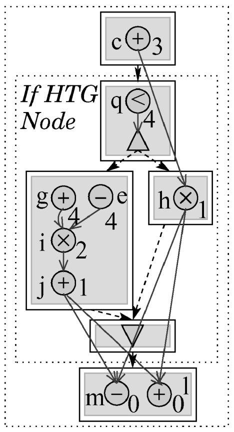

Fig. 7. Priority assignment for the operations of an example.

list scheduling heuristic, although the transformations presented here are applicable to other scheduling heuristics as well. Priority list scheduling works by ordering the unscheduled operations based on a cost determined by the operation's priority and picking the operation with the lowest cost for scheduling.

Our objective is to minimize the *longest delay* through the design; hence, priorities are assigned to each operation based on their distance, in terms of the data dependency chain, from the primary outputs of the design. The priority of an operation is calculated as the delay of the resource that the operation can be mapped to summed with the maximum of the priorities of all the operations that use its result. The algorithm starts by assigning operations that produce outputs a priority of zero, and subsequently, an operation whose result is read by an output has a priority equal to the delay of the resource on which the output operation can be mapped.

The priority assignment of operations of an example design are indicated next to the operations in Figure 7. In this design, the priority assignment of the output operations, m and l is 0, and the operations that depend on them have priority one (since m and l are single-cycle additions). Since operations i and h are two-cycle multiplications, the priority assignment of operations that they depend on is two more than that of i and h. The priority of an operation that creates a conditional check (operation q in the figure) is the maximum of the priorities of all the operations in the conditional branches of the If-HTG.

The scheduling heuristic assigns a cost for each operation based on its priority and favors operations that are on the *longest path* through the design. In this way, the objective is to minimize the longest delay through the design. A different objective such as minimizing average delay can be achieved by incorporating control flow information into the cost function. Also, profiling information, if available, can be used to give operations on paths more likely to be taken a higher priority than operations on less likely paths.

Algorithm 1: PriorityListScheduling ($\mathcal{DG}(G_{HTG}, G_{CFG}, G_{DFG}), \mathcal{R}, CMs$) /* Schedules the Design Graph $\mathcal{DG}$ */

```
1: $Pr \leftarrow$ CalculatePriority ($G_{DFG}$)
2: $step \leftarrow$ First step in first basic block in $G_{CFG}$
3: while $step \neq \phi$ do
4:    for all resources $res \in \mathcal{R}$ do
5:       $\mathcal{A} \leftarrow$ GetAvailableOperations ($\mathcal{DG}$, $step$, $res$, $CMs$)
6:       if ($\mathcal{A} \neq \emptyset$) then
7:          Pick Operation $op \in \mathcal{A}$ with lowest cost
8:          TrailblazeOp ($op$, $res$, $step$, $\mathcal{DG}$, $CMs$)
9:          Mark $op$ as scheduled
10:         ApplyDynamicCSE ($\mathcal{A}$, $op$, $\mathcal{DG}$)
11:      end if
12:   end for
13:   $step \leftarrow$ GetNextSchedulingStep($G_{HTG}$, $G_{CFG}$, $step$)
14: end while
```

The scheduling heuristic is presented in Algorithm 1. The inputs to this heuristic are the unscheduled design graph $\mathcal{DG}$ (that consists of G_{HTG}, G_{CFG}, and G_{DFG}), and the list of resource constraints, $\mathcal{R}$. Additionally, the designer may specify a list of *allowed* code motions (*CMs*), that is, whether speculation, reverse speculation, early condition execution, and conditional speculation are enabled [Gupta et al. 2003a]. The heuristic starts by calling the function *CalculatePriority* to assign a priority to each operation in the input description as explained above. Then scheduling is done by traversing the basic blocks in the design starting with the first basic block in the design (initial basic block in G_{CFG}). The design traversal algorithms are presented later in Section 8.2. The scheduler schedules each scheduling step in a basic block by first collecting a list of *available* operations for each resource in the resource list (lines 4 and 5 in Algorithm 1).

Available operations is a list of operations that can be scheduled on the given resource at the current scheduling step. The algorithm that collects the list of available operations is listed in Algorithm 2. Initially, all unscheduled operations in the design graph $\mathcal{DG}$ that can be scheduled on the current resource type are added to the available operations list. These unscheduled operations are collected by traversing the basic blocks on the control flow paths from the current basic block being scheduled (the unscheduled operations from the current basic block are also added). Operations in the available list whose data dependencies are not satisfied and cannot be satisfied by variable renaming [Gupta et al. 2003a] are removed from the list. Operations that cannot be moved in the design to the current scheduling step using the user-defined *allowed* code motions (*CMs*) are also removed from the available operations list. This is shown in lines 3 and 4 of Algorithm 2.

Next, the available operations algorithm determines the list of basic blocks, *BBList*, that the operation op_i will have to be duplicated into, if it is scheduled on $step_k$. This list, *BBList*, will be nonempty if the operation op_i is being conditionally speculated. For each basic block *bb* in *BBList*, the function

Algorithm 2: GetAvailableOperations ($\mathcal{DG}$, *step*, *res*, *CMs*)
Returns: Available Operations List $\mathcal{A}$
/* Gets operations available to be scheduled on *res* in *step* */

```
    /* Find all unscheduled operations in $G_{CFG}$ that can be scheduled on res */
 1: $\mathcal{A} \leftarrow$ Unscheduled operations in $\mathcal{DG}$ that can be scheduled on res
 2: for all $op_i \in \mathcal{A}$ do
 3:     if ((Data dependencies of $op_i$ cannot be satisfied) or
           (IsTrailblazePossible($op_i$, step, CMs) = false)) then
 4:         $\mathcal{A} \leftarrow \mathcal{A} - op_i$      /* Remove ops that cannot be moved to step using CMs
            */
 5:     else   /* Check if $op_i$ will be duplicated */
 6:         BBList $\leftarrow$ BBs that $op_i$ will be duplicated into
 7:         for all $bb \in$ BBList do
 8:             if (FindIdleRes($op_i$, bb) = $\phi$) then
 9:                 $\mathcal{A} \leftarrow \mathcal{A}$ - $op_i$
10:             end if
11:         end for
12:     end if
13: end for
14: Calculate cost of operations in $\mathcal{A}$
15: return $\mathcal{A}$
```

FindIdleRes is called to determine if there is an idle resource[1] in *bb* that the op_i can be scheduled on. If the *FindIdleRes* function is unable to accommodate a duplicated copy of op_i in even one of the basic blocks in *BBList*, then the operation op_i is removed from the available operation list. This is shown in lines 6 to 11 in Algorithm 2.

Finally, the cost for each of the available operations is calculated. Currently, this cost is the negative of the operations' priority. Future work entails enhancing this cost function to include control and area costs for the code transformations. The scheduling heuristic then picks the operation, *op*, with the *lowest* cost from the available operations list and calls the function, *TrailblazeOp* (not presented here) to schedule this operation on $step_k$ (lines 7 and 8 of Algorithm 1). The *TrailblazeOp* function employs the *Trailblazing* code motion technique [Gupta et al. 2003a; Nicolau and Novack 1993] to efficiently move *op* to the current scheduling step ($step_k$) in the design graph $\mathcal{DG}$.

8. DYNAMIC CSE ALGORITHM

Once the chosen operation has been moved and scheduled, the *dynamic CSE algorithm* comes into play (line 10 in Algorithm 1). This algorithm, listed in Algorithm 3, calls the function *GetOperationsWithCS* to determine the list of operations *csOpsList* that have a common subexpression (CS) with the scheduled operation *op*. The *GetOperationsWithCS* function (not shown here) examines only the remaining operations in the available list to get *csOpsList* since these are the only operations whose data dependencies are satisfied.

[1]A resource is idle in a scheduling step if there is no operation scheduled on the resource in that scheduling step.

Algorithm 3: ApplyDynamicCSE ($\mathcal{A}$, *op*, $\mathcal{DG}$)
/* Eliminates operations from $\mathcal{A}$ by employing dynamic CSE */

```
1: $csOpsList \leftarrow$ GetOperationsWithCS($\mathcal{A}$, $op$)
2: for all operations $csOp \in csOpsList$ do
3:    if ($BB_{Ops}(op)$ dominates $BB_{Ops})(csOp)$ then
4:       ApplyCSE($csOp$, $op$, $G_{DFG}$)
5:    end if
6: end for
```

For each operation *csOp* in *csOpsList*, if the basic block of *csOp* is dominated by the basic block of *op after* scheduling, then the common subexpression in *csOp* is replaced with the result from *op* by calling the *ApplyCSE* function (lines 2 to 4 in Algorithm 3). The *ApplyCSE* function (not described here) also updates the data dependencies in G_{DFG} due to the elimination of the computation in *csOp*. The functions *GetOperationsWithCS* and *ApplyCSE* are part of the basic CSE algorithm.

We illustrate the dynamic CSE algorithm using the earlier example from Figure 5(a). Consider that while scheduling basic block BB_1, the scheduling heuristic determines that the available operations are operations 2, 3 and 4. Of these, 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. It then detects and replaces the common subexpression $(b+c)$ in operation 4 with the result, e', of the scheduled operation 5, since $BB(op_5)$ dominates $BB(op_4)$.

After applying dynamic CSE, the scheduler repeats the procedure discussed above for each resource from $\mathcal{R}$ in the current scheduling step. It then calls the function, *GetNextSchedulingStep*, (discussed in Section 8.2.1) to get the next scheduling step to schedule and repeats the entire scheduling procedure until all the operations in the design are scheduled.

8.1 Scheduling Loops

Loops are scheduled in the same manner as described above. However, user-specified loop transformations such as loop unrolling are applied first. The scheduler cannot move operations into or out of the loop body. This can only be done by transformations such as loop-invariant code motion or loop pipelining. Hence, the unscheduled operations from within the loop body are not collected by the available operations algorithm while scheduling outside the loop body. Conversely, when the scheduler is scheduling the loop body of a loop node, available operations are collected only from within the loop body.

Also, no special consideration is required to schedule loops with unknown loop iteration bounds. Loop bounds are also not required for generating correct, synthesizable VHDL. This is because, in the finite state machine (FSM) generated by *Spark*, at the end of a loop body iteration, the FSM either goes back to the first state in the loop body or goes to the next state after the loop body, depending on whether the loop condition evaluates to true or false. However,

Algorithm 4: GetNextSchedulingStep($step_k$)
Returns: Next Step to schedule *nextStep* after $step_k$

```
1: currBB ← Basic block of $step_k$
2: nextStep ← NextStep(currBB, $step_k$)
3: if (nextStep = $\phi$) then      */ Last step in currBB reached */
      /* Traverse to next basic block */
4:    nextBB ← GetNextBasicBlock(currBB)
5:    if (nextBB ≠ $\phi$) then
6:       nextStep ← FirstStep(nextBB)      /* First step in nextBB */
7:    end if
8: end if
9: return nextStep
```

when the loop bounds are not known, several loop transformations cannot be applied to the design and the cycles that the loop will take to execute cannot be established.

8.2 Design Traversal Algorithms

The design traversal algorithms perform two tasks: get the next basic block to schedule and get the next scheduling step within this basic block. We present the algorithms for these two tasks in the following sections.

8.2.1 *Algorithm to Get Next Scheduling Step.* The scheduling heuristic in *Spark* schedules the design by traversing the design in a top-down manner starting from the first (or initial) basic block in G_{CFG}. For getting the steps to schedule in the design, the scheduler calls the *GetNextSchedulingStep* function listed in Algorithm 4. This function takes as input the current scheduling step $step_k$. First, the basic block *currBB* that $step_k$ belongs to is determined and then, the next scheduling step (*nextStep*) after $step_k$ in *currBB* is determined (lines 1 and 2 in the algorithm).

If *nextStep* is empty—this happens when $step_k$ is the last scheduling step in *currBB*—the algorithm proceeds to get the next basic block in the design by calling the *GetNextBasicBlock* function (discussed in the next section). If this function returns a new basic block, then the first scheduling step in the new basic block is set as the *nextStep* (lines 4 to 6 in Algorithm 4); otherwise there are no more steps left to schedule in the design.

8.2.2 *Algorithm to Get the Next Basic Block to Schedule.* As shown in the main scheduling heuristic in Algorithm 1, the design is scheduled starting at the initial basic block in G_{CFG}. Thereafter, the *GetNextBasicBlock* algorithm returns the next basic block (*nextBB*) to schedule by traversing the basic blocks in the design CFG in a topological manner. This algorithm is listed in Algorithm 5.

This algorithm takes the current basic block *currBB* as input and maintains a queue of basic blocks, *bbQueue*, that it uses to determine the next basic block to schedule. The algorithm inspects each successor basic block bb_i of *currBB* by employing the function *SUCCs(currBB)*. It removes *currBB* from the

Algorithm 5: GetNextBasicBlock(*currBB*, G_{CFG})
Returns: Next Basic Block to schedule *nextBB*
Static: Basic Block Queue *bbQueue*
/* Traverses basic blocks in G_{CFG} in topological order */

```
1: for all $bb_i \in$ SUCCs($currBB$) do
2:    PREDs($bb_i$) $\leftarrow$ PREDs($bb_i$) $-$ $currBB$
3:    if (PREDs($bb_i$) $= \phi$)) then
4:       EnqueueAtTail($bbQueue$, $bb_i$)
5:    end if
6: end for
7: $nextBB \leftarrow$ DequeueHead($bbQueue$)
8: return $nextBB$
```

predecessor basic block list of bb_i (*PREDs*(bb_i)). If bb_i has no more predecessors, that is, all predecessors of bb_i have been visited and scheduled, bb_i is added to the tail of *bbQueue* by calling the *EnqueueAtTail* function. Note that this algorithm is the same as the topological ordering algorithm. Also, note that in practice the functions *SUCCs* and *PREDs* use information maintained by the control flow graph G_{CFG}.

Finally, the *GetNextBasicBlock* returns the basic block at the head of *bbQueue*. The algorithm terminates by returning an empty basic block when the last basic block in the design has been processed. This indicates to the scheduling heuristic that it has finished scheduling the design. Note that, although not shown in Algorithm 5, the *GetNextBasicBlock* algorithm does not traverse the *backward* control flow edge of a loop, that is, the edge that iterates over the loop body. For loops, the loop head is scheduled first, followed by the loop body and then, the loop tail.

9. EXPERIMENTAL SETUP

We implemented the scheduling heuristics along with the pre-synthesis transformations and synthesis and compiler transformations presented in this article in the *Spark* high-level synthesis framework. *Spark* provides the ability to control and thus, experiment with, the various code transformations using user-defined scripts and command-line options. In this section, we present the results for our experiments and demonstrate the utility of the parallelizing transformations in improving the quality of synthesis results. *Spark* consists of 100,000+ lines of C++ code and is available for download from [SPARK website].

For our experiments, we chose three large and moderately complex real-life applications representative of the multimedia and image processing domains. These are the MPEG-1 video compression application [SPARK website], the MPEG-2 application [MediaBench] and the GIMP image processing tool [Gimp website]. From these applications, we derived a few functions that have a mixture of control and data. These designs consist of the *pred1* and *pred2* functions (with the *calcid* function inlined) from the *prediction* block of the MPEG-1 application, the *dpframe_estimate* function (shortened to *dpframe* for the rest

Table I. Characteristics of the Designs used in Our Experiments, Along with the Resources Allocated for Scheduling Them

Benchmark	Lines of C Code	# Ifs	# Loops	# BBs	# Ops	# Resources
MPEG-1 *pred1*	188	4	2	17	123	$2 +-, 1*, 2 <<, 2 ==, 2[]$
MPEG-1 *pred2*	347	11	6	45	287	$2 +-, 1*, 2 <<, 2 ==, 2[]$
MPEG-2 *dpframe*	238	18	4	61	260	$4 +-, 1*, 2 <<, 2 ==, 2[]$
GIMP *tiler*	93	11	2	35	150	$3 +-, 1/, 1*, 2 <<, 2 ==, 2[]$

of this article) from the *motion estimation* block of the MPEG-2 application and the *tile* function[2] (with the *scale* function inlined) from the "tiler" transform of the GIMP tool. The typical run time of *Spark* to produce the results for these designs was in the range of 5 user seconds on a 1.6-Ghz Linux PC.

Table I lists the characteristics of the various designs used in terms of the number of lines of "C" code and the number of if-then-else conditional blocks, for loops, nonempty basic blocks and operations in the input description. All these designs have doubly nested loops. Also, given in this table are the type and quantity of each resource allocated to schedule these designs for all the experiments presented in the following sections. The resources indicated in this table 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 5 cycles. All other resources are single cycle.

The scheduling results presented in the next few sections are in terms of the number of states in the finite state machine controller and the cycles on the longest path (i.e. execution cycles). For conditional (if-then-else) constructs, the longest path is the branch with the larger number of scheduling steps. For loops, the longest path length of the loop body is multiplied by the number of loop iterations. For all the designs used in our experiments, the loop bounds are known.

We also present logic synthesis results obtained after synthesizing the RTL VHDL generated by *Spark* using the Synopsys *Design Compiler* logic synthesis tool. The TSMC 0.13 micron synthesis library is used for technology mapping and components are allocated from the Synopsys *DesignWare Foundation* library. The logic synthesis results are presented in terms of three metrics: the critical path length (in nanoseconds), the unit area (in terms of synthesis library used) and the maximum delay through the design. The critical path length is the length of the longest combinational path in the netlist as reported by static timing analysis tool and this length dictates the clock period of the design. The maximum delay is the product of the longest path length (in cycles) and the critical path length (in ns) and signifies the maximum input to output latency of the design.

In all the results presented in the next few sections, we start with a *"baseline"* case that has all the speculative code motions enabled along with the compiler passes of copy propagation, constant propagation and dead code elimination.

[2]Note that, this floating point function has been arbitrarily converted to an integer function for the purpose of our experiments. This affects the type of data processed and not the nature of the control flow.

Table II. Scheduling Results after Applying Presynthesis Transformations on the Four Designs

	MPEG-1 *pred*1		*MPEG*-1 *pred*2	
Transformation Applied	Num of States	Cycles on Long Path	Num of States	Cycles on Long Path
Baseline	40	1824	84	4187
+ Loop Inv CM	51 (+27.5%)	1396 (−23.6%)	100 (+19.1%)	3266 (−22%)
+ CSE	36 (−10%)	1504 (−17.5%)	73 (−13.1%)	3482 (−16.8%)
+ LICM + CSE	40 (0 %)	1091 (−40.2%)	74 (−11.9%)	2575 (−38.5%)

	MPEG-2 *dpframe*		*GIMP tiler*	
Transformation Applied	Num of States	Cycles on Long Path	Num of States	Cycles on Long Path
Baseline	53	672	42	3931
+ Loop Inv CM	59 (+11.3%)	654 (−2.7%)	48 (+14.3%)	3163 (−19.5%)
+ CSE	50 (−5.7%)	602 (−10.4%)	31 (−26.2%)	2831 (−28%)
+ LICM + CSE	49 (−7.5%)	571 (−15%)	31 (−26.2%)	2534 (−35.5%)

These passes are applied both before and after scheduling. We have shown in past work that employing these speculative code motions significantly enhances the quality of high-level synthesis results [Gupta et al. 2003b]. *Hence, this baseline case represents a design that has already been optimized to a great extent*. Using this baseline case, we demonstrate how the various transformations discussed in this article can further improve the synthesis results, starting with the presynthesis transformations.

9.1 Results for Presynthesis Optimizations

We begin by studying the impact of the presynthesis optimizations on scheduling and logic synthesis results.

9.1.1 *Scheduling Results for Presynthesis Optimizations.* In Table II, we list the scheduling results obtained after the application of the pre-synthesis transformations to the four designs. The results in the first row are for the baseline case (as explained above); the second row for when only loop-invariant code motion (LICM) is applied, the third row for when only CSE is applied and the fourth row for when both LICM and CSE are applied. The percentage reductions of each row over the *baseline* case are given in parentheses.

The results in the second row of these two tables show that when loop-invariant code motion alone is applied, the number of states in the controller increases from 11% to as much as 27%, while the cycles on the longest path through the design decrease by up to 23%. The decrease in cycles is because when loop-invariant operations are moved out of the loop, the loop body becomes smaller, hence, fewer operations execute per loop execution. This allows better compaction (more parallelization) of, and hence, fewer cycles through, the loop body. However, the operations that have been moved outside the loop body require more states to execute and often this increase in the number of states outside the loop is greater than the decrease in the number of states required to execute operations within the loop. We will explore the trade-off this creates between area increase due to controller size and performance increase due to reduced longest path cycles in the next section.

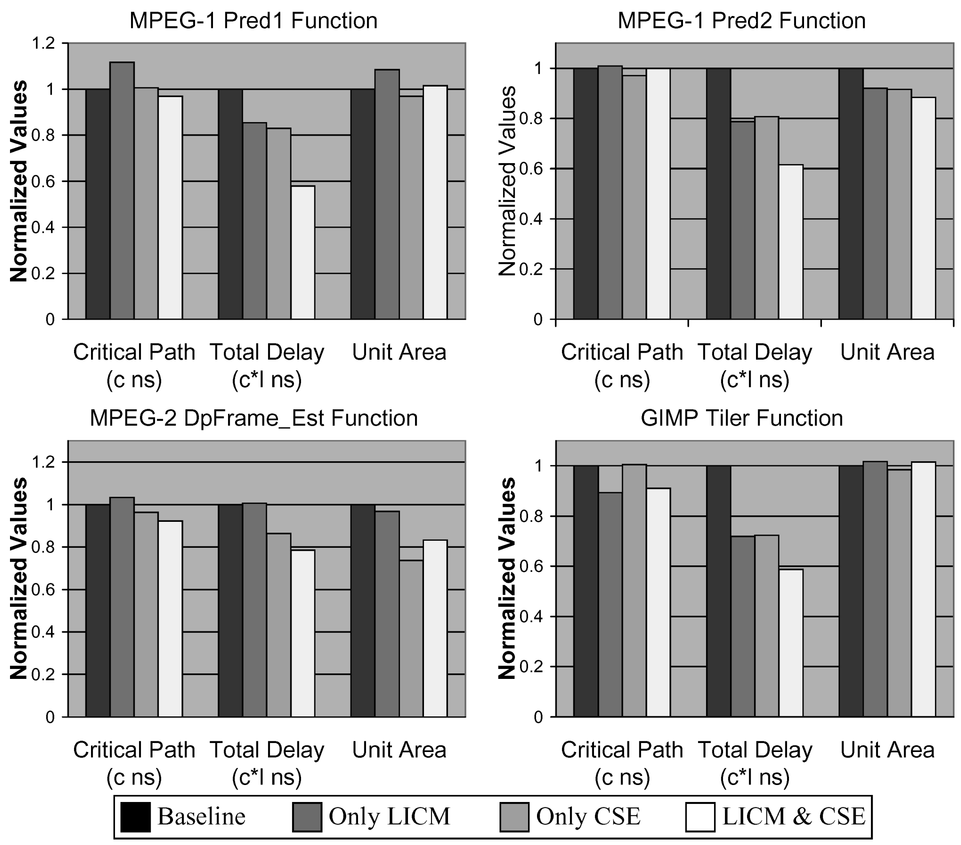

Fig. 8. Effects of the presynthesis transformations on logic synthesis results for the four designs.

The results in the third row of Table II show that when CSE is applied in addition to the transformations in the baseline case, the number of states and the longest path cycles decrease for all four designs; the *tiler* design sees improvements of up to 28%. Clearly, there exist numerous opportunities to apply CSE in off-the-shelf industrial application code. Furthermore, when both LICM and CSE are applied together (last row of the two tables), the reductions in the number of states due to CSE outweigh the corresponding increases due to LICM. Also, the improvements in the cycles on the longest path are additive to some extent, especially for the MPEG-1 designs. These improvements range between 15% to 40% for the four designs.

9.1.2 *Logic Synthesis Results for Presynthesis Optimizations.* We synthesized the VHDL generated by *Spark* corresponding to the presynthesis experiments using the Synopsys Design Compiler. The results for the critical path length, the total delay and the unit area (see Section 9) are presented in the graphs in Figure 8. The bars in these graphs represent the baseline case (1st bar), when only LICM is applied (2nd bar), when CSE is applied (3rd bar) and finally, when both LICM and CSE are applied (4th bar). All the metrics mapped are normalized with respect to the baseline case.

Table III. Scheduling Results after Applying CSE and Dynamic CSE for the MPEG-1, MPEG-2 and GIMP Designs

Transform	*MPEG*-1 *pred*1			*MPEG*-1 *pred*2		
Applied	# States	Long Path	# Regs	# States	Long Path	# Regs
Baseline	40	1824	17	84	4187	31
+ CSE	36 (−10%)	1504 (−18%)	14 (−17.6%)	73 (−13.1%)	3482 (−16.8%)	25 (−19.4%)
+ Dyn CSE	32 (−20%)	1184 (−35%)	10 (−41.2%)	65 (−22.6%)	2906 (−30.6%)	18 (−41.9%)
+CSE+DCSE	32 (−20%)	1184 (−35%)	10 (−41.2%)	65 (−22.6%)	2906 (−30.6%)	17 (−45.2%)

Transform	*MPEG*-2 *dpframe*			*GIMP tiler*		
Applied	# States	Long Path	# Regs	# States	Long Path	# Regs
Baseline	53	672	32	42	3931	14
+ CSE	50 (−5.7%)	602 (−10.4%)	31 (−3.1%)	31 (−26%)	2831 (−28%)	15 (+7.1%)
+ Dyn CSE	50 (−5.7%)	598 (−11%)	28 (−12.5%)	29 (−31%)	2631 (−33.1%)	15 (+7.1%)
+CSE+DCSE	49 (−7.5%)	598 (−11%)	30 (−6.3%)	29 (−31%)	2631 (−33.1%)	14 (0%)

These results show that the critical path length remains fairly constant when these transformations are applied. This is important because it signifies that the clock period in the design does not increase. Also, the total delay through the circuit reduces since the cycles on the longest path decrease. With LICM and CSE applied individually, the total delay through the circuit decreases by up to 20%. With CSE and LICM applied together, the total delay for all the designs decreases by between 20 to 45%.

However, LICM can lead to a higher area (for the *pred1* and *tiler* designs). This increase is less than 10% and is mainly due to the larger FSM controller size. However, when CSE is applied with LICM, the area of the circuit reduces to being almost the same as the baseline case. This is because of two reasons: (a) reductions in the size of the controller due to CSE, and (b) elimination of redundant operations by CSE. Hence, the number of operations mapped to the functional units reduces, thereby, reducing the steering logic (multiplexers and de-multiplexers) and thus, the area.

9.2 Results for Dynamic Transformations

In this section, we compare the effectiveness of the dynamic CSE transformation applied during scheduling with that of a traditional CSE pass applied before scheduling.

9.2.1 *Scheduling Results for Dynamic CSE.* The scheduling results for experiments with dynamic CSE are presented in Table III for the four designs. The rows in these tables list results for the baseline case (1st row), for when only CSE is applied as a pass before scheduling (2nd row), for when only dynamic CSE is applied during scheduling (3rd row), and for when 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). The percentage reductions of each row over the baseline case are also given in parentheses. These tables also give the number of *registers* required to bind the variables in the designs [Gupta et al. 2001a].

The results in the second row in these tables again show that applying CSE alone leads to improvements up to 26% in the number of states (for *tiler*) and between 10 to 28% in the longest path cycles. In itself, these improvements are significant. These improvements become even more significant when we consider that these functions are called multiple times from within loops in the MPEG and GIMP applications.

When dynamic CSE is applied, the improvements are even more dramatic as is evident by the results in the third row of the tables in Table III. Clearly, dynamic CSE eliminates many more operations with common subexpressions than traditional CSE can. Employing dynamic CSE during scheduling can at times improve schedule lengths by 41% (compared to a maximum of 28% obtained by applying CSE). The results in the last row in these tables show that applying both CSE and dynamic CSE together leads to no further improvements.

Also, our experiments show another important result; contrary to popular belief, we find that applying CSE and dynamic CSE leads to a *reduction* in the number of registers required. This decrease can be attributed to three interrelated 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 instead of requiring two registers to store the two variables/operands that are read by the operation, only one register is required to store the result of the operation; and (c) when operations with the same subexpression are eliminated, then they can reuse the result of only one of the operations. This saves on storing the results of several operations.

9.2.2 *Logic Synthesis Results for Dynamic CSE.* The logic synthesis results corresponding to the experiments on CSE and dynamic CSE are presented in the graphs in Figure 9. The values of each metric are mapped as before: for the baseline case (1st bar), for when only CSE is applied (2nd bar), when only dynamic CSE is applied (3rd bar), and the last bar is for when both CSE and dynamic CSE are applied.

The results in these graphs reflect the scheduling results we saw in the previous section. For all cases of applying CSE and dynamic CSE individually or together, the critical path length remains fairly constant. This coupled with the reductions in cycles on the longest path we saw earlier leads to dramatic reductions in the total delay when dynamic CSE is applied: from about 15% (for *dpframe*) to 40% (for the other three designs). Also, dynamic CSE leads to lower area; sometimes up to 30% less (for *pred2*). Again the decrease in area is due to the smaller controller size, fewer number of registers, and elimination of some operations. Thus, fewer operations are mapped to the functional units and this leads to reduced interconnect (multiplexers and demultiplexers).

The overall results in the graphs in Figure 9 demonstrate that enabling dynamic CSE reduces the total delay through the circuit by up to 40% while at the same time reducing the design area; these improvements are better than applying only CSE before scheduling. Also, these results validate our belief that transformations applied dynamically during scheduling can exploit several new opportunities created by scheduling decisions and the movement of operations due to the speculative code motions.

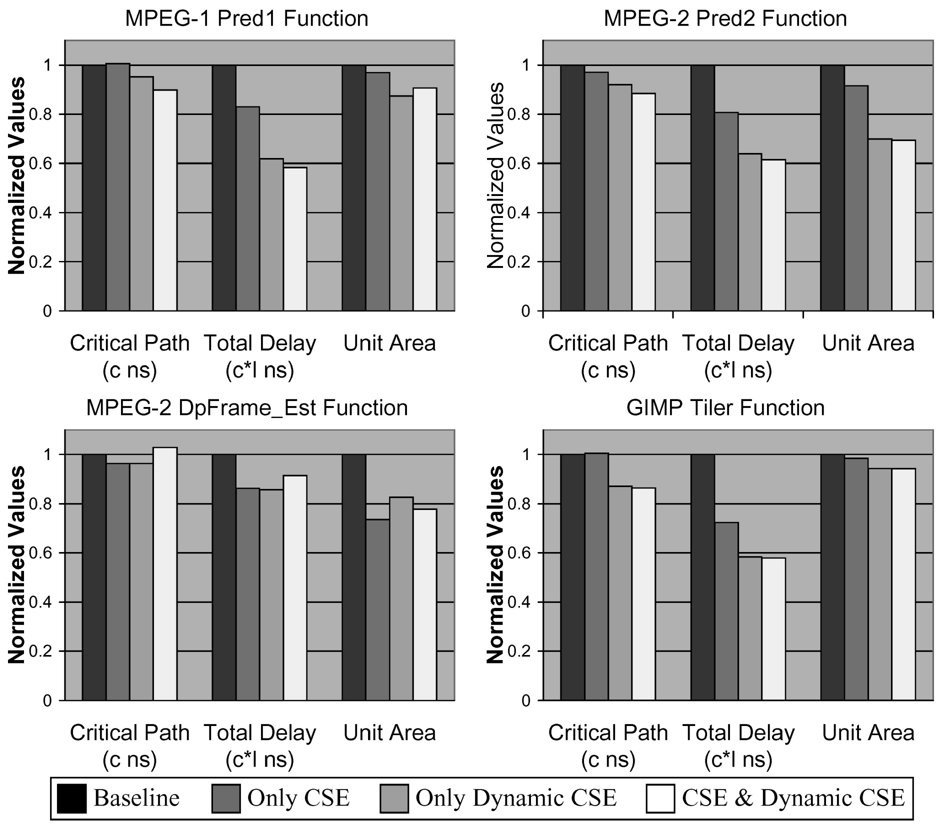

Fig. 9. Effects of CSE and dynamic CSE on logic synthesis results for the four designs.

9.3 Putting It All Together

In this section, we examine how the presynthesis and dynamic transformations perform when applied together. The rows in Table IV list the results for the baseline case with only the speculative code motions applied (1st row), when LICM and dynamic CSE are applied together (2nd row), and when LICM, CSE and dynamic CSE are all applied (3rd row). These results show that when dynamic CSE is enabled along with LICM, the cycles on the longest path decrease by 16% for the MPEG-2 *dpframe* design and by 35 to 50% for the other three designs. The reductions in the number of states in the controller are between 7 to 26%. Applying CSE gives no further improvements over applying only dynamic CSE.

The logic synthesis results corresponding to these experiments are presented in Figure 10. The bars in these graphs represent results for the baseline case (1st bar), LICM and dynamic CSE applied together (2nd bar), and LICM, CSE and dynamic CSE applied together (3rd bar). The improvements in the cycles on the longest path translate over to the longest delay through the circuit; this reduces by 20 to 55%. Also, the area of the design decreases by 5 to 20% when these transformations are applied. Applying CSE along with dynamic CSE and LICM results in better area results for all four designs. It is important to note that these improvements are obtained over designs *already* optimized by the

Table IV. Scheduling Results (Number of States and Cycles on the Longest Path) after Applying Loop-Invariant Code Motion, CSE, and Dynamic CSE on the Four Designs

	MPEG-1 *pred*1		*MPEG*-1 *pred*2	
Transformation Applied	Num of States	Cycles on Long Path	Num of States	Cycles on Long Path
Baseline	40	1824	84	4187
+LICM+DCSE	37 (−7.5%)	899 (−50.7%)	67 (−20.2%)	2127 (−49.2%)
+LICM+CSE+DCSE	37 (−7.5%)	899 (−50.7%)	67 (−20.2%)	2127 (−49.2%)

	MPEG−2 *dpframe*		*GIMP tiler*	
Transformation Applied	Num of States	Cycles on Long Path	Num of States	Cycles on Long Path
Baseline	53	672	42	3931
+LICM+DCSE	47 (−11.3%)	563 (−16.2%)	31 (−26.2%)	2534 (−35.5%)
+LICM+CSE+DCSE	47 (−11.3%)	563 (−16.2%)	31 (−26.2%)	2534 (−35.5%)

speculative code motions [Gupta et al. 2001b]. Note that the area for the *tiler* design remains fairly high due to the area-intensive resources used in this design, namely, a divider and a multiplier.

We found that when an optimizing transformation is applied, there are two conflicting factors that come into play. As the resource utilization increases, the steering logic (multiplexers and demultiplexers) connected to the functional units and the associated control logic increases. On the other hand, as the number of states in the controller decreases, the size and complexity of the controller decreases. We find that critical paths often originate in the controller, go through multiplexers, functional units and de-multiplexers, and finally, terminate in the registers that hold the results. Hence, optimizing transformations often lead to higher area and longer paths through the steering logic, but lower area and shorter paths through the FSM controller. Depending on the effectiveness of the transformation on the particular design being synthesized, one of these factors may overshadow the other. Also, even though the critical path length remains fairly constant, the area may increase. This is because we instruct the logic synthesis tool to sacrifice area to minimize critical path lengths.

10. CONCLUSIONS AND FUTURE WORK

We have developed a methodology for high-level synthesis that first applies coarse-grain and fine-grain source level transformations during a presynthesis phase. This presynthesis phase is followed by a scheduling phase that incorporates a range of parallelizing compiler transformations besides the traditional synthesis transformations. The parallelizing compiler transformations comprise of aggressive speculative code motions aided by transformations applied dynamically during scheduling such as dynamic CSE. These dynamic transformations take advantage of the movement of operations by the speculative code motions. We implemented this synthesis methodology and the various transformations, along with the heuristics that guide them, in the *Spark* synthesis framework. *Spark* takes a behavioral description in ANSI-C as input and produces synthesizable RTL VHDL. This enables us to analyze the impact of the various transformations on the scheduling *and* logic synthesis

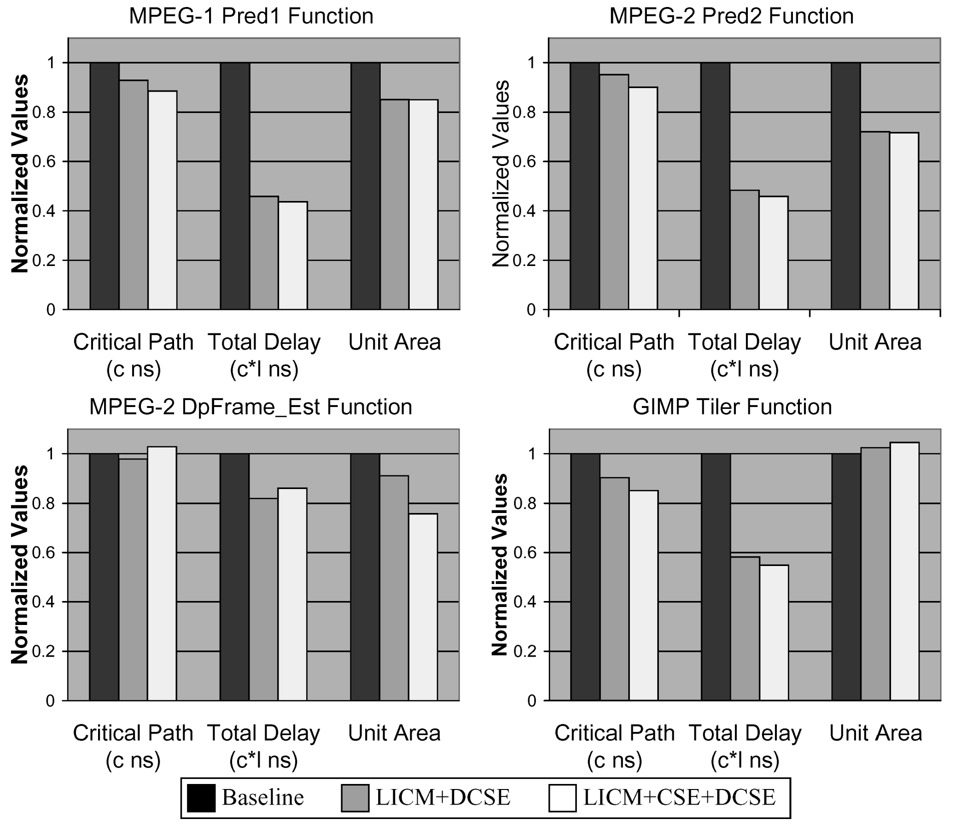

Fig. 10. Logic synthesis results after applying LICM, CSE and dynamic CSE to the four designs.

results. We presented results for experiments on functional blocks derived from applications that are representative of the multimedia and image processing domains, namely, the MPEG-1, MPEG-2 and the GIMP applications. These results demonstrate that when loop-invariant code motion and dynamic CSE are applied together, improvements of up to 60% can be obtained in the delay through the design with reductions of up to 20% in the design area. Furthermore, these improvements are over a design that has already been optimized by the speculative code motions. In future work, we want to study the impact of loop transformations on the quality of synthesis results.

ACKNOWLEDGMENTS

We would like to thank Nick Savoiu and Mehrdad Reshadi for their contribution to the *Spark* framework.

REFERENCES

Aho, A., Sethi, R., and Ullman, J. 1986. *Compilers: Principles and Techniques and Tools*. Addison-Wesley, Reading, Mass.

Bacon, D. F., Graham, S. L., and Sharp, O. J. 1994. Compiler transformations for high-performance computing. *ACM Comput. Surv. 26*, 4, 345–420.

Bergamaschi, R. 1999. Behavioral network graph unifying the domains of high-level and logic synthesis. In *Design Automation Conference.* 213–218.

BRAYTON, R., CAMPOSANO, R., MICHELI, G. D., OTTEN, R., AND VAN EIJNDHOVEN, J. 1988. *The Yorktown Silicon Compiler System*. Addison-Wesley, Chapter in Silicon Compilation.

CAMPOSANO, R. AND WOLF, W. 1991. *High Level VLSI Synthesis*. Kluwer Academic.

Celoxica. Celoxica incorporated. DK Design Suite.

CHAIYAKUL, V., GAJSKI, D., AND RAMACHANDRAN, L. 1992. Minimizing syntactic variance with assignment decision diagrams. Tech. Rep. ICS-TR-92-34, Department of Information and Computer Science, Univ. of California, Irvine.

DE MICHELI, G. 1994. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, New York.

DOS SANTOS, L. 1998. *Exploiting instruction-level parallelism: a constructive approach*. Ph.D. thesis, Electrical Engineering department, Eindhoven University of Technology.

EBCIOGLU, K. AND NICOLAU, A. 1989. A global resource-constrained parallelization technique. In *3rd International Conference on Supercomputing*. 154–163.

FISHER, J. 1981. Trace scheduling: A technique for global microcode compaction. *IEEE Trans. Comput. 30*, 478–490.

FORTE. Forte Design Systems. Behavioral Design Suite.

GAJSKI, D. D., DUTT, N. D., WU, A. C.-H., AND LIN, S. Y.-L. 1992. *High-Level Synthesis: Introduction to Chip and System Design*. Kluwer Academic.

GET2CHIP. Get2Chip Incorporated (acquired by Cadence). G2C Architectural Compiler.

GIMP WEBSITE. GNU Image Manipulation Program. http://www.gimp.org.

GIRKAR, M. AND POLYCHRONOPOULOS, C. 1992. Automatic extraction of functional parallelism from ordinary programs. *IEEE Trans. Parall. Distrib. Syst. 3*, 166–178.

GUPTA, S., DUTT, N., GUPTA, R., AND NICOLAU, A. 2003a. SPARK: A high-level synthesis framework for applying parallelizing compiler transformations. In *Proceedings of the International Conference on VLSI Design*. 461–466.

GUPTA, S., GUPTA, R., DUTT, N., AND NICOLAU, A. 2004. *SPARK: A Parallelizing Approach to the High-Level Synthesis of Digital Circuits*. Kluwer Academic.

GUPTA, S., KAM, T., KISHINEVSKY, M., ROTEM, S., SAVOIU, N., DUTT, N., GUPTA, R., AND NICOLAU, A. 2002a. Coordinated transformations for high-level synthesis of high performance microprocessor blocks. In *Proceedings of the Design Automation Conference*. 898–903.

GUPTA, S., MIRANDA, M., CATTHOOR, F., AND GUPTA, R. 2000. Analysis of high-level address code transformations for programmable processors. In *Design, Automation and Test in Europe*. 9–13.

GUPTA, S., RESHADI, M., SAVOIU, N., DUTT, N., GUPTA, R., AND NICOLAU, A. 2002b. Dynamic common sub-expression elimination during scheduling in high-level synthesis. In *Proceedings of the International Symposium on System Synthesis*. 261–266.

GUPTA, S., SAVOIU, N., DUTT, N., GUPTA, R., AND NICOLAU, A. 2001a. Conditional speculation and its effects on performance and area for high-level synthesis. In *Proceedings of the International Symposium on System Synthesis*. 171–176.

GUPTA, S., SAVOIU, N., DUTT, N., GUPTA, R., AND NICOLAU, A. 2003b. Using global code motions to improve the quality of results for high-level synthesis. *IEEE Trans. CAD 23*, 2 (Feb.).

GUPTA, S., SAVOIU, N., KIM, S., DUTT, N., GUPTA, R., AND NICOLAU, A. 2001b. Speculation techniques for high level synthesis of control intensive designs. In *Proceedings of the Design Automation Conference*. 269–272.

HAYNAL, S. 2000. *Automata-Based Symbolic Scheduling*. Ph.D. dissertation, Electrical and Computer Engineering department, University of California, Santa Barbara.

IQBAL, Z., POTKONJAK, M., DEY, S., AND PARKER, A. 1993. Critical path optimization using retiming and algebraic speed-up. In *Proceedings of the Design Automation Conference*. 573–577.

JANSSEN, M., CATTHOOR, F., AND MAN, H. D. 1994. A specification invariant technique for operation cost minimisation in flow-graphs. In *Proceedings of the International Symposium on High-level Synthesis*. 146–151.

KENNEDY, R., CHAN, S., LIU, S.-M., IO, R., TU, P., AND CHOW, F. 1999. Partial redundancy elimination in SSA form. *ACM Trans. Prog. Lang. Syst.* 21, 3 (May), 627–676.

KOUNTOURIS, A. AND WOLINSKI, C. 1999. High level pre-synthesis optimization steps using hierarchical conditional dependency graphs. In *Proceedings of the Euromicro Confernce*. 1290–1294.

KU, D. AND MICHELI, G. D. 1990. Relative scheduling under timing constraints. In *Proceedings of the Design Automation Conference*. 59–64.

LAKSHMINARAYANA, G., RAGHUNATHAN, A., AND JHA, N. 1999. Wavesched: A novel scheduling technique for control-flow intensive designs. *IEEE Trans. CAD*, 18, 505–523.

LI, J. AND GUPTA, R. 1997. Decomposition of Timed Decision Tables and its use in presynthesis optimizations. In *Proceedings of the International Conference on Computer Aided Design*. 22–27.

LOBO, D. AND PANGRLE, B. 1991. Redundant operator creation: A scheduling optimization technique. In *Proceedings of the Design Automation Conference*. 775–778.

MCFARLAND, M. C. 1978. The Value Trace: A database for automated digital design. Technical Report DRC-01-4-80, Carnegie-Mellon University, Design Research Center.

MEDIABENCH. UCLA Mediabench benchmark suite. `http://cares.icsl.ucla.edu/MediaBench/`.

MIRANDA, M., CATTHOOR, F., JANSSEN, M., AND MAN, H. D. 1998. High-level address optimisation and synthesis techniques for data-transfer intensive applications. *IEEE Trans. VLSI Syst. 6*, 4 (December), 677-686.

MUCHNICK, S. S. 1997. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, Reading, Mass.

NICOLAU, A. 1985. A development environment for scientific parallel programs. Tech. Rep. TR 86-722, Department of Computer Science, Cornell University.

NICOLAU, A. AND NOVACK, S. 1993. Trailblazing: A hierarchical approach to Percolation Scheduling. In *Proceedings of the International Conference on Parallel Processing*. 87–96.

NOVACK, S. AND NICOLAU, A. 1994. Mutation scheduling: A unified approach to compiling for fine-grain parallelism. In *Languages and Compilers for Parallel Computing*. 16–30.

NOVACK, S. AND NICOLAU, A. 1996. An efficient, global resource-directed approach to exploiting instruction-level parallelism. In *Proceedings of the Conference on Parallel Architectures and Compilation Techniques*. 87–96.

ORAILOGLU, A. AND GAJSKI, D. 1986. Flow graph representation. In *Proceedings of the Design Automation Conference*. 503–509.

PARK, S. AND CHOI, K. 2001. Performance-driven high-level synthesis with bit-level chaining and clock selection. *IEEE Trans. CAD*, *20*, 2 (Feb.), 1999-212.

PASKO, R., SCHAUMONT, P., DERUDDER, V., VERNALDE, S., AND DURACKOVA, D. 1999. A new algorithm for elimination of common subexpressions. *IEEE Trans. CAD*, *18*, 1 (Jan), 58–68.

PEYMANDOUST, A. AND MICHELI, G. D. 2001. Symbolic algebra and timing driven data-flow synthesis. In *Proceedings of the International Conference on Computer Aided Design*. 300–305.

POTKONJAK, M. AND RABAEY, J. 1994. Optimizing resource utlization using tranformations. *IEEE Trans. CAD*, *13*, 277-293.

POTKONJAK, M., SRIVASTAVA, M., AND CHANDRAKASAN, A. 1996. Multiple constant multiplications: Efficient and versatile framework and algorithms for exploring common subexpression elimination. *IEEE Trans. CAD*, *15*, 2 (Mar), 141–150.

RADIVOJEVIC, I. AND BREWER, F. 1996. A new symbolic technique for control-dependent scheduling. *IEEE Trans. CAD*, *15*, 45–57.

RIM, M., FANN, Y., AND JAIN, R. 1995. Global scheduling with code-motions for high-level synthesis applications. *IEEE Trans. VLSI Systems*, *3*, 379–392.

SPARK WEBSITE. SPARK parallelizing high-level synthesis framework website. `http://mesl.ucsd.edu/spark`.

SREEDHAR, V., GAO, G. R., AND LEE, Y.-F. 1996. A new framework for exhaustive and incremental data flow analysis using DJ graphs. In *Proceedings of the ACM SIGPLAN Conf. on PLDI*, ACM, New York, 279–290.

SREEDHAR, V., GAO, G. R., AND LEE, Y.-F. 1997. Incremental computation of dominator trees. *ACM Trans. Prog. Lang. Syst. 19*, 2 (Mar.), 239–252.

WAKABAYASHI, K. 1999. C-based synthesis experiences with a behavior synthesizer, "Cyber". In *Proceedings of the Design, Automation and Test in Europe*. 390–393.

WAKABAYASHI, K. AND TANAKA, H. 1992. Global scheduling independent of control dependencies based on condition vectors. In *Proceedings of the Design Automation Conference*. 112–115.

WALKER, R. AND THOMAS, D. 1989. Behavioral transformation for algorithmic level IC design. *IEEE Trans. CAD*, *8*, 1115–1128.

Received March 2003; revised August 2003 and May 2004; accepted May 2004