Coordinated Parallelizing Compiler Optimizations and High-Level Synthesis*

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

CECS
Technical Report #02-35
December 2002

Center for Embedded Computer Systems

‡Dept. of Information and Computer Science
University of California at Irvine
{sumitg, dutt, nicolau}@cecs.uci.edu

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

http://www.cecs.uci.edu/∼spark

Abstract

We present a framework for high-level synthesis that enables the designer to explore the best choice of source level and low level parallelizing transformations for improved synthesis. Within this framework, we have implemented a methodology that applies a set of parallelizing code transformations, both at the source level and during scheduling. Using these transformations, the designer can optimize high-level synthesis results and reduce the impact of control flow constructs on the quality of results. In our methodology, we first apply a set of source level pre-synthesis 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 also show how operation chaining across conditional boundaries can be used to optimize control flow. We have built the Spark *high-level synthesis framework that takes a behavioral description in ANSI-C as input and generates synthesizable register-transfer level VHDL. Our results from 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.*

*A version of this technical report is under revision with the ACM Transactions on Design Automation of Electronic Systems. Also, some parts of this work were presented at DAC 2002 and ISSS 2002. This work is supported by the Semiconductor Research Corporation: Task I.D. 781.001

Contents

List of Figures

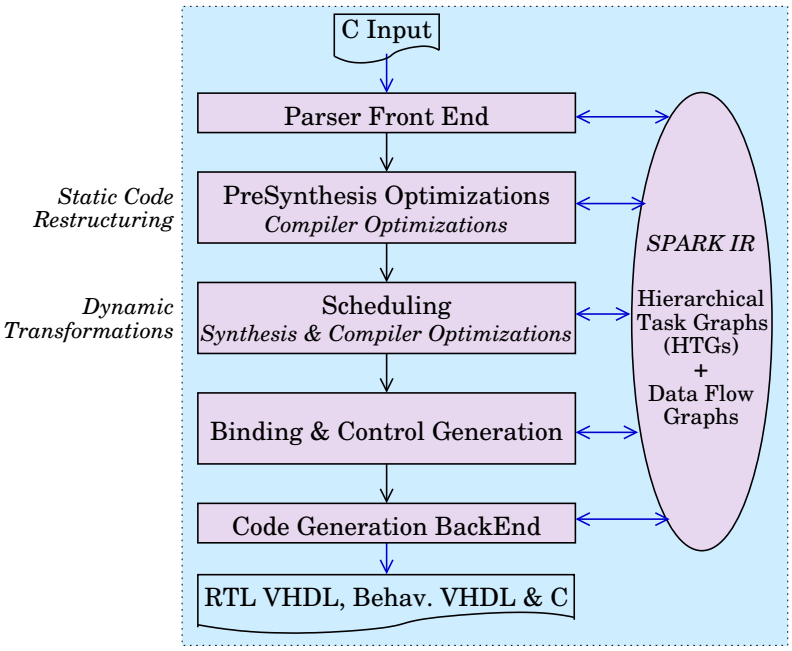

Figure 1. **An overview of the proposed high-level synthesis flow incorporating compiler transformations during the pre-synthesis source-to-source transformation phase and the scheduling phase.**

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 [1, 2, 3, 4, 5]. Recent years have seen the widespread acceptance and use of language level modeling of digital designs. A high level language such as a behavioral HDL (hardware description language) or "C" allows for additional freedom in the way a behavior is described compared to register-transfer level (RTL) descriptions.

However, there are several challenges to this migration to high-level synthesis that limit the utility of high-level synthesis and its wider acceptance. There is a loss of control on the size and quality of the synthesized result. 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 allow us to achieve the best compiler optimizations and synthesis results irrespective of the programming style used in the high level descriptions.

Our approach is outlined in Figure 1. It includes a pre-synthesis phase that makes available a number of transformations to restructure a given description. These include transformations to reduce the number of operations executed such as common sub-expression elimination (CSE), copy propagation, dead code elimination and loop-invariant code motion [6]. Also, we use coarse-level loop transformation techniques such as loop unrolling to restructure the code. This increases 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 syntactic variance or programming style 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. Also, since our methodology targets mixed control-data flow designs, often operations have to be chained across conditional boundaries. 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 generator that can interface with standard logic synthesis tools to generate the gate level netlist.

1.1 What exactly is new here ?

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 with mixed success. In this work, we have experimented with a large number of compiler and parallelizing compiler techniques and identified a select set of these techniques that have the potential for improving high-level synthesis results. However, this requires co-ordination of these techniques across several optimization layers.

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 [7]. While this is a very useful result, in high-level synthesis, code transformations have to be tempered by their effects on the control and area (in terms of interconnect) 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 [8].

The rest of this paper is organized as follows: we first review previous related work. In Section 3, we describe our high-level synthesis methodology, followed by the representation model used. In Section 5, we describe the pre-synthesis transformations. Next, we present the speculative code motion transformations, dynamic CSE and dynamic copy propagation and chaining of operations across conditionals. We then present a priority-based list scheduling heuristic that incorporates these transformations, followed by experimental results.

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 [9, 10, 11, 12]. Early high-level synthesis work

focused on data-flow designs and applied optimizations such as algebraic transformations, re-timing and code motions across multiplexors for improved synthesis results [13, 14]. More recent work – during the last decade – has presented speculative code motions for mixed control-data flow type of designs and demonstrated their effects on schedule lengths [15, 16, 17, 18, 19, 20, 21, 22, 23].

Synthesis transformations such as chaining operations and scheduling on multi-cycle operations are standard in several academic and commercial synthesis tools [11, 12]. However, earlier works discussed these transformations in the context of data-flow designs. In the presence of control, these transformations have to modified to consider resource utilization across multiple control paths. Furthermore, the control and interconnect (multiplexors et cetera) costs of these transformations are higher for control-intensive designs.

Prior work on pre-synthesis transformations has focused on altering the control flow or extracting the maximal set of mutually exclusive operations [24, 25]. Li and Gupta [26] restructure the control flow and attempt to extract common sets of operations within conditionals to improve synthesis results. Kountouris and Wolinski [27] perform operation movement and duplication *before* scheduling and also attempt to detect and eliminate false control paths in the design.

On the other hand, compiler transformations such as CSE and copy propagation predate high-level synthesis and are standard in most software compilers [6, 28]. 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 [29], for optimizing multiple constant multiplications [30, 31] and as an algebraic transformation for operation cost minimization [32, 33].

A converse of CSE, namely, *common sub-expression replication* has been proposed to aid scheduling by adding redundant operations [34, 35]. *Partial redundancy elimination* (PRE) [36] inserts copies of operations present in only one conditional branch into the other conditional branch, so as to eliminate common sub-expressions in subsequent operations. The authors in [32, 37] 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* [38] 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) [39, 40]. 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 [8].

Loop transformations can also be used in high-level synthesis. Potasman et al. [41] demonstrated the improvements that can be obtained by applying the parallelizing compiler technique of *perfect loop pipelining* to data-flow designs. Holtmann et al. [42] apply loop pipelining to the program path that has the highest predicted probability to be taken, thereby deferring operations belonging to other paths.

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 effects of syntactic variance in the input description on the quality of synthesis results [8, 43].

In this paper, we propose a 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 *pre-synthesis* phase in which coarse-grain transformations are applied to the input description before performing high-level synthesis. This new methodology has been implemented in the *Spark* high-level synthesis framework. *Spark* takes a behavioral description in ANSI-C as input and produces synthesizable register-transfer level (RTL) VHDL. An overview of the *Spark* framework is shown in Figure 2. As shown in this figure, *Spark* also takes additional information as input, such as a hardware resource library, resource and timing constraints and user directives for the various heuristics and transformations.

The transformations in the *pre-synthesis 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 sub-expression elimination (CSE), copy propagation, and dead code elimination (c) transformations such as loop-invariant code motion, induction variable analysis (IVA) and operation strength reduction, that reduce the number of operations within loops and replace expensive operations (multiplications and divisions) with simpler operations (shifts, additions and subtractions).

The pre-synthesis phase is followed by the scheduling and allocation phase (see Figure 2). The synthesis transformations applied during scheduling include chaining operations across conditional blocks, scheduling on multicycle operations, resource sharing et cetera [12]. 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) [44]. The scheduler employs code motion techniques such as *Trailblazing* that efficiently move operations in moderately complex control-intensive designs [45].

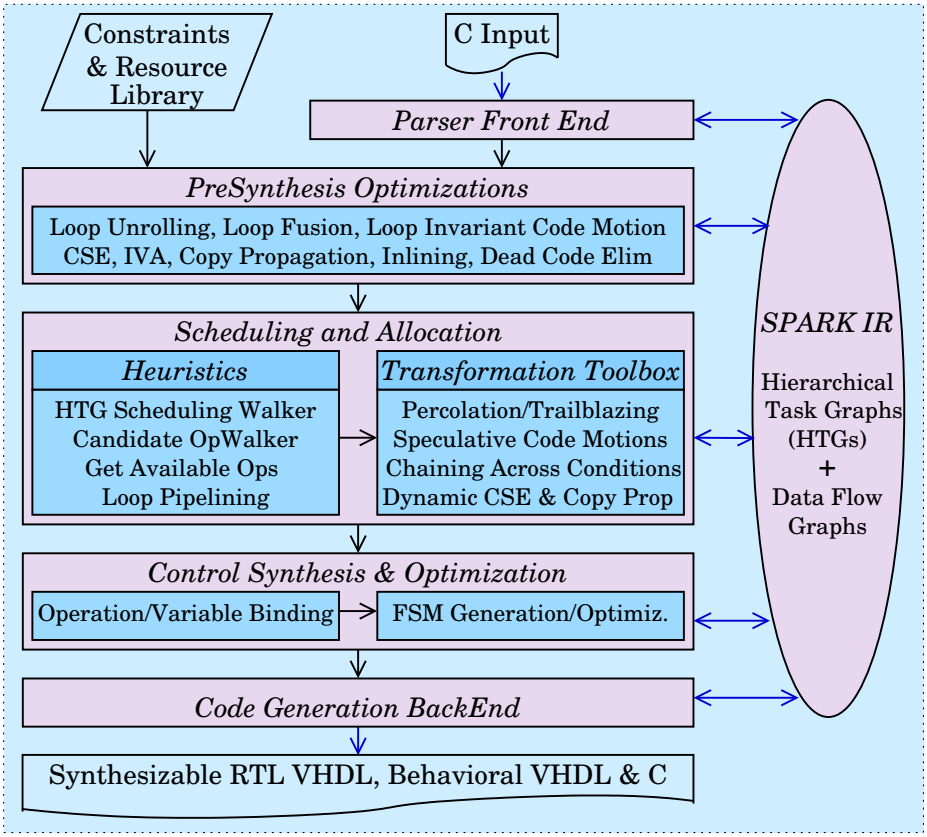

Figure 2. **An overview of the *Spark* High-Level Synthesis Framework.**

As shown in Figure 2, 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 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, sometimes at the expense of a higher number of registers [8]. The control generation pass generates a finite state machine (FSM) that implements the schedule and the controller for it.

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 have also implemented back-end code generation passes that generate ANSI-C and behavioral VHDL. These behavioral output codes represents 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.

In the rest of this paper, we discuss several of the transformations from the pre-synthesis phase and the scheduling phase implemented in the *Spark* framework. However, to enable the various coarse and fine-grain transformations employed by *Spark*, an intermediate representation more powerful than control-data flow graphs is required, as explained in the next section.

4 HTGs: A Intermediate Representation for Control-Intensive Designs

In order to enable the range of optimizations explored by our work, the *Spark* system 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) [45, 46, 47]. In the past, control-data flow graphs (CDFGs) [10, 48] have been primary model for capturing design descriptions for high-level synthesis. CDFGs work very well for traditional scheduling and binding techniques. However, for source-to-source optimizations and other coarse grain transformations, the abstraction level offered by CDFGs is too thin to evaluate the trade-offs. Since HTGs maintain a hierarchy of nodes, they are able to retain coarse, high level information about program structure, in addition to operation level and basic block level information. This aids in operation movement by reducing the amount of compensation code required. Non-incremental moves of operations across large blocks of code are possible without visiting each intermediate node [47].

Of course, several other representation models such as Value Trace (VT) [49], Yorktown Intermediate format (YIF) [50], Assignment Decision Diagrams (ADDs) [51], Hierarchical Conditional Dependency Graphs (HCDGs) [52], et cetera have been proposed earlier for high-level synthesis. Also, Rim et al. [22] and Bergamaschi [53] have proposed new design representation models that attempt to bridge the gap between high-level and logic-level synthesis and aid in estimating the effects of one on the other. However, we have found HTGs to be the most natural choice for our parallelizing transformations. Also, our scheduler relies heavily on compiler transformations such as *Trailblazing* [47] and *Resource-Directed Loop Pipelining* [54] that were originally developed using HTGs as the underlying intermediate representation.

We define HTGs as follows: an HTG is a directed acyclic graph with unique *Start* and *Stop* nodes such that there exists a path from the *Start* node to every node in the HTG and a path from every node in the HTG to the *Stop* node. Edges in a HTG represent control flow. Each node in a HTG can be one of the following three types: single nodes, compound nodes and loop nodes. *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. Operations that execute concurrently are aggregated into *statements* within basic blocks. These statements correspond to control or scheduling steps in high-level synthesis and to VLIW instructions in compilers [46]. The second type of HTG nodes, namely, *compound nodes*, are hierarchical in nature, i.e., they can contain other HTG nodes. They are used to represent structures like if-then-else blocks, switch-case blocks or a series of HTGs. *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. Note that operations represent the expressions in the code and are stored as abstract syntax trees [6]. Each expression or operation is initially encapsulated in a statement of its own.

Figure 3(a) illustrates the HTG for the synthetic benchmark “waka” [16] along with the data flow dependencies.

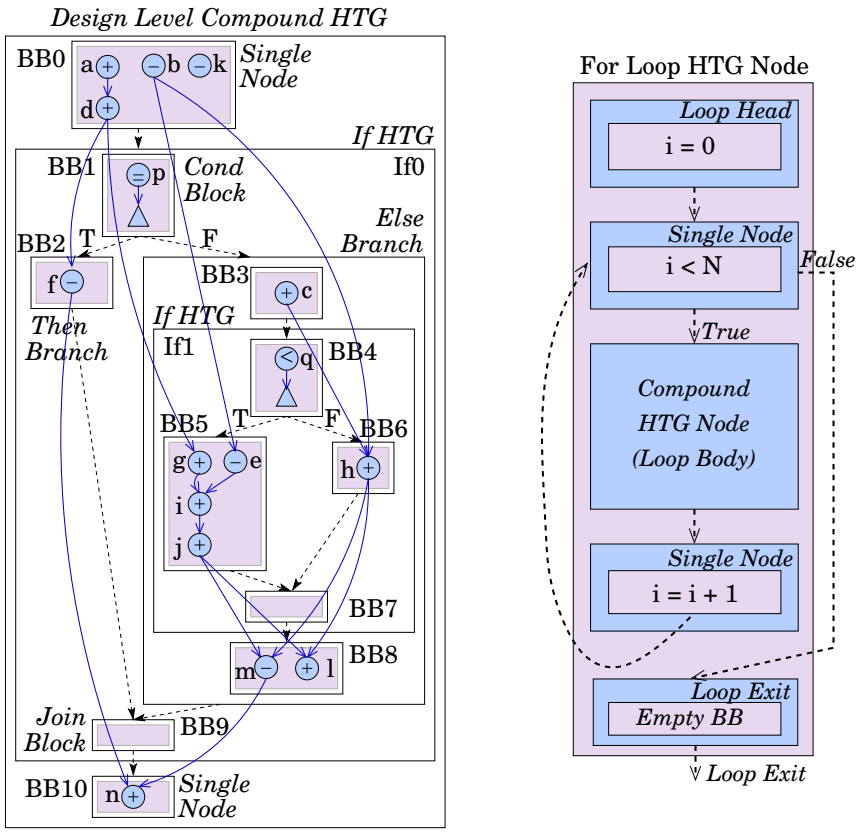

Figure 3. **(a) The hierarchical task graph (HTG) representation of the "waka" benchmark. Flow data dependencies are also shown. (b) The HTG representation of a For-Loop.**

In this figure, the dashed arrows indicate control flow and the solid lines indicate data flow. Operations are denoted by circular nodes with the operator sign within and the triangle indicates a Boolean conditional check. BB_0 to BB_{10} denote basic blocks. 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, a compound HTG for the true branch, a compound HTG for the false branch and a single node with an empty basic block for the *Join* node. The *Start* node for an If-HTG is the single node with the conditional check and the *Stop* node is the Join node. In this example, the true/then branch contains only one basic block (encapsulated in a single node).

The conceptual HTG representation of a *For-loop HTG* is shown in Figure 3(b); only control flow dependencies are shown in this figure. A For-loop HTG consists of a single node with an initialization basic block (*Start* node), a single node for the conditional check basic block and a compound HTG node for the body of the loop with an optional basic block for the loop index increment and a loop exit node. There is a backward control flow arc from the end of the loop body to the conditional check single node. The loop exit is a single node with an empty basic block; this is also the *Stop* node. Note that, the *Start* and *Stop* nodes of a single node are the node itself.

HTGs are constructed from the input description by first creating a compound HTG node for the design level HTG. Each sequential piece of code in the input description forms a sub-node of this HTG. The *Start* and *Stop*

nodes of the design level HTG correspond to the *Start* node of the first sub-node HTG and *Stop* node of the last sub-node in the design respectively. Hence, for the *waka* design shown in Figure 3(a), the design level HTG node has three sub-nodes. The first sub-node is a single node with basic block BB_0, the second sub-node is the If-HTG node, IF_0 and the third sub-node is the single node with basic block BB_{10}. The *Start* and *Stop* nodes for this design are the single nodes that encapsulate basic blocks BB_0 and BB_{10}.

The HTG representation of the *waka* design in Figure 3(a) also shows the data flow dependencies between the operations. However, these data flow dependencies are actually stored in data dependency graphs that augment HTGs. The data dependency graphs in our implementation capture all the different types of data dependencies present in the input description. This is in contrast to traditional synthesis approaches that typically only capture *flow* data dependencies, i.e., the data dependencies between two operations wherein one operation reads the result produced by the other. However, a design specification in a high-level language usually contains other types of data dependencies such as write-after-read and write-after-write dependencies [28]. Maintaining only flow data dependencies means that the information about the variable names from the original description are discarded. Hence, the correlation between the original description and the intermediate representation is lost. This leads to an inability to *visualize* the intermediate results of the various transformations vis-a-vis the original input description. Hence, we maintain the full set of data dependencies as given in the input description and employ techniques such as dynamic variable renaming to aid in reducing the restrictions imposed by these dependencies [45].

Another important feature of HTGs is that they are *strongly connected components* (SCC) [46]. An SCC region (in this case a HTG node) has the property of having a single entry and a single exit point; for a HTG, these correspond to the *Start* and *Stop* nodes respectively. This property is exploited by code motion techniques such as *Trailblazing* [47] to make hierarchical moves when moving operations in a HTG. For example, when the *Stop* node of a HTG node is encountered while moving an operation, trailblazing can check if the operation has any data dependencies with the HTG node. If it does not, the operation can be moved directly to the *Start* node of the HTG node without visiting each node in the HTG. For if-then-else blocks, this also means that the operation can be moved across the conditional block without duplicating into the branches of the conditional. Details of this can be found in [47].

In the next section, we present source level transformations that use the various properties of HTGs, specifically, the coarse-level and hierarchical information maintained by HTGs to restructure the code.

5 Pre-Synthesis Optimizations

Operation level compiler transformations such as common sub-expression elimination and coarse-level transformations such as loop transformations have been used extensively in optimizing compilers, especially in the context of parallelizing compilers [6, 28, 55]. However, high-level synthesis tools have to take into account the additional area and performance costs of code transformations; these may occur due to the additional control and

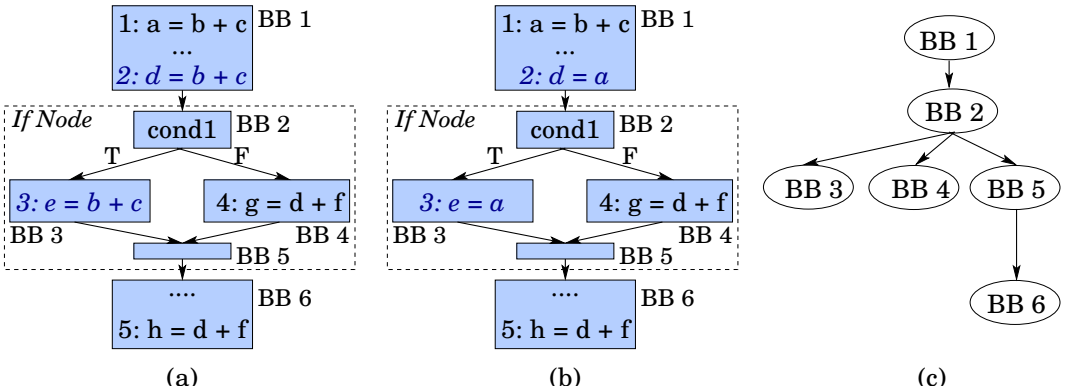

Figure 4. **CSE: (a) a sample HTG (b) the common sub-expression $b+c$ in operations 2 and 3 has been replaced with the variable a from operation 1. (c) Basic block dominator tree for this example. Operation 5 cannot be eliminated by the expression in operation 4, since BB_4 does not dominate BB_6**

steering logic (multiplexors) generated.

We have implemented several compiler transformations in the *Spark* HLS framework both at the pre-synthesis phase and at the scheduling phase. In this section, we will discuss common sub-expression elimination (CSE), loop-invariant code motion and loop unrolling. Although transformations such as CSE were originally proposed as operation level transformations, recent work has shown that these optimizations are more effective when applied at the source level with a global view of the code structure [37].

5.1 Common Sub-Expression Elimination

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 [6]. This is demonstrated by the the example in Figure 4(a). The common sub-expression $b+c$ in operations 2 and 3 can be replaced with the result of operation 1, resulting in the code in Figure 4(b).

Whether a common sub-expression between two operations can be eliminated depends on the control flow between the locations or basic blocks of the two operations. One common approach to capture the relationship between basic blocks in a control flow graph is using *dominator trees* [6]. These trees can be constructed using the following definition: a node d in a control flow graph (CFG) is said to *dominate* another node n, if every path from the initial node of the flow graph to n goes through d. The dominator tree for the example in Figure 4(a) is given in Figure 4(c). In this example, basic block BB_2 dominates basic blocks BB_3, BB_4 and BB_5 and is itself dominated by BB_1. BB_5 in turn dominates BB_6.

In order to preserve the control-flow semantics of a CFG, the common sub-expression in an operation op_2 can only be replaced with the result of another operation op_1, if op_1 resides in a basic block BB_1 that dominates the basic block BB_2 in which op_2 resides. So, in the example in Figure 4(a), operations 2 and 3 can be eliminated using the result of operation 1 as per the dominator tree shown in Figure 4(c). Conversely, BB_4 does *not* dominate BB_6 and hence, the common sub-expression in operation 5 cannot be replaced with the result of operation 4.

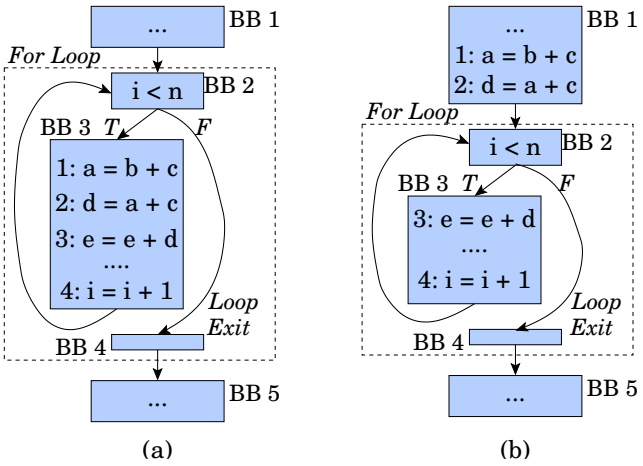

Figure 5. **(a) The HTG of an example with a loop. (b) The loop-invariant operations Op_1 and Op_2 are moved outside the loop body.**

Dominator trees have been extensively used previously for data flow analysis and transformations such as loop-invariant code motion and CSE [6, 56]. They have recently been extended to incorporate the notion of sets of basic blocks dominating over other basic blocks (see Section 6.2.1) [57]. In our work as well, we apply CSE based on the dominator information of the basic blocks that contain the operations.

5.2 Loop-Invariant Code Motion

Frequently, there exist computations within a loop body that produce the same results each time the loop is executed. These computations are known as *loop-invariant code* and 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. Consequently, this leads to better design performance, albeit only if the loop is executed at least once.

An operation op is said to 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 [6, 28].

Figure 5 demonstrates loop-invariant code motion with an example. In the example in Figure 5(a), the operands *b* and *c* of operations Op_1 are not written to by any operation within the loop. Hence, Op_1 is loop-invariant. Similarly, the operand *a* of operation Op_2 is written only by the loop-invariant operation Op_1 and its other operand, *c*, is not written within the loop. Hence, Op_2 is also loop-invariant. Thus, these operations can be moved out of the loop body into basic block BB_1 as shown in Figure 5(b). Operations Op_3 and Op_4 are not loop-invariant since one of their operands is written to, from within the loop.

When describing behaviors in high-level languages, designers frequently place several loop-invariant operations within loops for ease of understanding and readability of the code. Furthermore, loops themselves are used as a programming convenience and often do not expose all the available parallelism in the design. Hence, fine grain

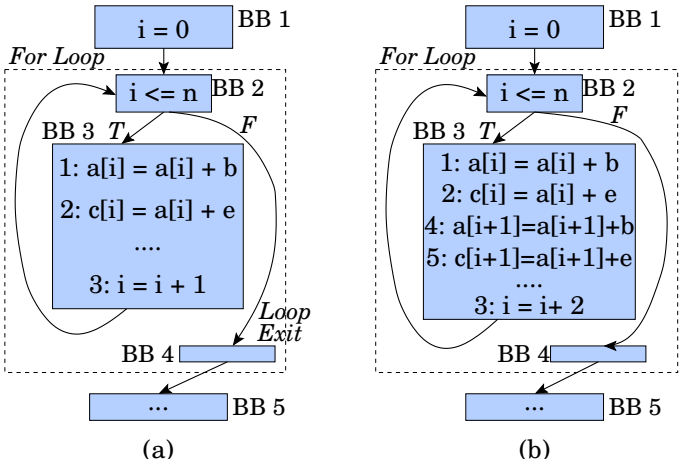

Figure 6. **(a) The HTG of an example with a loop and some operations. (b) The loop is unrolled once**

transformations such as CSE, copy propagation and loop-invariant code motion, need to be complemented with a set of loop transformations that can significantly alter the structure of the code.

A wide range of loop transformations have been explored in the context of parallelizing compilers [55]. However, their use in high-level synthesis has been limited. It is not clear if, and under what criteria, are any of these loop transformations useful. In the next two sections, we discuss one loop transformation, namely, loop unrolling and one of the subsequent optimizations that can be performed in the context of high-level synthesis.

5.3 Loop Unrolling

Loop unrolling is the process of placing a duplicate of one or more iterations of the loop body at the end of the current loop body. The loop bounds and loop index variable increment are updated as necessary. Loop unrolling was developed to enable software compilers to perform optimizations across loop iterations and facilitate global code optimizations [28]. However, loop unrolling can lead to code explosion; so, loops are unrolled one iteration at a time, followed by code compaction by parallelizing transformations, until no further improvements can be obtained. Loops are seldom unrolled fully.

On the other hand, in the context of hardware design descriptions, loops are only a programming convenience and latency constraints generally dictate the amount of unrolling a loop has to undergo. For instance, if a design is targeted to, say, three clock cycles, it implies that *all* the operations within *all* the iterations of the loop have to be executed in these three cycles. Hence, when this design is mapped to hardware, it will generate a design in which the loop is, in essence, unrolled within these three cycles. Some hardware architectures such as microprocessor functional blocks are low latency designs that must often be executed in just one cycle [58]. Loops in single cycle designs must be unrolled completely.

Loop unrolling is demonstrated in Figure 6. Figure 6(a) shows the HTG of a synthetic example, which has a loop and some operations within this loop. Operation Op_1 uses the loop index variable i to read the array a and another operand, b, to generate the result $a[i]$. This result is used by operation Op_2 to generate the result $c[i]$. When

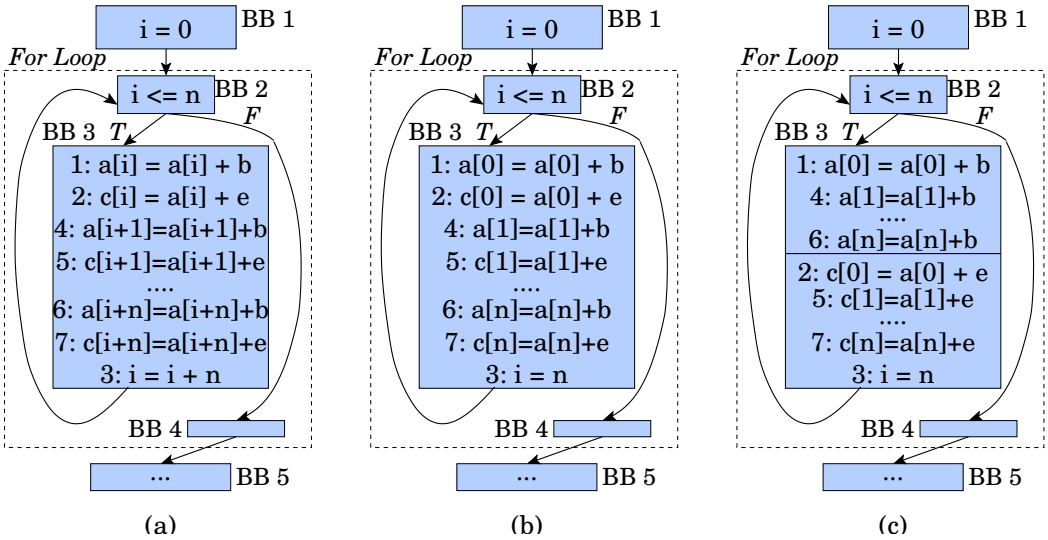

Figure 7. **(a) The loop is unrolled completely from the example in Figure 6(a). (b) Constant propagation of loop index variable, i. (c) The calculation of array $a[]$ values are performed concurrently followed by concurrent calculation of the $c[]$ array values**

this loop is unrolled once, the resulting HTG is as shown in Figure 6(b). This unrolled loop exposes the inherent parallelism among the operations in the loop body – the operations Op_1 and Op_4 can be executed concurrently, followed by the concurrent execution of operations Op_2 and Op_5. Without loop unrolling, the two iterations of the loop body would have been executed sequentially.

In the *Spark* framework, the amount of loop unrolling for each loop is currently user-directed. The designer can experiment with different unrolls of the loop and determine the trade-offs. Also, unlike software, code explosion is not a matter of concern in hardware design. However, loop unrolling leads to a larger controller and more complex interconnect (multiplexors) among operations due to increased resource sharing. As we will show in the results section, given enough resources, the increase in concurrency offsets the overheads incurred by unrolling.

5.4 Loop Index Variable Elimination

When a loop is unrolled completely, it enables the applicability of another transformation; loop index variable elimination. We will demonstrate this transformation with the previous example from Figure 6(a). Consider that the loop in this code is unrolled completely; the resulting design is shown in Figure 7(a). The value of the loop index variable is now known statically in all the loop iterations. Hence, the initial value assigned to the loop index variable, $i = 0$, can be propagated as a constant throughout all the iterations (known as constant propagation). The resultant design is shown in Figure 7(b).

In this way, the loop index variable is completely eliminated from the loop body. This removes the data dependencies that exist between the operations in the loop body and loop index variable, thus allowing the application of further code parallelizing transformations. In this example, the code motion transformations applied during the later scheduling phase can then concurrently calculate all the values of the $a[]$ array followed by the concurrent

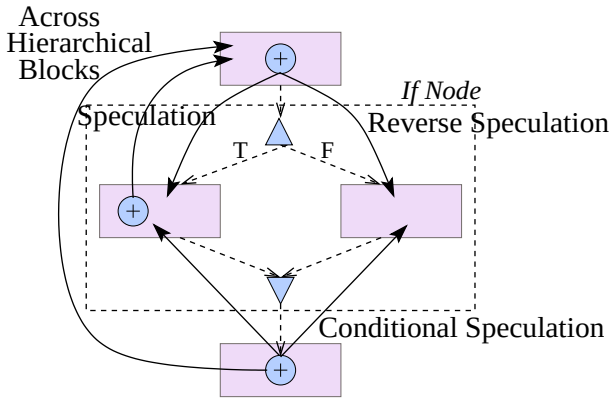

Figure 8. **Various speculative code motions: operations may be speculated, reverse speculated, conditionally speculated or moved across entire conditional blocks.**

calculation of all $c[]$ array values (assuming that the resources to do so are available), as shown in Figure 7(c).

We have shown earlier that this combination of full loop unrolling along with loop index variable elimination is an essential transformation for the synthesis of microprocessor functional blocks [58]. However, we apply only partial loop unrolling for designs from the multimedia and image processing domains, since these designs usually have latencies of 10s to 100s of cycles. Hence, full loop unrolling would lead to an explosion in the controller size and interconnect costs.

6 Transformations Employed During Scheduling

One of the aims of the transformations applied in the pre-synthesis stage is to increase the applicability and scope of the parallelizing transformations employed by scheduling. Scheduling employs several transformations, some being compiler transformations and some being high-level synthesis transformations. In this section, we discuss some of them. We start off with an overview of a set of speculative code motions and demonstrate how these code motions can enable new opportunities for applying compiler transformations such as CSE and copy propagation, dynamically, during scheduling. We then discuss a classical high-level synthesis transformation, namely, operation chaining, albeit modified to handle the control-intensive designs that our approach targets.

6.1 Speculative Code Motions

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

Generally, speculation refers to the unconditional execution of operations that were originally supposed to have executed conditionally. However, frequently there are situations when there is a need to move operations *into* conditionals [43, 8]. This may be done by *reverse speculation*, where operations before conditionals are moved into *subsequent* conditional blocks and executed conditionally, or this may be done by *conditional speculation*,

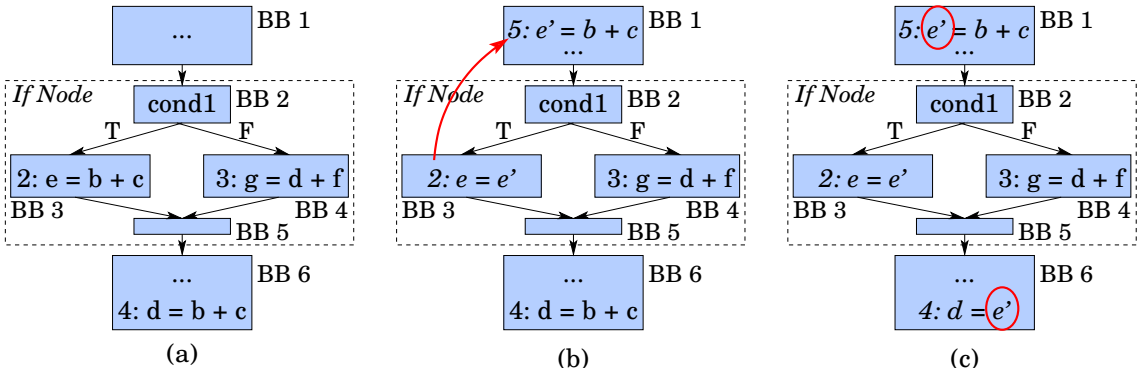

Figure 9. **Dynamic CSE: (a) HTG representation of an example, (b) Speculative execution of operation** 2 **as operation** 5 **in** BB_1**, (c) This allows dynamic CSE to replace the common sub-expression in operation** 4**.**

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 involves restructuring the original code, so as to evaluate conditional checks as soon as possible. The motivation for this transformation comes from the fact that once a condition has been evaluated, all the operations in its branches are ready to be scheduled.

The various speculative code motions are shown in Figure 8 by solid lines. Also, shown is the movement of operations across entire hierarchical blocks, such as if-then-else blocks or loops. These code motions have been shown to be effective in improving both the scheduling and the synthesis results of high-level synthesis, particularly for control-intensive designs.

These beyond-basic-block code motions usually re-order, speculate and sometimes duplicate operations. This often creates new opportunities for dynamically applying transformations such as common sub-expression elimination during scheduling as discussed in the next section.

6.2 Dynamic Common Sub-Expression Elimination

To illustrate how these speculative code motions can create new opportunities for applying CSE, consider the example in Figure 9(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 (see Section 5.1 for a description of dominator trees). 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 9(b). Now, the basic block BB_1 containing this speculated operation 5, dominates operation 4's basic block BB_6. Hence, operation 4 in Figure 9(b) can be eliminated and simply replaced by the result of operation 5, as shown in Figure 9(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, i.e., 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 [44]. It examines the list

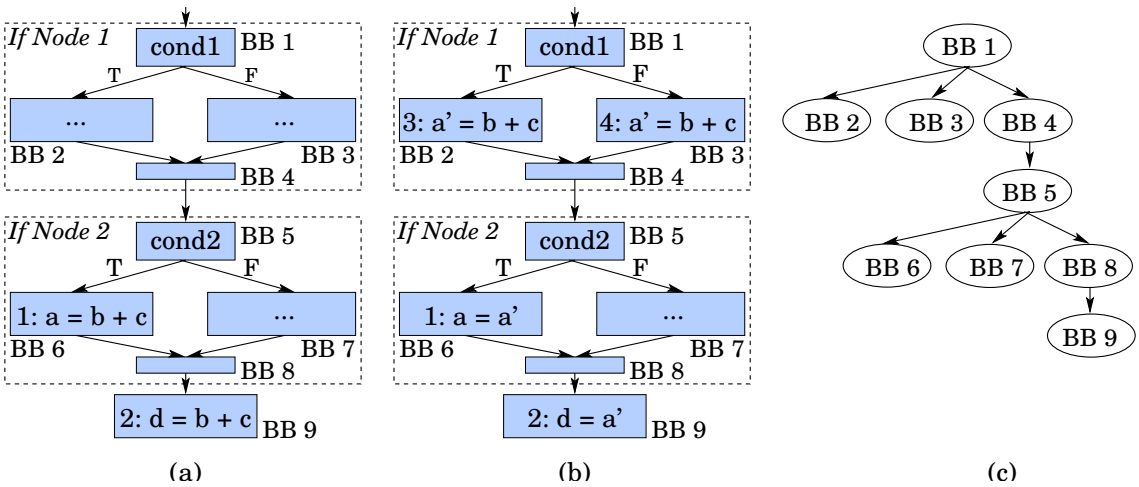

Figure 10. **(a) Dynamic CSE after conditional speculation: (a) A sample HTG (b) Operation** 1 **has been conditionally speculated into** BB_2 **and** BB_3**. This allows dynamic CSE to be performed for operation** 2 **in** BB_9**. (b) Dominator tree for this example**

of remaining ready-to-be-scheduled operations and determines which of these have a common sub-expression with the currently scheduled operation. This common sub-expression can now be eliminated if the new basic block containing the currently scheduled operation dominates the basic block of the operation with the common sub-expression. 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 9 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 [8]. *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 10(a). Consider that the scheduling heuristic decides to conditionally speculate operation 1 into the branches of the if-then-else conditional block, $IfNode_1$. Hence, as shown in Figure 10(b), the operation is duplicated up as operations 3 and 4 in basic blocks BB_2 and BB_3 respectively.

Looking at the original description in Figure 10(a) again, we note that operation 2 in BB_9 has a common sub-expression with operation 1 in BB_6. But since BB_9, is not dominated by BB_6, this common sub-expression cannot be eliminated by classical CSE. However, after conditional speculation, operations with this common sub-expression exist in all control paths leading up to BB_9. Hence, we can apply dynamic CSE now and operation 2 uses the result, a', of operations 3 and 4 as shown in Figure 10(b).

This leads to the notion of dominance by sets of basic blocks [57]. 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 10(b), basic blocks BB_2 and BB_3 together dominate basic block BB_9, hence, enabling dynamic CSE of operation 2. In this manner, we use this property of domination by sets of basic blocks while performing dynamic CSE along with code motions such as reverse and conditional speculation that duplicate operations into multiple basic blocks.

Another case, in which dynamic CSE is applied in conjunction with conditional speculation, arises when an operation is duplicated into a basic block in which another operation with the same expression already exists. In this case, the operation being duplicated is instead replaced with a copy operation using the result of the already present operation with the same expression. Hence, in the example in Figure 10(a), if there had existed an operation with the same sub-expression $(b+c)$ as operation 1 in the basic block BB_2, then dynamic CSE would have eliminated the common sub-expression when operation 1 got duplicated as operation 3 into BB_2. Using these various approaches of applying CSE dynamically during scheduling can significantly reduce the number of operations in the final scheduled design, as demonstrated in the results section (see Section 10).

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

In this way, the scheduling heuristic can dynamically employ compiler transformations such as speculative code motions, dynamic CSE and dynamic copy propagation, to increase resource utilization and improve the quality of synthesis results. These transformations form an integral part of the scheduling strategy employed by the *Spark* framework. Besides these compiler transformations, scheduling also employs several high-level synthesis transformations such as operation chaining. This transformation is discussed in the next section.

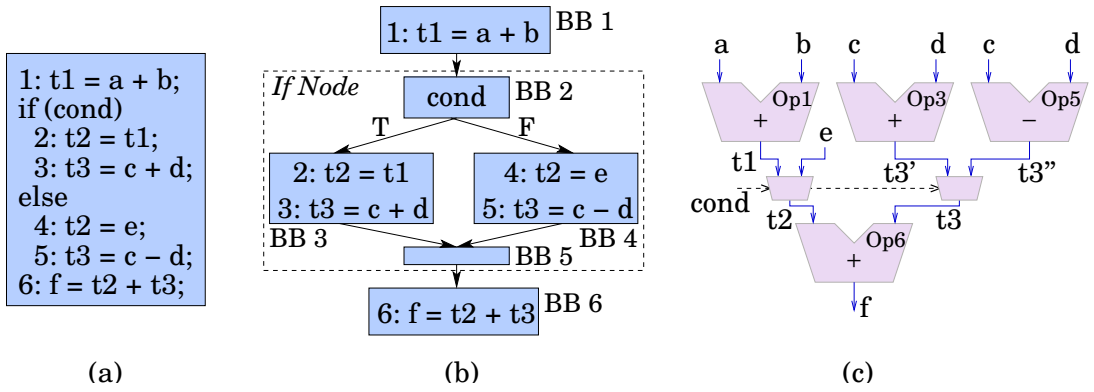

Figure 11. **An example of operation chaining across conditional boundaries; (a) sample "C" code, (b) its HTG representation, (c) corresponding hardware, with functional units connected via steering logic.**

6.3 Chaining Operations Across Conditional Boundaries

Operation chaining is an important technique that is supported by most high-level synthesis tools [10]. *Chaining* of operations means that the result of one operation is used immediately by the next operation without storing it in an intermediary latch or register. In the corresponding hardware, the functional units, on to which the operations are mapped, have to be connected to each other without any memory elements in between. However, in designs that are a mix of control and data operations, operations may be chained across basic blocks, i.e., across conditional boundaries. In hardware, this means that the functional units are connected via steering logic such as multiplexors.

Let us understand this with the aid of an example. Consider the sample fragment of "C" code in Figure 11(a) and the corresponding HTG representation in Figure 11(b). Consider also that this design description has to be scheduled in one cycle. To achieve this, all the operations in the description have to be chained together, across the if-then-else conditional block. One possible hardware implementation for this is shown in Figure 11(c). The operations Op_1 to Op_6 correspond to the line numbers in Figure 11(a). In the circuit in Figure 11(c), the inputs to the operation Op_6 are obtained by multiplexing the outputs of the Op_1, Op_3 and Op_5, based on the condition *cond*. Variables $t3'$ and $t3''$ are the temporary results of operations Op_3 and Op_5, that are immediately multiplexed to produce the result $t3$. Since all the operations in this fragment of code are chained together, none of the variables, $t1$, $t2$ and $t3$, have to be stored in registers. We will discuss these new types of variables that are not stored in registers in Section 6.3.2.

Hence, chaining operations across conditional boundaries has two effects on the scheduling strategy: firstly, the scheduling heuristic has to keep track of the resource utilization of multiple scheduling steps in several basic blocks that are chained into the same clock cycle. Thus, the scheduler has to use a modified resource utilization and operation scheduling model that looks across the conditional boundaries. Secondly, chaining an operation with operations that are in the branches of a conditional check requires a detailed analysis of the control flow paths in which the chained operations are, as discussed in the next section.

6.3.1 Operation Chaining with Operations in the Branches of a Conditional Block

To be eligible for chaining across a conditional boundary, an operation has to satisfy two main criteria: (a) a resource on which the operation can execute should be idle in all the scheduling steps chained together, including the current scheduling step, and (b) if there are operations in the steps being chained together that the current operation being scheduled has dependencies with, then the total execution time of these chain of operations should be less than the clock period of the design. Note that a scheduling step is equivalent to a statement node (see Section 4) and represents an aggregation of operations that execute in the same cycle within a basic block.

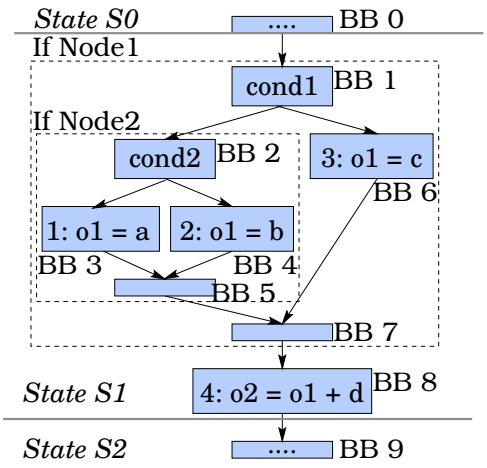

Figure 12. **Operation** 4 **is scheduled in the same cycle as operations** 1**,** 2 **and** 3**. Hence, we have to check that chaining is possible on all *chaining trails* up from** BB_8**.**

We explain these two criteria using the example in in Figure 12; we want to schedule operation 4 in the same cycle as operation 1. First, the chaining heuristic determines all the basic blocks that have scheduling steps scheduled in the same cycle as the current step under consideration. The heuristic does this by traversing all the paths or *chaining trails* back wards from the basic block that operation 4 is in (BB_8), looking for scheduling steps scheduled in the same cycle. In this example, there are three trails comprising the basic blocks: $<BB_8, BB_7, BB_5, BB_3, BB_2, BB_1>$,$<BB_8, BB_7, BB_5, BB_4, BB_2, BB_1>$ and $<BB_8, BB_7, BB_6, BB_1>$. Note that, in this example, since each basic block has just one scheduling step, the basic block name will be used to refer to the corresponding scheduling step in it.

Clearly, the first criteria is satisfied in this case, since none of the operations in the steps being chained together uses an adder (we assume that at least one adder has been allocated to schedule this design). For the second criteria, we determine the dependency chain of operation 4 in each trail. The operations that will be chained with operation 4 in the trails are operations 1, 2 and 3 respectively, each of which writes to the variable $o1$. We determine that operation 4 can be executed in the same cycle as these operations by using the appropriate value of $o1$, depending on the evaluation of the condition. However, to actually chain these operations together, the chaining algorithm along with the code motion algorithm has to ensure that the correct hardware corresponding to the chained operations is generated to implement the schedule, as discussed next.

6.3.2 Creating Wire-Variables to enable Chaining on each Chaining Trail

The *Spark* synthesis tool initially assumes that each variable in the input behavioral description is mapped to a virtual register. In fact, some of these variables may be eliminated after scheduling by dead code elimination. It is only during register binding that a variable life-time analysis pass determines which variables are actually mapped

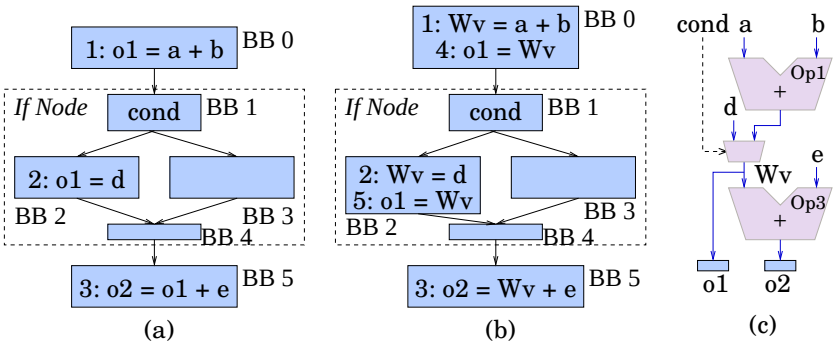

Figure 13. **(a) HTG of an example, (b) operation 3 is chained with operations. 1 and 2; so, wire-variable Wv and copy operations 4 and 5 are inserted (c) corresponding hardware; Wv becomes a wire and $o1$ a register.**

to registers. However, since registers can only be read in the next cycle after being written, to enable operation chaining, we introduce the notion of a *wire-variable*. Wire-variables are explicitly marked as being wires and are not mapped to registers, and thus, can be read in the same cycle as they are written to.

Consider an operation Op_1 that writes a result, r_1 and another operation Op_2 that reads this result:

$$r_1 = Op_1(arguments); r_2 = Op_2(r_1)$$

To chain operations Op_1 and Op_2, the code has to be modified to:

$$temp = Op_1(arguments); r_2 = Op_2(temp); r_1 = temp$$

where variable *temp* is marked as being a wire and r_1 is (potentially) mapped to a register[1].

Often, as was the case in the example in Figure 12, a variable may be written by several operations in different basic blocks. When operations are chained across conditional checks, operations that write to "wire-variables" have to be inserted in all the trails leading back from the chained operation, i.e., in all the branches of the preceding conditional blocks. This is explained by an example in Figure 13(a). In this HTG representation, variable $o1$ is written to by operations 1 and 2 in basic blocks BB_0 and BB_2 respectively. Operation 3 in basic block BB_5 reads the value of this variable to produce another variable $o2$. Consider that the scheduling algorithm schedules the entire fragment of code in this figure within one clock cycle. Then, to enable operation chaining, a wire-variable Wv is introduced and the copy operations 4 and 5 are inserted, as shown in Figure 13(b). In the resulting hardware, shown in Figure 13(c), variable Wv becomes a wire and the variables $o1$ and $o2$ are bound to registers. Operation 3 uses the multiplexed result of both the operations that write to wire-variable Wv. Note that, the copy operation 4 could also have been inserted into basic block BB_3, leading to the same hardware.

Similarly consider the fragment of code in the Figure 14(a). In this example, variable $o1$ is written to only in the true branch of a conditional block and is read by operation 2 in basic block BB_5. This code implies that if the condition evaluates to "false", then a value of $o1$ from a previous write (not shown here) will be used by operation 2. In order to chain the operations in this code, a variable copy to wire-variable Wv has to be inserted in both branches of the conditional block, as shown in Figure 14(b). So, the operation 2 now reads the variable Wv

[1]In the RTL VHDL generated after synthesis, r_1 is mapped to a VHDL signal and *temp* is mapped to a VHDL variable

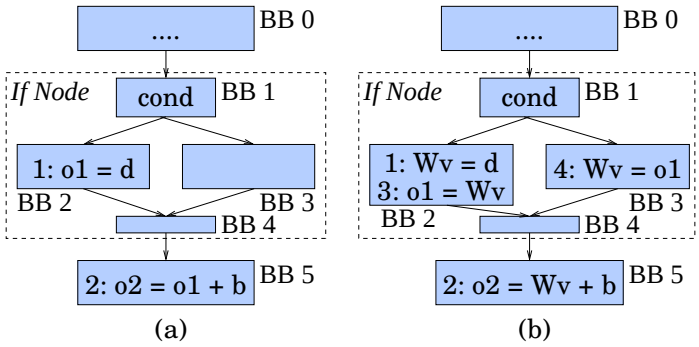

Figure 14. **(a) HTG of another example (b) Wire-variable Wv and copy operations (3 and 4) are added in all chaining trails.**

instead. In hardware, variable Wv will be mapped to a wire and variable $o1$ to a register. In this way, wire-variables are introduced as and when required and a dead code elimination pass later removes any unnecessary variables and variable copies. Hence, the chaining heuristic has to traverse all the chaining trails leading up to the current scheduling step and insert copy operations to wire-variables for all the variables/operands read by the operation being scheduled.

Chaining operations across conditional blocks is particularly useful for the design of low latency blocks such as microprocessor functional blocks [58]. These blocks are usually targeted to an implementation within a single or a few cycles and hence, all the operations in the design description have to be chained together. In other control-intensive designs, situations such as those shown in Figures 13 and 14 are more common, wherein there exist copy operations within conditional branches that assign a value, computed before the conditional, to a variable that is subsequently read by an operation after the conditional block.

In the next section, we show how chaining operations across conditionals and the various compiler transformations presented so far can be integrated into a list scheduling heuristic.

7 Priority-based Global List Scheduling Heuristic

Scheduling is the task of assignment of operations to control steps or time intervals so that the allocated resources can compute the operations assigned to each step [10]. For the purpose of evaluating the various code motion transformations, we have chosen a *Priority-based* global list scheduling heuristic, although the transformations presented here can be applied to other scheduling heuristics as well. Priority list scheduling works by ordering operations to be scheduled based on a *priority* associated with them.

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 one more than 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 hence, operations whose results are read by these outputs have a priority of one and so on. The priority assignment of

operations for the waka benchmark are indicated next to the operations in Figure 15. In this design, the priority assignment of the output operations, *n*, *l* and *k* is 0, and the operations that depend on them have priority 1 and so on. The priority of an operation that creates a conditional check (operations *p* and *q* in the figure), is assigned as the maximum priority in the conditional branches of the If-HTG.

The scheduling heuristic assigns a cost for each operation based on its global priority and favors operations that are on the longest path through the design. In this way, the cost function attempts to minimize the longest delay through the design. It is important to note that minimizing a different cost function, such as average delay can be done by incorporating control flow information into the cost function. Also, if we have profiling information about which control paths are more likely to be taken, then we can give operations on those paths a higher priority than operations on less taken paths.

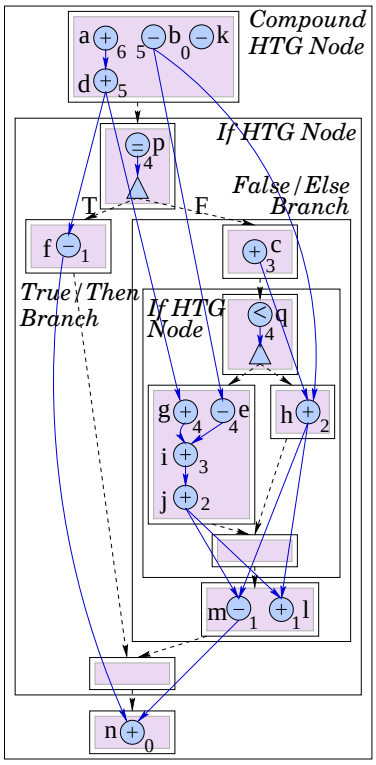

Figure 15. **Priority assignment for the operations in the "waka" benchmark**

The scheduling heuristic is presented in Figure 16(a). The inputs to this heuristic are the unscheduled hierarchical task graph (HTG) of the design and the list of resource constraints. Additionally, the designer may specify a list of allowed code motions (i.e. speculation, reverse speculation, conditional speculation et cetera), whether dynamic variable renaming is allowed, and the code motion technique (percolation or trailblazing) for moving the operations [45]. The heuristic starts by assigning a priority to each operation in the input description as explained above. Then scheduling is done one control or scheduling step at a time while traversing the basic blocks in the hierarchical task graph (HTG). In our implementation, control paths are followed such that at the fork node of a conditional block, the true branch is scheduled first and then the false branch. Within a basic block, each scheduling step corresponds to a statement HTG node in the basic block (see Section 4). The heuristic to get the next scheduling step is explained later in Section 7.1. At each time step in the basic block, a list of *available* operations is collected, for each resource in the resource list, as shown in line 5 in the algorithm in Figure 16(a).

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 16(b). Initially, all unscheduled operations in the HTG 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. However, this basic block traversal algorithm skips over the loop body of any loop nodes it encounters. This is because operations from within loop nodes can only be moved outside the loop body by transformations such as loop-invariant code motion and loop pipelining. Similarly, when the scheduler is

scheduling the loop body of a loop node, available operations are only collected from within the loop body.

Once these unscheduled operations have been collected, operations whose data dependencies are not satisfied and cannot be satisfied by variable renaming are removed from this list. Similarly, operations that cannot be moved in the HTG to the current scheduling step using the *allowed* code motions are also removed from the available list. This list of *allowed* code motions is provided by the user and hence, allows experimentation on the effectiveness of the various code motions [8].

Next, the available operations heuristic determines the list of basic blocks (*bbList*) that the operation op will have to be duplicated into. This list, *bbList*, will be non-empty if the operation op is being conditionally speculated. The heuristic then calls the function $FindResSlot$ for each basic block *bb* in *bbList*; this function determines if there is an idle resource in the basic block that the op can be scheduled on. If there is no idle resource, the code motion heuristic employed later by the scheduler may even instruct this function to create new scheduling steps [59]. If the $FindResSlot$ function is unable to accommodate a duplicated copy of op in any of the basic blocks in *bbList*, then the operation op is removed from the available operation list. This is shown in lines 7 to 9 in the algorithm in Figure 16(b).

Finally, the available operations heuristic calculates a cost for each of the operations that remain in the available operations list. Currently, this cost is the negative of the operations global priority. The scheduling heuristic picks the operation with the *lowest* cost from the available operations list (line 6 of Figure 16(a)). Effectively, this chooses the operation with the highest priority in the available list and hence, favors operations that are on the longest path through the design. More work needs to be done to enhance this cost function to include hardware (control and area) cost models of the code transformations.

The code motion algorithm is then instructed to schedule this operation op with the lowest cost on the current resource (*res*) at the current scheduling step ($step$), by a making a call to the function *MoveOp*. This function is presented later in Section 7.5. Once the chosen operation has been moved and scheduled, the dynamic CSE heuristic comes into play, as shown by the boxed region enclosing lines 8 to 12 in the algorithm in Figure 16(a). From the remaining operations in the available list, dynamic CSE determines the list of operations, *cseOpsList*, that have a common sub-expression with the scheduled operation op. Then, for each operation *cseOp* in *cseOpsList*, if the basic block of *cseOp* is dominated by the basic block of op *after* scheduling, then the common sub-expression in *cseOp* is replaced with the result from op (by calling $ApplyCSE$).

We illustrate the dynamic CSE heuristic using the earlier example from Figure 9(a). In this example, consider that while scheduling basic block BB_1, the scheduling heuristic determines that available operations are operations 2, 3 and 4. Of these operations, the heuristic schedules operation 2 in BB_1. Then, the dynamic CSE heuristic examines the remaining operations in the available list, namely operations 3 and 4, and detects and replaces the common sub-expression $(b+c)$ in operation 4 with the result, e', of the scheduled operation 5, since $BB(op_5)$ dominates $BB(op_4)$.

Algorithm 1: Priority List Scheduling Heuristic

Inputs: Unscheduled HTG of design, Resource List R

Output: Scheduled HTG of design

1: Calculate Priority Pr of all Operations in HTG
2: $step$ = getNextSchedulingStep(HTG, ϕ)
3: **while** ($step \neq \phi$) **do**
4: **foreach** (resource *res* in Resource List R) **do**
5: Get List of Available Operations $\mathcal{A}$
6: Pick Operation op with lowest cost in $\mathcal{A}$
7: MoveOp(op, *res*, $step$)
8: Get list of operations *cseOpsList* from $\mathcal{A}$ that have common sub-expressions with op
9: **foreach** (operation *cseOp* in *cseOpsList*) **do**
10: **if** (BB(op) dominates BB(*cseOp*) **then**
11: ApplyCSE(*cseOp*, op)
12: **endforeach** */* Dynamic CSE Heuristic*/*
13: **endforeach**
14: $step$ = getNextSchedulingStep(HTG, $step$)
15: **endwhile**

(a)

Algorithm 2: Get List of Available Operations

Inputs: Resource *res*, Scheduling $step$, *AllowedCodeMotions*

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)
4: Remove op from $\mathcal{A}$
5: **if** (op cannot be moved to $step$ using *AllowedCodeMotions*)
6: Remove op from $\mathcal{A}$
7: *bbList* = Basic Blocks that op will be duplicated into
8: **if** (*FindResSlot*(op, any *bb* in *bbList*) = ϕ)
9: Remove op from $\mathcal{A}$
10: Calculate cost of operation op
11: **endforeach**

(b)

Figure 16. **(a) Priority-based List Scheduling Heuristic (b) Determining the list of *Available* operations.**

The entire scheduling procedure is repeated for all the resources in each scheduling step as the basic blocks in the HTG are traversed from top to bottom (HTG traversal is explained in more detail in the next two sections). Since operations with higher priority may be speculated into a basic block, the (lower priority) operations that were originally placed in that basic block by the designer can be left unscheduled. Either new scheduling steps are added to the current basic block to schedule them or if reverse speculation has been enabled, then these operations are reverse speculated into the subsequent conditional branches (if possible).

Scheduling of loops is done in the same manner. However, user-specified loop transformations such as loop unrolling et cetera 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. The *Spark* framework can schedule all types of loops, including those with unknown loop iteration bounds. 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 is satisfied or not. Hence, loop bounds are not required for generating correct, synthesizable VHDL. However, 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.

7.1 Algorithm to Get the Next Scheduling Step

The scheduling heuristic in *Spark* schedules the design by traversing the HTG of the design in a top-down manner starting at the first (*Start*) node of the design level HTG and walking down till the last (*Stop*) node in this HTG. For getting the steps to schedule in the design, the scheduler calls the algorithm given in Figure 17(a). This algorithm starts by determining the current basic block, *currentBB*, that the current scheduling step, *step*, is in. If this is the first call to the algorithm (i.e. *step* is ϕ), then the *currentBB* is assigned as the first basic block in the top level HTG of the design. The next scheduling step is then the scheduling step after the current *step* in *currentBB* (line 6 in the algorithm). If *step* is currently empty, then *nextStep* is the first step in *currentBB*.

Next, the algorithm checks if *nextStep* is empty; this happens when the current scheduling step, *step*, is the last scheduling step in *currentBB*. In this case, the algorithm should traverse the HTG and get the next basic block in the HTG to schedule. However, it is at this point that we employ a novel technique that inserts new scheduling steps in conditional branches that have fewer scheduling steps [59].

Often design descriptions are such that one conditional branch in an if-then-else HTG node has fewer scheduling steps than the other. We call this an if-HTG with *unbalanced* conditional branches. In such unbalanced if-HTGs, it is possible to insert a new scheduling step in the branch with fewer scheduling steps, without increasing the length of the longest path through the if-HTG. This can increase the opportunities to schedule operations by conditional speculation (i.e. by duplicating operations into both branches of the conditional block). The procedure in the boxed section in the algorithm in Figure 17(a) inserts new scheduling steps to better balance out the branches of conditional blocks.

As shown in line 7, when *nextStep* is empty, the algorithm determines if the *currentBB* has a complementary basic block, *complementBB*. A *complementBB* exists if *currentBB* is in a if-HTG node; if the *currentBB* is in the true branch, then its *complementBB* is the false branch and vice versa. If a *complementBB* exists *and* if it has already been scheduled *and* it has more scheduling steps than *currentBB*, then the algorithm creates a new scheduling step in *currentBB*. (lines 8 through 12 in Figure 17(a)). The reason that only a scheduled *complementBB* is considered is to have an accurate picture of the resource utilization in that basic block before adding any more scheduling steps to the design.

If a new scheduling step is not created in the *currentBB* and the *nextStep* is still empty (line 13), then the algorithm proceeds to get the next basic block in the HTG 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* (line 16 in Figure 17(a)).

The new scheduling step insertion procedure can be explained by the example in Figure 18(a). While scheduling this design, the scheduling heuristic schedules the true branch, i.e., basic block BB_2 first, followed by the false

Algorithm 3: Get Next Scheduling Step

Inputs: HTG of design, Current Scheduling Step *step*

Output: Next Scheduling Step *nextStep*

```
1: if (Scheduling step step = ϕ) then
2:    currentBB = getFirstBasicBlock(HTG)
3: else
4:    currentBB = getBasicBlockOf(step)
5: endif
6: nextStep = Scheduling step after step in currentBB
7: if (nextStep = ϕ) then
8:    complementBB = getComplementBB(currentBB)
9:    if (complementBB ≠ ϕ and is scheduled) then
10:      if (numOfStepsInBB(currentBB) <
               numOfStepsInBB(complementBB)) then
11:        nextStep = createNewStepInBB(currentBB)
12: endif        /* Balance Conditional Branches */
13: if (nextStep = ϕ) then
14:    nextBB = getNextBasicBlock(currentBB)
15:    if (nextBB ≠ ϕ) then
16:       nextStep = First schduling step in nextBB
17: endif
18: return nextStep
```

(a)

Algorithm 4: Get Next Basic Block

Inputs: HTG of design, Current Basic Block *currentBB*

Output: Next Basic Block *nextBB*

Static: Basic Block Queue *bbQueue*

```
1: nextTrueBB = getNextTrueBB(currentBB)
2: if (nextTrueBB ≠ ϕ) then
3:    isJoinBB = isThisJoinBB(nextTrueBB)
4:    bbVisited = isBBMarkedVisited(nextTrueBB)
5:    if (NOT(bbVisited) and NOT(isJoinBB)) or
            (bbVisited and isJoinBB) then
6:     bbQueue->pushBack(nextTrueBB)
7:    if (NOT(bbVisited)) then
8:     markBBAsVisited(nextTrueBB)
9: endif
10: nextFalseBB = getNextFalseBB(currentBB)
11: if (nextFalseBB ≠ ϕ and has not been visited) then
12:    markBBAsVisited(nextFalseBB)
13:    bbQueue->pushBack(nextFalseBB)
14: endif
15: return (nextBB = bbQueue->popFront())
```

(b)

Figure 17. **(a) Get Next Scheduling Step Algorithm; (b) Get Next Basic Block Algorithm**

branch, i.e., BB_3. So, if the design in this example is allocated two resources, namely, an adder and a subtracter, then the resulting design after scheduling is as shown in Figure 18(b).

This figure shows that, after scheduling, the false branch of the if-then-else HTG node has fewer scheduling steps than the true branch. Hence, the procedure outlined above inserts a new scheduling step in basic block BB_3. This new step and the presence of a corresponding idle resource in the other branch (BB_2) of the if-HTG node, enables the conditional speculation of operation "*e*", as operations "e_1" and "e_2" in basic blocks BB_2 and BB_3 respectively. The resulting design is shown in Figure 18(c).

This example illustrates how inserting new scheduling steps in the shorter of the two branches of a conditional block can enable code motions such as conditional speculation, without increasing the longest path through the conditional block. This can lead to shorter schedule lengths for the design and improve resource utilization in

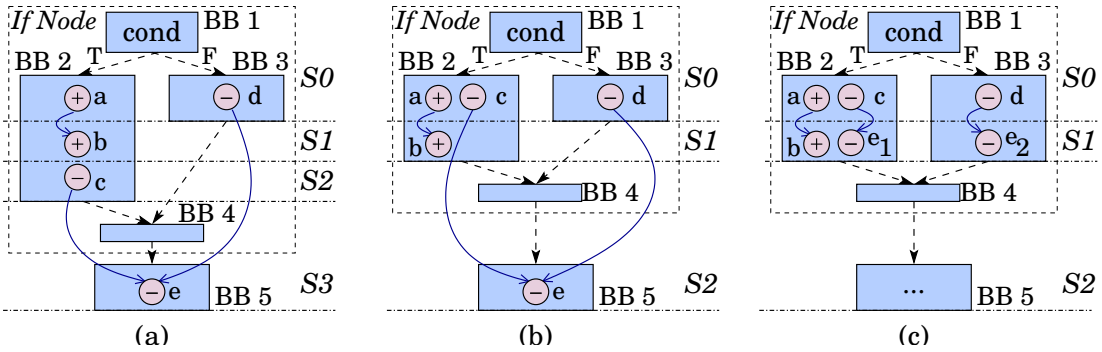

Figure 18. **(a) HTG representation of an example, (b) After scheduling basic block BB_2, (c) Insertion of a new scheduling step in basic block BB_3 enables conditional speculation of operation *e*.**

the conditional block, while at the same time leading to better balanced conditional branches. Also, if profiling information is available, this heuristic can be modified so that it does not add new scheduling steps in basic blocks on control paths that are more likely to be taken.

7.2 Algorithm to Get the Next Basic Block to Schedule

The algorithm to get the next basic block in the HTG is given in Figure 17(b). This algorithm traverses the basic blocks in the design HTG in a top down manner starting at the *Start* node of the design level HTG. Each basic block in the HTG design may have two successor basic blocks; one that is reached by traversing the "true" path and the other by traversing the "false" path. This nomenclature arises to accommodate basic blocks that are the condition basic blocks of if-then-else blocks or loops (see Section 4). Hence, if the current basic block is a condition basic block and if its condition evaluates to true, the program flow traverses down the "true" path, else it traverses the "false" path. For basic blocks that are *not* condition basic blocks, only the default true path exists. The last basic block in the design has no successor basic blocks; this is *Stop* node of the design level HTG.

The algorithm in Figure 17(b) maintains a queue of basic blocks (*bbQueue*) to process. It gets as input the current basic block, uses this to update the basic block queue and finally, returns the next basic block from the queue. This algorithm starts off by looking at the next basic block on the true path as shown in line 1 of in Figure 17(b). If this *nextTrueBB* exists and if this basic block has not been visited before, it is pushed into the basic block queue. However, basic blocks that are join nodes of if-then-else HTG nodes, are treated in a special manner. *Join* basic blocks are visited twice; once from the true branch of the if-then-else and once from the false branch. Hence, the first time they are visited (from the true branch), they are marked as having been visited, however, they are not pushed onto the *bbQueue*. It is pushed into the queue on the second visit (from the false branch), when the join basic block has already been marked as "visited" from an earlier visit (lines 3 to 8 of the algorithm). This is done so as to schedule hierarchical nodes such as if-then-else blocks completely, before proceeding to subsequent nodes or basic blocks. Also, although not shown in Figure 17(b), this algorithm does not traverse true paths that are the *backward* control flow edge of a loop, i.e., the edge that iterates over the loop body.

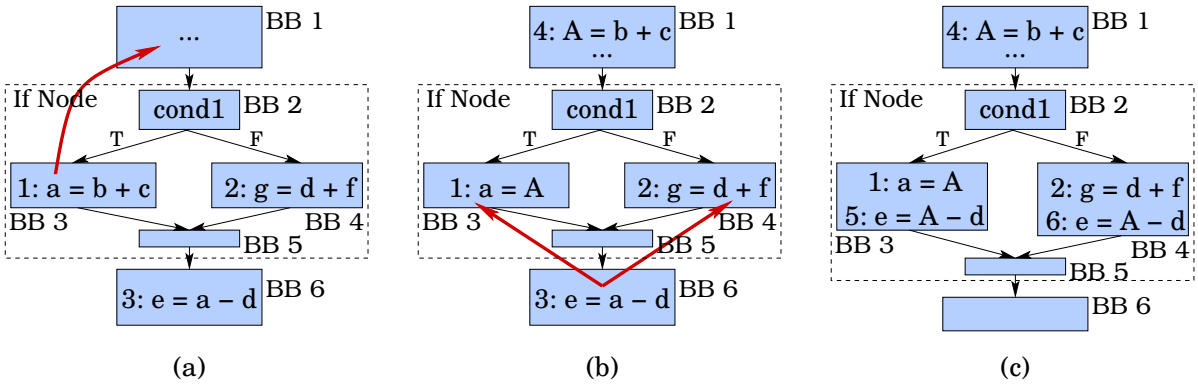

Figure 19. **(a) HTG representation of an example (b) Operations** 1 **is speculatively executed as operation** 4 **in** BB_1 **(c) Operation** 3 **is conditionally speculated into conditionals** BB_3 **and** BB_4**. Also shown is the dynamic renaming of variable** *a* **with the speculatively calculated value** *A* **in operation** 5**.**

Similarly, if a successor basic block exists on the "false" path of the queue and it has not been visited earlier, it is pushed into the queue. The algorithm returns the first entry in the front of the queue as the next basic block to schedule. When the last basic block in the HTG of the design has been reached, this algorithm returns an empty next basic block and hence, the algorithm to get the next scheduling step terminates by returning an empty scheduling step. This indicates to the scheduling heuristic that it has finished scheduling the HTG of the design.

7.3 An Illustrative Example of the *Spark* Scheduler

In this section, we will walk through an example to understand how the scheduling heuristic works and particularly to show how code motions are employed by the heuristic. Consider the example in Figure 19(a) and consider that the resources allocated to schedule this design are one adder and one subtracter. The first node in the HTG of this example is basic block BB_1. The scheduler starts by scheduling on the adder in basic block BB_1. Lets say that among the available operations for the scheduling step in BB_1, operation 1 from basic block BB_3 is chosen for scheduling. The code motion technique determines that it has to speculate this operation in order to schedule it in BB_1. The resultant design is shown in Figure 19(b). In this figure, operation 1 is speculatively executed as operation 4 in BB_1 and the result of operation 4, variable *A*, is still written back to variable *a* in operation 1 in basic block BB_3. This is to ensure that variable *a* gets updated with the result *A* only if the condition *cond*1 evaluates to "true".

Next, the scheduler receives basic block BB_3 to schedule from the HTG traversal algorithms in Figure 17(b) (since BB_2 does not have any operations in it). At this point, initially operation 3 will be picked among the available operations. However, this operation requires conditional speculation to be scheduled in basic block BB_3 and as per our conditional speculation heurisitic [45], we only allow conditional speculation when the basic block in the other conditional branch of the if-then-else has already been scheduled. In other words, in this case, we cannot allow conditional speculation of operation 3 into BB_3, since basic block BB_4 has not yet been scheduled. Thus, only the copy operation, operation 1, is scheduled into basic block BB_3.

The scheduler receives basic block BB_4 to schedule next, as per the algorithm in Figure 17(b). Operation 2 is scheduled on to the adder. The scheduler then determines that operation 3 can be scheduled on the subtracter in this scheduling step by conditionally speculating it into basic blocks BB_3 and BB_4. The conditional speculation is allowed this time around because the basic block in the other conditional branch of the if-then-else node, namely, BB_3, has already been scheduled and has an idle resource (subtracter) on which operation 3 can be scheduled.

The resultant design is as shown in Figure 19(c). Operation 3 has been duplicated as operations 5 and 6 in basic blocks BB_3 and BB_4. In basic block BB_3, operation 5 directly uses the speculatively calculated value A of operation 1 by employing dynamic renaming [45]. Finally, since basic blocks BB_5 and BB_6 are empty and there are no more unscheduled operations, the scheduling heuristic terminates.

This illustrative example demonstrates the working of the scheduling heuristics presented so far. In the next section, we show how synthesis transformations such as chaining can be incorporated into the various algorithms of the scheduling heuristic.

7.4 Incorporating Chaining into the Scheduling Heuristic

Chaining can be incorporated into the scheduling heuristic and the heuristic to get the available operations as shown by the boxed sections of these two heuristics in Figure 20 (a) and (b) respectively. The modified priority-based list scheduling heuristic keeps track of not only the current scheduling step, but also the previous scheduling step, *prevStep*. Chaining across conditional boundaries is attempted if the current scheduling *step* is in a basic block different from the basic block that the *prevStep* was in. This is because chaining of operations within a basic block, i.e., with no control flow between them, is done within the same scheduling step. Note that although not shown in the algorithm in this figure, if the scheduling heuristic fails to schedule anything on *step* with chaining across conditional boundaries enabled, then it tries to schedule on *step* again, albeit *without chaining*.

When chaining across conditional boundaries is enabled, the scheduler determines all the steps in previous basic blocks that the current step has to be chained with (*stepsToChainWith*). This is done by the *getChainSteps* function; this function (not shown here) traverses back up all the control paths leading up to the current basic block, looking for steps scheduled in the same cycle as the current scheduling step. If chaining is not enabled, then *stepsToChainWith* is empty.

With chaining enabled, the scheduler skips over any resource *res* in the resource list that is used in any of the *stepsToChainWith*. This is shown in lines 8 to 10 in Figure 20(a). If the resource is available for scheduling, then the scheduling heuristic proceeds as before and calls the heuristic to collect available operations (see Section 7).

The *Available Operations* heuristic also requires a modification to enable chaining across conditional boundaries as shown by the boxed section in Figure 20(b). If *stepsToChainWith* is not empty, then this heuristic inspects each step, *chainStep*, in the *stepsToChainWith* to determine if the current operation under consideration, *op*, is dependent on any operations in *chainStep*. If it is, then the total run time of the current operation is calculated

Algorithm 1': Scheduling Heuristic with Chaining

Inputs: Unscheduled *HTG* of design, Resource List *R*

Output: Scheduled *HTG* of design

1: Calculate Priority *Pr* of all Operations in HTG
2: *prevStep* = *step* = getNextSchedulingStep(*HTG*, ϕ)
3: **while** (*step* $\neq \phi$) **do**
4: **if** (basicBlock(*step*) $\neq$ basicBlock(*prevStep*)) **then**
5: *stepsToChainWith* = getChainSteps(*step*)
6: **endif** */* Determine whether to do Chaining */*
7: **foreach** (resource *res* in Resource List *R*) **do**
8: **if** (isResUsed(*res*, *stepsToChainWith*)) **then**
9: Continue to next resource in foreach loop
10: **endif** */* Can Chaining be done on res */*
11: Get List of Available Operations $\mathcal{A}$
12: Pick Operation *op* with lowest cost in $\mathcal{A}$
13: MoveOp(*op, res, step*)
14: PerformDynamicCSE($\mathcal{A}$, *op*)
15: **endforeach**
16: *prevStep* = *step*
17: *step* = getNextSchedulingStep(*HTG*, *step*)
18: **endwhile** (a)

Algorithm 2': Get List of Available Operations

Inputs: Resource *res*, Scheduling *step*, *AllowedCMs*, *stepsToChainWith*

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* not satisfied)
4: remove *op* from $\mathcal{A}$
5: **if** (*op* cannot be moved to *step* with *AllowedCMs*)
6: remove *op* from $\mathcal{A}$
7: *FindResSlot* for *op* in each *bb* of basic block list *bbList* that *op* will be duplicated into
8: **foreach** (*chainStep* in *stepsToChainWith*) **do**
9: *TotalRunTime* = run time of *res* + run time of ops in dependency chain of *op* in *chainStep*
10: **if** (*TotalRunTime* > *clockPeriod*) **then**
11: remove *op* from $\mathcal{A}$
12: **endforeach** */* Can op be chained in step */*
13: Calculate cost of operation *op*
14: **endforeach** (b)

Figure 20. **Incorporating Chaining into the (a) Priority-based List Scheduling Heuristic (b) *Available* operations Algorithm. Although not shown here, if scheduling on a step with chaining enabled fails, then the same step is scheduled again without chaining.**

as the summation of the execution/run time of the resource *res* and the execution time of the dependency chain of operations in *chainStep*. If this *TotalRunTime* is greater than the clock period allocated to the design, then the operation under consideration is removed from the available operations list (lines 7 and 8 in Figure 20(b)). Note that the resource execution times and the clock period of the design are specified by the user in a hardware description file that the *Spark* tool reads during its initialization.

The final modification required in the scheduling framework to enable chaining operations across conditional boundaries is in the code motion heuristic as explained in the next section.

7.5 Incorporating Chaining into the Code Motion Technique

Once the scheduling heuristic decides to schedule an operation *op* on a scheduling *step*, it calls the code motion technique to actually move this operation. It is the code motion technique's responsibility to take care of operation

Algorithm 5: MoveOp: TrailSynth Technique

Inputs: Operation *op*, Scheduling *step*

Output: Operation *op* is moved to *step*

1: *TrailList* = findTrails(*op*, *step*)
2: *targetBB* = getBasicBlockOf(*step*)
3: **foreach** (*trail* in *TrailList*) **do**
4: *lastBBInTrail* = last Basic Block on *trail*
5: **if** (*lastBBInTrail* = *targetBB*) **then**
6: *trailStep* = *step*
7: Insert *op* in *trailStep*
8: **else**
9: *trailStep* = FindResSlot(*op*, *lastBBInTrail*)
10: Insert *duplicate* of *op* in *trailStep*
11: **endif**
12: Update data dependencies effected by *op* move
13: **if** (*trailStep* chained across conditionals) **then**
14: ChainOpWithPrevSteps(*op*, *trailStep*)
15: **endforeach**

(a)

Algorithm 6: ChainOpWithPrevSteps

Inputs: Operation *op*, Scheduling *trailStep*,

Output: Inserts Wire-Variables in all chained steps

1: *stepsToChainWith* = getChainSteps(*trailStep*)
2: *chainingTrailList* = getChainingTrails(*trailStep*)
3: *LeftWv* and *RightWv* = New Left and Right Operand Wire-Variables for *op*
4: **foreach** (*chainTrail* in *chainingTrailList*) **do**
5: *depOpList* = findDependentOps(*op*, *chainTrail*)
6: **foreach** (*depOp* in *depOpList*) **do**
7: **if** (*depOp* writes to left operand of *op*) **then**
8: *Wv* = *LeftWv*
9: **else** /* *depOp* writes to right operand of *op* */
10: *Wv* = *RightWv*
11: Make *depOp* write to wire-variable *Wv*
12: Insert copy operation from *Wv* to original variable of *depOp* in its scheduling step
13: **endforeach**
14: **endforeach**

(b)

Figure 21. **(a)** *TrailSynth***: Trailblazing code motion technique modified for high-level synthesis (b) Chaining heuristic that inserts wire-variables into all chaining trails**

duplication if required and update the data dependencies effected by the code motion. At the same time, the chaining heuristic inspects each scheduling step into which the scheduled operation has been moved or duplicated and inserts wire-variables if the scheduling step is chained with any other step into the same cycle.

In the *Spark* framework we have implemented two code motion techniques: *Percolation* scheduling [60, 7] and *Trailblazing* [47]. However, the *Spark* framework primarily employs the trailblazing technique to perform code motions due to its better performance and lower compensation code overheads compared to percolation [45]. *Trailblazing* is a code motion technique that uses the structured and hierarchical nature of hierarchical task graphs to perform efficient operation moves [47]. Hierarchical Task Graphs (HTGs), as explained in Section 4, structure the input description's operations and global information so that non-incremental moves can be made without visiting every operation that is bypassed. 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.

The code motion heuristic incorporating trailblazing is presented in Figure 21(a). We call this heuristic *Trail-*

Synth. The TrailSynth heuristic starts by calling the basic trailblazing algorithm; this algorithm is not presented here and is similar to the algorithm presented in the context of compilers in [61]. The trailblazing algorithm returns a list of *trails*; there is a trail for each control path that leads from the current basic block that operation *op* is in, to the scheduling step, *step*, that the operation *op* is being scheduled on **or** to one of the basic blocks on to which the operation will be duplicated.

The code motion heuristic examines each trail in the list of trails (*TrailList*) and inserts the operation into the last basic block on each trail (lines 3 to 11 in Figure 21(a)). Multiple trails indicate that the operation has to be duplicated into the basic block that each trail ends in, namely, *lastBBInTrail*. If the trail ends in the basic block that the scheduling step (*step*) is in, i.e., the "target" basic block (*targetBB*), then the operation itself is inserted into *step*. For the other trails, the heuristic calls a procedure that finds a scheduling step (*trailStep*) in the *lastBBInTrail* with an un-utilized resource on which operation *op* can execute. This procedure, *FindResSlot*, may also insert a new scheduling step into a basic block if no scheduling step is found with an empty resource slot [59]. A duplicate copy of *op* is then inserted into this *trailStep* in *lastBBInTrail*. This is shown in lines 9 and 10 of the heuristic.

After the operation has been inserted into the current trail being examined, the code motion heuristic calls a procedure that updates the data dependencies affected by the operation move and/or duplication (line 12 in Figure 21(a)). It is at this point that the chaining heuristic comes into play. If the scheduling step on the current trail (*trailStep*) that the operation has been moved or duplicated into, is chained across its previous conditional boundaries, then the code motion heuristic calls the chaining function *ChainOpWithPrevSteps* presented in Figure 21(b). This function inserts wire-variables into all the chaining trails (*chainingTrailList*) that lead up to the *trailStep*. As explained in Section 6.3.1, chaining trails consist of the basic blocks that have a control-flow path to the basic block of the current scheduling *step* and have a scheduling step scheduled in the same cycle as the current scheduling *step*.

The chaining heuristic in Figure 21(b) first creates the new wire-variables, *LeftWv* and *RightWv*, for the left and right operands, respectively, of the operation being scheduled, *op* (line 3). Then, for each chaining trail, *chainTrail*, in the list of chaining trails, *chainingTrailList*, the heuristic calls the function *findDependentOps*. This function finds the list the operations (*depOpList*) in *chainTrail* that the current operation *op* has a dependency with. Each dependent operation, *depOp*, in this list is checked and if *depOp* writes to the left operand of *op*, then the result of *depOp* is written to *LeftWv* instead. Conversely, if *depOp* writes to the right operand of *op*, its result is written to *RightWv* instead. Also, a copy operation from *LeftWv* or *RightWv* to the original variable that *depOp* wrote to, is inserted after *depOp* in its scheduling step. This entire procedure is outlined within the *foreach* loop in lines 5 to 13 in Figure 21(b).

In this way, this chaining heuristic inserts write operations to wire-variables into the scheduling step, *trailStep*, of each trail that the scheduled operation is duplicated or moved into. Note that, if the *depOpList* is empty for any

Benchmark	# Basic Blocks	# Operations	# Resources
MPEG-1 *pred*2	45	287	$2+-,1*,2<<,2==,2[]$
MPEG-1 *pred*1	17	123	$2+-,1*,2<<,2==,2[]$
MPEG-2 $dpframe\_est$	61	272	$4+-,1*,2<<,2==,2[]$
GIMP tiler	35	150	$3+-,1/,1*,2<<,2==,2[]$

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

chaining trail in *chainingTrailList*, the chaining heuristic will insert simple copy operations from both the original left operand of op to its left operand wire-variable ($LeftWv$) and from its original right operand to *RightWv* (as explained earlier in the example in Figure 14). Hence, all the chaining trails will now have writes to these wire-variables and the scheduled operation op reads the wire-variables instead of its original operands. Any unnecessary copy operations are eliminated by a dead code elimination pass performed after scheduling.

8 Experimental Setup

We have implemented the scheduling heuristic along with the pre-synthesis transformations and synthesis and compiler transformations presented in this paper in the *Spark* high-level synthesis framework. *Spark* provides the ability to control the various code transformations by user-defined scripts and command-line options. This enables us to experiment with the various transformations presented in this paper. In this section, we present the results for these experiments and demonstrate the utility of these transformations in improving the quality of synthesis results.

We have chosen three large and moderately complex real-life applications, representative of the multimedia and image processing domains, to perform our experiments, namely, the MPEG-1 algorithm [62], the MPEG-2 algorithm [63] and the GIMP image processing tool [64]. From these applications, we have taken a few designs that are relatively control-intensive. These designs consist of two functions from the *Prediction* block of the MPEG-1 algorithm, one function from the *Motion Estimation* block of the MPEG-2 algorithm and one function from the GIMP image processing tool. The MPEG-1 functions used are the *pred*1 and *pred*2 functions, the MPEG-2 function is the $dpframe\_estimate$ function and GIMP function is the $tile$ function (with the scale function inlined) from the "tiler" transform [2].

Table 1 lists the characteristics of the various designs used in terms of the number of non-empty basic blocks and the number of operations in the input description. The number of basic blocks is indicative of the control complexity of the design. 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

[2]Note that this floating point function has been arbitrarily converted to an integer function for the purpose of our experiments. This does not affect the nature of the control flow, but only the type of data that is handled.

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 4 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). The longest path through a if-then-else conditional block is the cycles on the longer branch and 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 LSI-10K 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 that 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 [8, 43]. 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 paper can further improve the synthesis results. We start with the pre-synthesis transformations.

9 Results for Pre-Synthesis Optimizations

9.1 Function Inlining

In our discussion of source level transformations, we have left out one important coarse grain source-to-source transformation, namely, *function inlining*. Function inlining is a transformation that replaces a call to a function by an instance of the function itself. This transformation is usually applied to increase the scope of application of other compiler transformations. Although this transformation has not been implemented in the *Spark* framework, we have applied it manually to the MPEG-1 designs and to the tiler transform from the GIMP. To demonstrate the effectiveness of function inlining, we present scheduling results for the MPEG-1 designs.

Both the functions, *pred*2 and *pred*1, of the MPEG-1 design call the function "calcid" at the start of doubly nested loops. For this reason and because *calcid* is a small function that consists of only straight-line code (no control), it is ideal for inlining. Table 2 presents the scheduling results before and after inlining the *calcid* function

Transformation Applied	*MPEG*-1 *pred*2		*MPEG*-1 *pred*1	
	# States	# cycles	# States	# cycles
Initial Code	84	4059	44	1825
After inlining	85(+1.2%)	4123(+1.6%)	43(-2.3%)	1761(-3.5%)

Table 2. **Results before and after inlining the** *calcid* **function for the MPEG-1** *pred*2 **and** *pred*1 **functions. Before inlining** *calcid* **has the resource allocation shown in Table 1. After inlining, the operations from the inlined** *calcid* **function share the resources with the existing operations in the** *pred*2 **and** *pred*1 **functions.**

Transformation Applied	*MPEG*-1 *pred*2		*MPEG*-1 *pred*1	
	# States	# cycles	# States	# cycles
Initial Code + LICM + CSE	74	2974	41	1323
After inlining + LICM + CSE	75(+1.4%)	2702(-9.1%)	41(0%)	1155(-12.7%)

Table 3. **Results before and after inlining the** *calcid* **function and applying loop-invariant code motion (LICM) and common sub-expression elimination (CSE) for the MPEG-1** *pred*2 **and** *pred*1 **functions. Recall that before inlining** *calcid* **has a full set of resources exclusively to itself.**

into the *pred*2 and *pred*1 functions. These results show only modest improvements in number of states and longest path cycles due to inlining and as a matter of fact, the number of cycles increases for the *pred*2 design. However, this is because when *calcid* is synthesized separately from the *pred*1 and *pred*2 designs, it has the same number of resources allocated to it as the *pred*2 and *pred*1 designs (see Table 1). But after inlining, the operations that were part of *calcid* now have to share the resources allocated to the *pred*1 and *pred*2 functions with the existing operations in these two designs. Hence, the scheduling results in Table 2 demonstrate that after inlining, the speculative code motions are able to schedule these two functions with *calcid* inlined in almost the same cycles – using fewer resources – than when *calcid* is not inlined and has a set of resources allocated exclusively to it.

To demonstrate that inlining indeed increases the scope of application of other parallelizing transformations, we present the results for the same experiments, with loop-invariant code motion (LICM) and common sub-expression elimination (CSE) enabled. These results are presented in Table 3. The first row lists the results when these two transformations are applied before inlining and the second row lists the results after inlining.

The number of states in the controller is constant for the *pred*1 design and increases by 1 for the *pred*2 design after inlining. However, the number of cycles on the longest path is considerably less when these transformations are applied after inlining rather than before inlining. We have found that this gap between the non-inlined design and the inlined design continues to increase as the number of code optimizations applied to the designs increases.

Based on these results, for the rest of the experiments presented in this paper, we use the inlined versions of the *pred*2 and *pred*1 functions as the "baseline" case. Also, as mentioned earlier, the tiler function from the GIMP also has the "scale" function inlined. These inlining decisions have been made by inspecting the design and based

Transformation Applied	*MPEG*-1 *pred*2		*MPEG*-1 *pred*1	
	# States	Long Path	# States	Long Path
Baseline	85	4123	43	1761
with Loop Inv CM	102(+20.0%)	3394(-17.7%)	53(+23.3%)	1461(-17.0%)
with CSE	73(-14.1%)	3355(-18.6%)	38(-11.6%)	1441(-18.2%)
with LICM + CSE	75(-11.8%)	2702(-34.5%)	41(-4.7%)	1155(-34.4%)

Table 4. **Results after applying pre-synthesis transformations on the MPEG-1** *pred*2 **and** *pred*1 **functions**

Transformation Applied	*MPEG*-2 *dpframe_est*		*GIMP tiler*	
	# States	Long Path	# States	Long Path
Baseline	56	684	43	3031
with Loop Inv CM	66(+17.9%)	682(-0.3%)	48(+11.6%)	3163(-21.5%)
with CSE	56(0%)	626(-8.5%)	29(-32.6%)	2631(-34.7%)
with LICM + CSE	56(0%)	599(-12.4%)	29(-32.6%)	2334(-42.1%)

Table 5. **Results after applying Pre-Synthesis transformations on the** *dpframe_estimate* **function from the MPEG-2 Motion Estimation block and the** *tiler* **function from the Gimp Image Processing tool**

on experimentation when it became clear that inlining would significantly enhance the opportunities to apply the transformations in the *Spark* toolkit.

9.2 Scheduling Results for Pre-Synthesis Optimizations

Tables 4 and 5 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 (all code motions enabled along with copy propagation and dead code elimination); the second row for when only loop-invariant code motion (LICM) is applied, the third row for when only common sub-expression elimination (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 by 11 to 23 %, while the cycles on the longest path through the design decrease by up to 21 % (for *tiler*). This 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 leads to the lower cycles on the longest path. 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 increase in performance due to reduced longest path cycles in the next

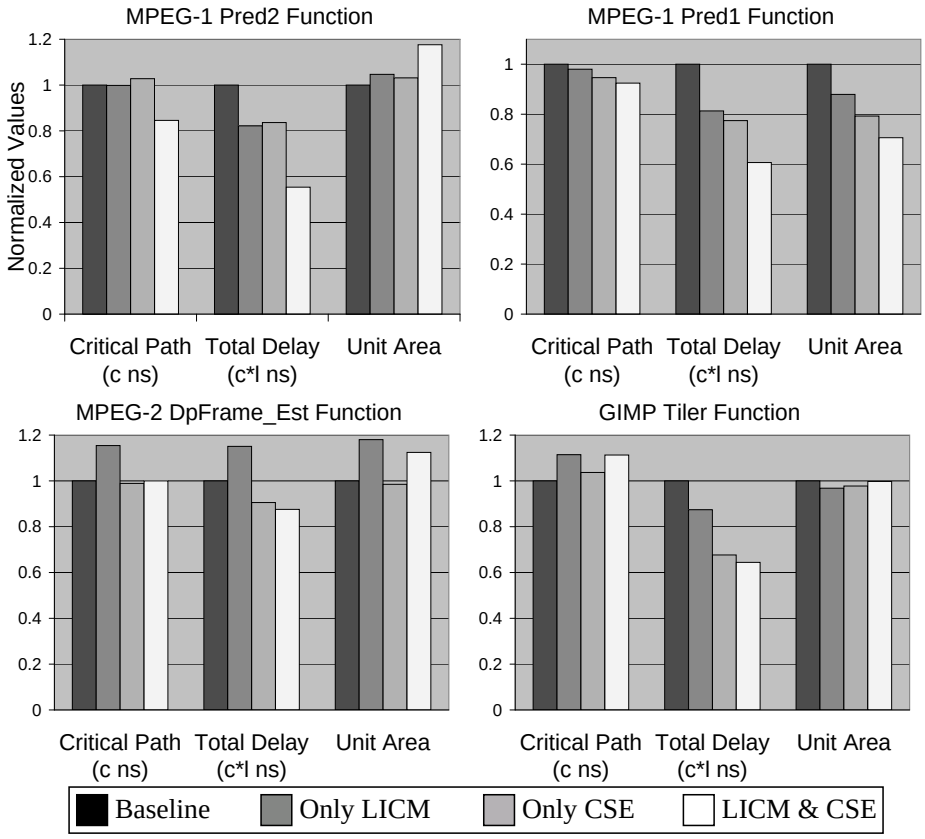

Figure 22. **Effects of the pre-synthesis transformations, loop-invariant code motion (LICM) and common sub-expression elimination (CSE), on logic synthesis results for the various designs**

section.

We see from the third row in Tables 4 and 5 that when CSE is applied in addition to the transformations in the baseline case, the number of states and the longest path cycles decrease significantly for all four designs; by more than 30 % for the *tiler* design. Clearly, there exist numerous opportunities to apply CSE in off-the-shelf code for these industrial applications. Also, as shown by the results in the last row of these tables, when both loop-invariant code motion and CSE are applied, the improvements in the cycles on the longest path are to some extent additive, especially for the MPEG-1 designs. The number of states reduces for the MPEG-1 designs and for *tiler* and remains constant for the MPEG-2 design. Performance increases for all the designs by between 12 % to 42 %.

9.3 Logic Synthesis Results for Pre-Synthesis Optimizations

We synthesized the VHDL generated by *Spark* corresponding to the pre-synthesis experiments using the Synopsys Design Compiler. The results for the critical path length, the total delay and the unit area (see Section 8) are presented in the graphs in Figure 22. 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.

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. However, LICM can lead to a higher area (for the *pred*2 and *dpframe_estimate* designs). This increase is less than 20 % and is mainly due to the larger FSM controller size. With LICM alone, the decreases in total delay through the circuit are up to 20 %. However, for the *dpframe_estimate* design, since LICM has little effect on the longest path cycles (see Table 5), its total delay increases with LICM alone. With CSE and LICM enabled, the total delay for all the designs decreases by between 17 to 45 %. Area decreases for the *pred*1 design, but increases by about 15 % for the *pred*2 and *dpframe_estimate* designs. Note that the area for the *tiler* design remains high due to the area-intensive resources used in this design, namely, a divider and a multiplier.

Loop-invariant code motion has two opposing effects on the synthesized designs. On the one hand, it reduces the cycles on the longest path through design by executing fewer operations within the loop body. On the other hand, LICM also leads to a bigger FSM controller. Also, because LICM increases resource utilization, the complexity of the steering logic (multiplexors and de-multiplexors) increases and hence, the area increases. However, the reduction in the size of the controller brought about by CSE overcomes the increases due to LICM. Also, since CSE eliminates redundant operations, the number of operations mapped to the functional units reduces, hence reducing area. For the *pred*2 and *dpframe_estimate* designs, the increase in area due to higher steering and control logic is larger than the area reduction due to the operations eliminated by CSE. As a matter of fact, as we will see in Section 12, when dynamic CSE is applied in conjunction with LICM, the area decreases for all the designs.

9.4 Results for Loop Unrolling

The *Spark* framework enables the designer to unroll loops in the source code by specifying the index variable of the loop to be unrolled and the number of times the loop should be unrolled. This is done by means of a script file that is read by *Spark* during initialization. We have used this feature to analyze the affects of loop unrolling on the functions from the MPEG-1 design. The results are presented in Table 6. The first row in this table lists the results for the baseline case with no loop unrolls. Since loop unrolling is applied to increase the scope for application of other optimizing transformations, we have enabled loop invariant code motion (LICM), common sub-expression elimination (CSE) and dynamic CSE (DCSE) for this baseline case.

The *pred*2 and the *pred*1 functions both have doubly nested loops. For our experiments, we choose to unroll one innermost loop from each design. Hence, Table 6 lists scheduling results for when the “p” loop from the *pred*2 function and the “j” loop from the *pred*1 function are unrolled and then the designs are scheduled. The number of unrolls for each design is specified in the first column of this table; it signifies the number of copies of the original loop body that is added to the loop. Hence, after one unroll, the new loop body of the loop has two loop bodies of the original loop.

Note that, when the upper iteration bound of the loop is not divisible by the number of unrolls, then the cycle

Transformation Applied	*MPEG*-1 *pred*2		*MPEG*-1 *pred*1	
	# States	# cycles	# States	# cycles
Baseline+LICM+CSE+DCSE	66	2126	36	835
1 Unroll	77(+16.7%)	2094(-1.5%)	47(+30.6%)	803(-3.8%)
3 Unrolls	99(+50%)	2078(-2.3%)	72(+91.7%)	787(-5.7%)

Table 6. **Results after unrolling the "p" loop in the** *pred*2 **function and the "j" loop in the** *pred*1 **function.**

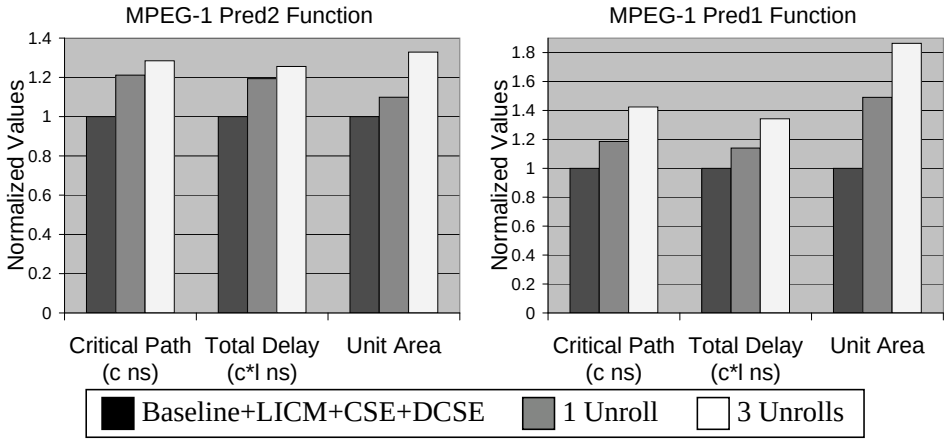

Figure 23. **Effects of loop unrolling on logic synthesis results for the MPEG-1** *Pred*2 **and** *Pred*1 **functions.**

count will not be accurate. This is because, say that the upper iteration bound of the loop is 10, then when the loop is unrolled twice, there are 3 original loop bodies in the new loop body. Hence, now the loop body will execute 4 times although a conditional check on the loop index variable count within the loop body will exit the loop early in the 4th iteration of the loop. However, *Spark* counts cycles for loops as the maximum number of cycles through the loop body multiplied by the number of loop body iterations. Hence, we only present results for 1 unroll and 3 unrolls in Table 6 and not for, say, 2 and 4 unrolls (the iteration upper bound for the unrolled loops for these designs is 8 iterations).

The results in Table 6 show that the number of states increase. This is because when the loop body is unrolled, there are more states required to execute the loop body. Also, there is a modest decrease in the number of cycles on the longest path for both the designs. This is because there are not enough resources allocated to take advantage of the increased opportunities available for operation parallelism. This is validated by the logic synthesis results presented in Figure 23. The graphs in this figure show the results for the baseline case (1st bar), with one loop unroll (2nd bar) and with three loop unrolls (3rd bar). The results are normalized with respect to the baseline case values.

From these graphs we see that as the amount of unrolling is increased, both the critical path in the design and the design area increase significantly. For example, in the case of the *pred*1 design with 3 loop unrolls, critical path length increases by over 40 % and the area almost doubles. These increases are because the number of operations mapped to the same resources increase as the loop are unrolled. This increased resource utilization comes at the

Transformation Applied	*MPEG*-1 *pred*2		*MPEG*-1 *pred*1	
	# States	# cycles	# States	# cycles
Baseline+LICM+CSE+DCSE	65	2062	36	835
1 Unroll	71(+9.2%)	1870(-9.3%)	42(+16.7%)	547(-34.5%)
3 Unrolls	84(+29.2%)	1790(-13.2%)	55(+52.8%)	435(-47.9%)

Table 7. **Loop unrolling results using a resource allocation of 4 adders instead of 2 for the** *pred*2 **and the** *pred*1 **functions.**

price of increasing costs of steering logic such as multiplexors and de-multiplexors and the control logic associated with them. The increase in the controller size also contributes to the increase in area.

The logic synthesis results in the graphs in Figure 23 show that the increases in critical path length outweigh the reductions in the number of cycles on the longest path through the design. Effectively, in these experiments, loop unrolling leads to worse total delays through the circuit and larger design area. However, we can do better with loop unrolling, if we increase the resources allocated to these designs, as explained in the next section.

9.5 Loop Unrolling with Increased Resource Allocations

For the experiments performed for this paper, we chose a minimal resource allocation. This is to mimic real-life situations where design area (which is correlated with resource allocation) is often severely constrained. However, loop unrolling is a technique that increases the opportunities for design parallelization. Hence, in the context of high-level synthesis, loop unrolling should be applied when there is a low resource utilization, i.e., when the resources are idle in several cycles. Clearly, our approach of allocating a small number of resources does not leave many idle resources after applying the current set of parallelization techniques. In this section, we demonstrate that loop unrolling can lead to the improved synthesis results when a higher resource allocation is available.

For both the MPEG-1 designs, we have found that the ALU is the critical component, since increasing this component leads to most improvements in the results (with and without loop unrolling). Hence, we increased the number of adders from two to four and ran our experiments with loop unrolling again. The scheduling results are presented in Table 7. The metrics presented and loops unrolled are the same as before.

With a resource allocation of 4 adders, the improvements in the number of cycles for the *pred*2 design are 9.3 % and 13.2 % for 1 and 3 loop unrolls respectively. Similarly, the *pred*1 function shows reductions of almost 48 % in cycles when the “j” loop is unrolled 3 times. Also, the increase in the number of states for the both the designs is almost half as compared to the results with 2 adders. This is because the scheduler is able to schedule the loop body in much fewer cycles with the increased resource allocation.

The logic synthesis results for loop unrolling with the higher resource allocation of 4 adders are given in Figure 24. From these results, we can see that using the higher resource allocation, loop unrolling can achieve lower

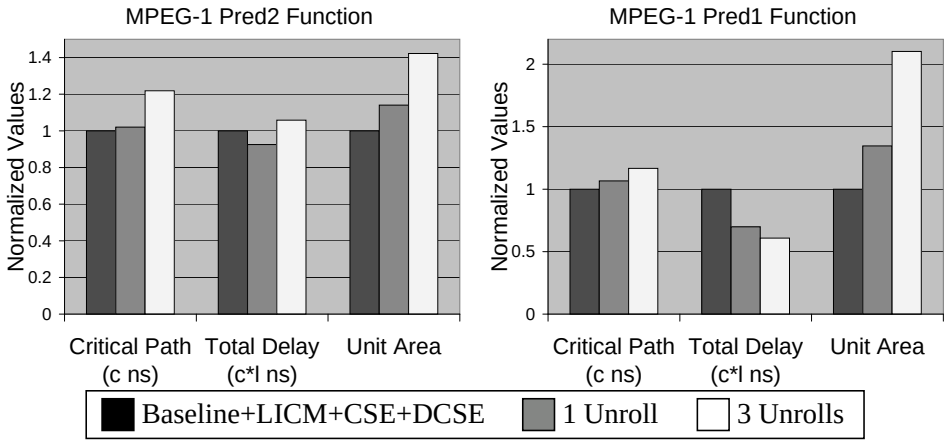

Figure 24. **Loop unrolling with a resource allocation of 4 Adders and 2 Shifters for the MPEG-1** *Pred*2 **and** *Pred*1 **functions**

delays through the circuit. The increase in critical path length is much lower and both the *pred*2 and *pred*1 designs achieve a lower total delay for 1 loop unroll over the baseline case. Clearly, a resource allocation of 4 adders enables the parallelizing optimizations to exploit the increased opportunities available after 1 loop unroll.

However, this resource allocation is not enough for handling the interconnect and controller overhead when the loops are unrolled 3 times. Although longest delay through the *pred*1 design is lower by almost 45 % for 3 loop unrolls, the critical path length for the *pred*2 design increases with 3 loop unrolls and hence, the total delay through the *pred*2 design is worse. The area also increases significantly for both the designs with 3 loop unrolls; it more than doubles for *pred*1 design.

From these experiments, it is evident that the area-performance trade-offs of loop unrolling are complex. On the one hand, loop unrolling provides code optimizations more freedom to parallelize the design and on the other hand, the controller complexity and steering and associated control logic increases dramatically. When a higher resource allocation is available to schedule the design, loop unrolling can be a useful transformation. Future work can include developing heuristics to guide this transformation and/or aid the designer in making decisions about which loops to unroll and to what extent.

10 Results for Dynamic CSE

10.1 Scheduling Results for Dynamic CSE

Next, we compare the effectiveness of the dynamic CSE transformation applied during scheduling with that of a traditional CSE pass applied before scheduling. The synthesis results for these experiments are presented in Tables 8 and 9 for the four designs. The first row in these tables lists results for the baseline case with all code motions enabled along with copy propagation and dead code elimination. The second row is for when only CSE is applied as a pass before scheduling, the 3rd row for when only dynamic CSE is applied during scheduling and finally, the 4th row presents results for when both CSE and dynamic CSE are applied. In all these experiments, dynamic copy propagation is done whenever possible (even when dynamic CSE is not applied). The percentage

Transformation Applied	*MPEG*-1 *pred*2			*MPEG*-1 *pred*1		
	# States	Long Path	# Regs	# States	Long Path	# Regs
Baseline	85	4123	31	43	1761	22
with CSE	73(-14.1%)	3355(-18.6%)	24(-22.6%)	38(-11.6%)	1441(-18.2%)	19(-13.6%)
with Dyn CSE	64(-24.7%)	2779(-32.6%)	20(-35.5%)	33(-23.3%)	1121(-36.3%)	12(-45.5%)
with CSE & Dyn CSE	63(-25.9%)	2715(-34.1%)	21(-32.3%)	32(-25.6%)	1057(-40%)	13(-40.9%)

Table 8. **Scheduling results after applying CSE and Dynamic CSE for MPEG-1 designs**

Transformation Applied	*MPEG*-2 $dpframe\_estimate$			*GIMP tiler*		
	# States	Long Path	# Regs	# States	Long Path	# Regs
Baseline	56	684	42	43	4031	27
with CSE	56(0%)	626(-8.5%)	40(-4.8%)	29(-32.6%)	2631(-34.7%)	18(-33.3%)
with Dyn CSE	49(-12.5%)	598(-12.6%)	31(-26.2%)	28(-34.9%)	2531(-37.2%)	17(-37%)
with CSE & Dyn CSE	49(-12.5%)	598(-12.6%)	33(-21.4%)	28(-34.9%)	2531(-37.2%)	16(-40.7%)

Table 9. **Scheduling results after applying CSE and Dynamic CSE for designs from MPEG-2 and the GIMP image processing tool**

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 [8].

The results in these tables demonstrate that applying CSE alone can lead to improvements up to 32 % in the number of states (for *tiler*) and between 8 to 34 % in the longest path cycles. In itself, these improvements are significant. Note that several of these functions are called multiple times from within loops and hence, the improvements multiply by the number of iterations of the loops.

When dynamic CSE is applied, the improvements are even more dramatic for all the designs as is evident by the results in the third row of Tables 8 and 9. Clearly, dynamic CSE is able to eliminate many more operations with common sub-expressions than traditional CSE can. Employing dynamic CSE during scheduling can at times improve schedule lengths by 18 % over applying CSE as a pass before scheduling. The last row in these tables show that applying both CSE and dynamic CSE together leads to further improvements in the synthesis metrics for the MPEG-1 designs.

Also, our experiments show another important result; contrary to common belief, the results show that applying CSE and dynamic CSE leads to a *reduction* in the number of registers required. This decrease can be attributed to three inter-related 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 sub-expression are eliminated,

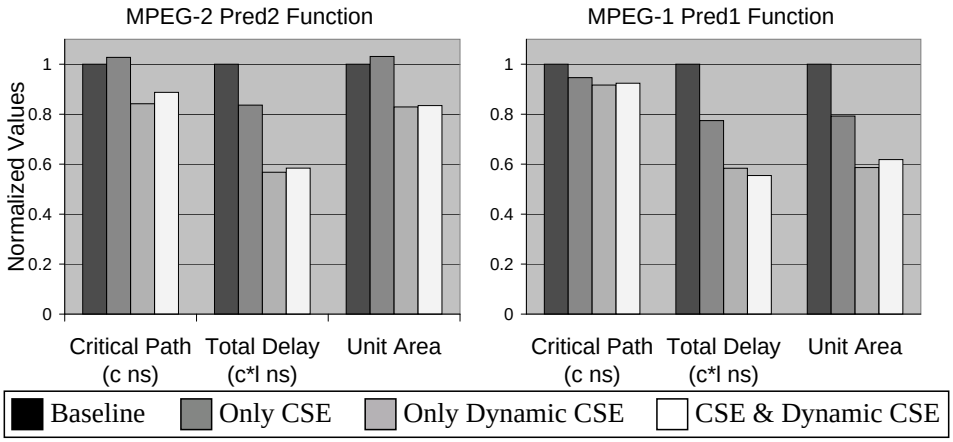

Figure 25. **Effects of CSE and dynamic CSE on logic synthesis results for the MPEG-1** *Pred*2 **and** *Pred*1 **designs**

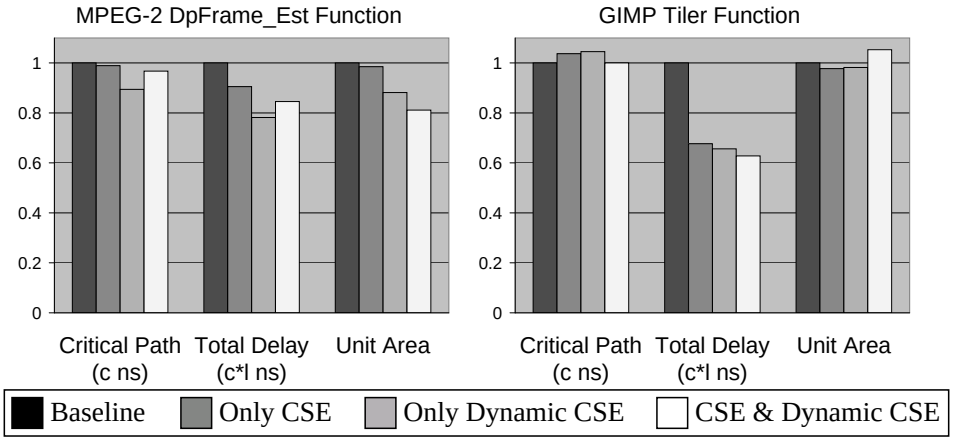

Figure 26. **Effects of CSE and dynamic CSE on logic synthesis results for the MPEG-2** $dpframe\_estimate$ **and the GIMP** *tiler* **functions**

then they can reuse the result of only one of the operations. This saves on storing the results of several operations.

10.2 Logic Synthesis Results for Dynamic CSE

Once again, we synthesized the VHDL corresponding to the experiments presented in the last section using the Synopsys logic synthesis tool. The logic synthesis results are presented in the graphs in Figures 25 and 26. The values of each metric are mapped as before: for when all the code motions are enabled but no CSE or dynamic CSE is applied (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 20 % (for $dpframe\_estimate$) to 40 % (for *pred*1). Also, dynamic CSE can lead to lower area; sometimes up to 40 % less (for *pred*1). This decrease in area can be attributed to two factors. Firstly, the elimination of some operations due to CSE and dynamic CSE means that fewer operations are mapped

Transf.	*MPEG*-1 *pred*2		*MPEG*-1 *pred*1		*MPEG*-2 *dpframe*		*GIMP tiler*	
Applied	# States	# cycles	# States	# cycles	# States	# cycles	# States	# cycles
Baseline	85	4123	43	1761	56	684	33	3031
+Chaining	81(-4.7%)	4095(-.7%)	40(-7%)	1749(-.7%)	56(-0%)	646(-5.6%)	32(-3%)	3021(-.3%)

Table 10. **Scheduling results after chaining operations across conditionals for all the four designs**

to the functional units and this leads to reduced interconnect (multiplexors and demultiplexors). Secondly, the reductions in the controller size and the number of registers required lead to further reductions in the area.

The overall results in the graphs in Figures 25 and 26 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.

11 Results for Chaining Across Conditionals

We developed the chaining across conditionals transformation primarily for the synthesis of microprocessor functional blocks [58]. However, even in the domain of the multimedia and image processing applications considered in this paper, we find that there exist several opportunities to chain simple assign (copy) operations that occur in conditional blocks with operations that produce their values. This sometimes can generate a result one cycle earlier than it would otherwise would have been available (see Section 6.3).

In Table 10 we compare the scheduling results for the baseline case of the four designs (1st row) with the results for when chaining across conditionals is enabled (2nd row). The improvements in number of states due to chaining range between 0 % to 7 % and in the cycles on the longest path range between 0.3 % to 5.6 %. Clearly, these improvements are marginal.

The logic synthesis metrics corresponding to these scheduling results are presented in the graphs in Figure 27. The first bar is the baseline case and the second bar is with chaining across conditionals. Chaining operations across conditionals leads to almost constant critical path lengths. This coupled with the modest improvements in longest path cycles translates to almost constant total delays through the circuit.

The design area decreases only for the *pred*1, but increases for the other three designs. This increase can be attributed to the fact that as more operations (even variable copy operations) are packed into a cycle, the multiplexing costs increase. The decreases in the controller size (if any) work to counterbalance this increase in area to some extent.

Clearly the gains from chaining across conditionals for the applications considered in this paper are minimal if any. However, this does not diminish the value of this technique; it is indispensable for the synthesis of microprocessor functional blocks.

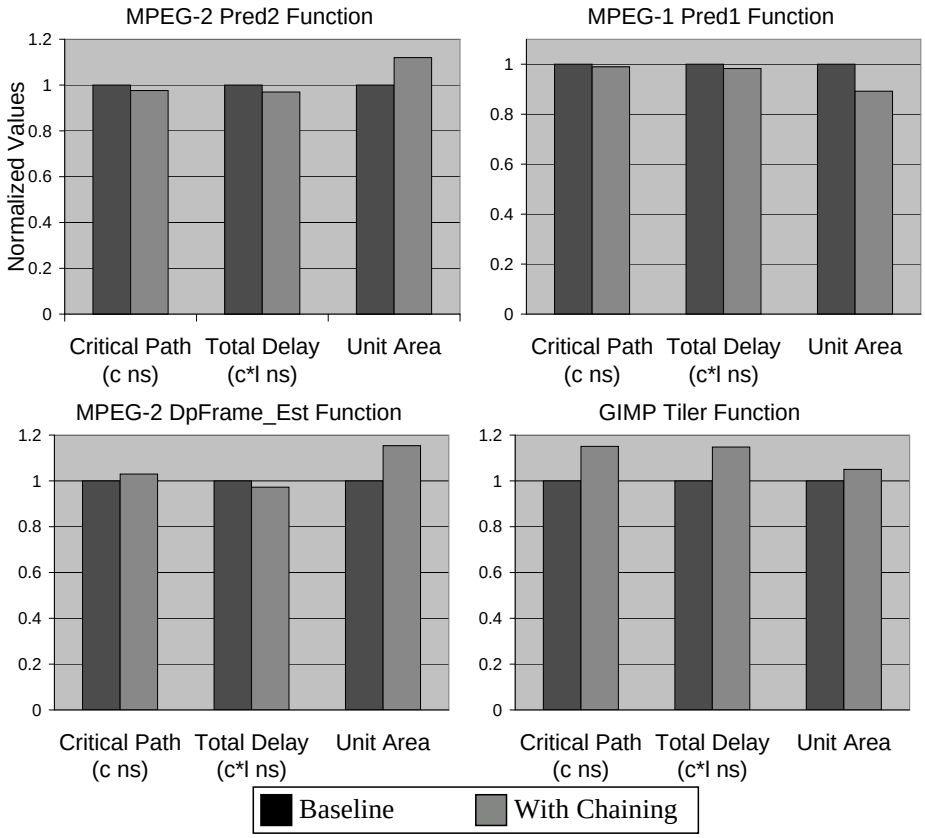

Figure 27. **Effects of chaining across conditionals on the logic synthesis results for the MPEG-1** *Pred*2 **and** *Pred*1 **functions and the MPEG-2** $dpframe\_estimate$ **and the GIMP** *tiler* **functions**

12 Putting it all together

In the last 3 sections, we have analyzed the scheduling and logic synthesis results of the various transformations presented in this paper. From this analysis, we conclude that the transformations that lead to the most improvements in the quality of synthesis results are: loop invariant code motion (LICM), common sub-expression elimination (CSE) and dynamic CSE. Let us now examine how these techniques perform when applied together.

Tables 11 and 12 list the results for applying LICM along with dynamic CSE and CSE. The first row is the baseline case with only speculative code motions applied, the second row has LICM and dynamic CSE applied and the last row has LICM, CSE and dynamic CSE applied. When dynamic CSE is enabled along with LICM, the cycles on the longest path decrease by 16 % for the MPEG-2 $dpframe\_estimate$ design and by 44 to 52 % for the other three designs. The reductions in the number of states in the controller are between 12 to 34 %. Furthermore, applying CSE gives no further improvements over applying only dynamic CSE.

The logic synthesis results corresponding to these experiments are presented in Figure 28. The first bar corresponds to the baseline case, the second bar to LICM and dynamic CSE applied and the last bar has LICM, CSE and dynamic CSE applied. The improvements in the cycles on the longest path more or less translate over to the longest delay through the circuit; this reduces by 20 to 60 %. Also, the area of the design decreases by 5 to 40 % when these transformations are applied. It is important to note that these improvements are obtained over designs

Transformation Applied	*MPEG*-1 *pred*2		*MPEG*-1 *pred*1	
	# States	Long Path	# States	Long Path
Baseline	85	4123	43	1761
with LICM+DCSE	66(-22.4%)	2126(-48.4%)	36(-16.3%)	835(-52.6%)
with LICM+CSE+DCSE	66(-22.4%)	2126(-48.4%)	36(-16.3%)	835(-52.6%)

Table 11. **Results after applying loop-invariant code motion, dynamic CSE and CSE on the MPEG-1 designs.**

Transformation Applied	*MPEG*-2 $dpframe\_est$		*GIMP* $tiler$	
	# States	Long Path	# States	Long Path
Baseline	56	684	43	4031
with LICM+DCSE	49(-12.5%)	571(-16.5%)	28(-34.9%)	2234(-44.6%)
with LICM+CSE+DCSE	49(-12.5%)	571(-16.5%)	28(-34.9%)	2234(-44.6%)

Table 12. **Results after applying LICM, dynamic CSE and CSE for the** $dpframe\_estimate$ **and the** $tiler$ **designs**

already optimized by the speculative code motions [8]. Also, as stated earlier, 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 note that when an optimizing transformation is applied, there are two conflicting factors that come into play. As the resource utilization increases, the steering logic (multiplexors and demultiplexors) 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 multiplexors, functional units and demultiplexors, 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, the fact that the critical path length remains fairly constant as these optimizing transformations are applied is an important result because the critical path length dictates the minimum clock period for the design.

13 Conclusions and Future Work

We have proposed a methodology for high-level synthesis that first applies coarse-grain and fine-grain source level transformations during a pre-synthesis phase. This pre-synthesis 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. Also, we have proposed an enhancement of operation chaining that chains operations across conditional boundaries. This transformation is motivated

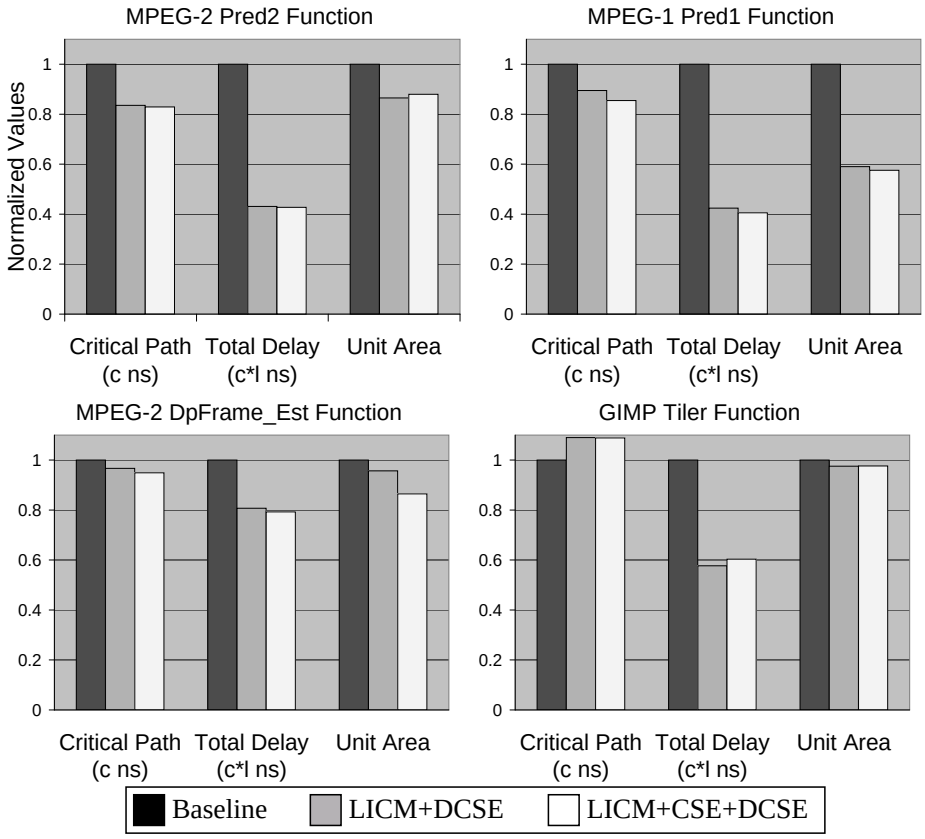

Figure 28. **Final logic synthesis results after applying loop-invariant code motion (LICM), CSE and dynamic CSE to the MPEG-1, MPEG-2 and GIMP designs**

by the control-intensive nature of the applications targeted by our methodology.

We have 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 perform an analysis of the effects of the various transformations on the scheduling *and* logic synthesis 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 the various transformations like 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 40 % in the design area. Furthermore, these improvements are over a design that has already been optimized by the speculative code motions. In this paper, we also explored the effects of loop unrolling on synthesis results. In future work, we plan to expand this work and develop a comprehensive strategy for the application of loop transformations both during the pre-synthesis phase and during scheduling.

Acknowledgments

This project is funded by the Semiconductor Research Corporation (SRC) under Task I.D. 781.001. We would like to thank Mehrdad Reshadi and Nick Savoiu for their contribution to the *Spark* framework.

References

[1] K. Wakabayashi. C-based synthesis experiences with a behavior synthesizer, "Cyber". In *Design, Automation and Test in Europe*, 1999.

[2] Get2Chip Incorporated. Volare multi-level synthesis. http://www.get2chip.com.

[3] L.C.V. dos Santos. *Exploiting instruction-level parallelism: a constructive approach*. PhD thesis, Eindhoven University of Technology, 1998.

[4] S. Haynal. *Automata-Based Symbolic Scheduling*. PhD thesis, University of California, Santa Barbara, 2000.

[5] G. Lakshminarayana, A. Raghunathan, and N.K. Jha. Wavesched: a novel scheduling technique for control-flow intensive designs. *IEEE Transactions on CAD*, May 1999.

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

[7] A. Nicolau. A development environment for scientific parallel programs. Technical Report TR 86-722, Department of Computer Science, Cornell University, 1985.

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

[9] D. D. Gajski. *Silicon Compilation*. Addison-Wesley, 1988.

[10] D. D. Gajski, N. D. Dutt, A. C-H. Wu, and S. Y-L. Lin. *High-Level Synthesis: Introduction to Chip and System Design*. Kluwer Academic, 1992.

[11] R. Camposano and W. Wolf. *High Level VLSI Synthesis*. Kluwer Academic, 1991.

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

[13] M. Potkonjak and J. Rabaey. Optimizing resource utlization using tranformations. *IEEE Trans. on CAD*, March 1994.

[14] R. Walker and D. Thomas. Behavioral transformation for algorithmic level IC design. *IEEE Trans. on CAD*, Oct. 1989.

[15] R. Camposano. Path–based scheduling for synthesis. *IEEE Transactions on Computer–Aided Design*, Jan. 1991.

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

[17] T. Kim, N. Yonezawa, J.W.S. Liu, and C.L. Liu. A scheduling algorithm for conditional resource sharing - a hierarchical reduction approach. *IEEE Transactions on CAD*, April 1994.

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

[19] R. A. Bergamaschi, S. Raje, and L. Trevillyan. Control-flow versus data-flow-based scheduling: combining both approaches in an adaptive scheduling system. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, March 1997.

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

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

[22] M. Rim, Y. Fann, and R. Jain. Global scheduling with code-motions for high-level synthesis applications. *IEEE Transactions on VLSI Systems*, September 1995.

[23] A.A. Kountouris and C. Wolinski. Efficient scheduling of conditional behaviors for high-level synthesis. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, July 2002.

[24] R.K. Gupta J. Li. Hdl optimizations using timed decision tables. In *Design Automation Conference*, 1996.

[25] J.M. Mendas O. Pealba and R. Hermida. Maximizing conditional reuse by pre-synthesis transformations. In *Design, Automation and Test in Europe*, 2002.

[26] R.K. Gupta J. Li. Decomposition of timed decision tables and its use in presynthesis optimizations. In *International Conference on Computer Aided Design*, 1997.

[27] C. Wolinski A. Kountouris. High level pre-synthesis optimization steps using hierarchical conditional dependency graphs. In *Euromicro Confernce*, 1999.

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

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

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

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

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

[33] M.Miranda, F.Catthoor, M. Janssen, and H.De Man. High-level address optimisation and synthesis techniques for data-transfer intensive applications. *IEEE Transactions on VLSI Systems*, December 1998.

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

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

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

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

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

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

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

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

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

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

[44] S. Gupta, M. Reshadi, N. Savoiu, N.D. Dutt, R.K. Gupta, and A. Nicolau. Dynamic common sub-expression elimination during scheduling in high-level synthesis. In *International Symposium on System Synthesis*, 2002.

[45] S. Gupta, N. Savoiu, N.D. Dutt, R.K. Gupta, and A. Nicolau. Using global code motions to improve the quality of results for high-level synthesis. Technical Report CECS-TR-02-29, Center for Embedded Computer Systems, Univ. of California, Irvine, 2002.

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

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

[48] A. Orailoglu and D.D. Gajski. Flow graph representation. In *Design Automation Conference*, 1986.

[49] M. C. McFarland. The value trace: A data base for automated digital design. Technical Report DRC-01-4-80, Carnegie-Mellon University, Design Research Center, 1978.

[50] R.K. Brayton, R. Camposano, G. De Micheli, R.H.J.M. Otten, and J. van Eijndhoven. *The Yorktown Silicon Compiler System*, chapter in Silicon Compilation. Addison-Wesley, 1988.

[51] V. Chaiyakul, D.D. Gajski, and L. Ramachandran. Minimizing syntactic variance with assignment decision diagrams. Technical Report ICS-TR-92-34, UC Irvine, 1992.

[52] A.A. Kountouris and C. Wolinski. Hierarchical conditional dependency graphs as a unifying design representation in the codesis high-level synthesis system. In *International Symposium on System Synthesis*, 2000.

[53] R.A. Bergamaschi. Behavioral network graph unifying the domains of high-level and logic synthesis. In *Design Automation Conference*, 1999.

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

[55] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4):345–420, 1994.

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

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

[58] S. Gupta, T. Kam, M. Kishinevsky, S. Rotem, N. Savoiu, N.D. Dutt, R.K. Gupta, and A. Nicolau. Coordinated transformations for high-level synthesis of high performance microprocessor blocks. In *Design Automation Conference*, 2002.

[59] S. Gupta, N.D. Dutt, R.K. Gupta, and A. Nicolau. Dynamic conditional branch balancing during the high-level synthesis of control-intensive designs. In *To appear in the Design, Automation and Test Conference*, 2003.

[60] A. Nicolau. Uniform parallelism exploitation in ordinary programs. In *International Conf. on Parallel Processing*, 1985.

[61] S. Novack and A. Nicolau. A hierarchical approach to instruction-level parallelization. *International Journal of Parallel Programming*, 1(23), 1995.

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

[63] C. Lee, M. Potkonjak, and W. H. M.-Smith. UCLA Mediabench benchmark suite. http://www.cs.ucla.edu/∼leec/mediabench/.

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