Politecnico di Milano

FACOLTÀ DI INGEGNERIA
Corso di Laurea in Computer Science and Engineering

Tesi di laurea magistrale

Balancing Code Motion for FPGA Resource Reduction

Candidato:
Biagio Festa
Matricola 841988

Relatore:
Ch.mo Prof. Fabrizio Ferrandi

Anno Accademico 2016-2017

A miei genitori,
e mia sorella Maria.

Abstract

Versione Italiana

La *sintesi ad alto livello* (HLS) è quel processo che si occupa della *traduzione* da una descrizione comportamentale algoritmica in un codice che può essere direttamente implementato su *hardware*.

Solitamente il processo di sintesi parte dalla descrizione formale di un algoritmo attraverso un linguaggio di *alto livello* (come il *C* o il *C++*) per poi essere tradotto in qualcosa che meglio si adatta alla descrizione di un circuito, come ad esempio, i vari linguaggi di *HDL* (Hardware Description Languages).

La recente diffusione di tecnologie hardware, in particolare *FPGA* (Field-Programmable Gate Array), ha messo in luce l'importanza e il bisogno di efficaci metodologie nel processo di sintesi. Il risultato prodotto da tale processo, infatti, ha considerevole impatto nel determinare le performance del circuito finale, in termini di latenza e area.

Il presente lavoro di tesi si focalizza sull'analisi della *rappresentazione intermedia* del codice di input al fine di incrementare ed ottimizzare l'*area* complessiva del circuito mediante la *condivisione* delle *unità funzionali*.

La metodologia presentata parte dall'analisi dei *blocchi basici* all'interno del *flusso di controllo del programma*. Attraverso l'impiego di strategie *euristiche*, successivamente, vengono sfruttati approcci di *spostamento delle istruzioni* al fine di bilanciare la distribuzione di un tipo di operazioni all'interno del flusso. In questo modo, si favorisce, dunque, la condivisione (riuso) di uno stesso modulo funzionale per differenti operazioni compatibili.

English Version

HLS (High-Level Synthesis) is a *translation* process from a behavioural description of an algorithm into a code which can directly be implemented on *hardware*.

Usually, the synthesis process starts from the formal description of an algorithm through a high-level language (such as *C* or *C ++*) and then it translates that into something which best suits the circuit description, e.g., *HDL* (Hardware Description Languages).

The recent diffusion of hardware technologies, in particular *FPGA* (Field-Programmable Gate Array), has highlighted the importance of effective methodologies during the synthesis process. Indeed, the synthesis result can have a significant impact in determining the performance of the final circuit, in terms of *latency* and *area*.

This thesis work focuses on the analysis of the *intermediate representation* of the input code in order to increase and optimize the overall area of the circuit by *functional units sharing*.

Therefore, the proposed methodology starts from the analysis of the *basic blocks* in the *program control flow*. Through the use of *heuristic* strategies, the statements are moved (*code motion*) in order to facilitate the sharing (reuse) of the same functional module for different operations of the same type.

Ringraziamenti

Desidero innanzitutto ringraziare i miei *genitori* per l'*enorme* sostegno che mi ha accompagnato durante questi anni di studio.

Ringrazio i miei *amici*, colleghi di università e non, con cui ho condiviso gioie e dolori.

Infine, un ringraziamento va al *Professor* Fabrizio Ferrandi e all'*ing.* Marco Lattuada per i preziosi suggerimenti che mi hanno permesso di realizzare questo lavoro.

Contents

Chapter 1

Introduction

In the recent years, the market around hardware solutions is considerately grown because of the optimizations and possibilities which those kinds of implementations can achieve. The intrinsic hardware parallelism easily allows accelerating computation process. Another important aspect of hardware solutions regards the possibility to obtain a dedicated solution which can be efficient in terms of power consumption per area.

In this regard, the FPGAs market has considerably grown because of its simplicity in the design process. Indeed, in these years, several synthesis tools have been developed and improved which allows implementing a functionality (algorithm) starting from a common high-level programming language and guiding the designer following his/her requirements.

It is clear, therefore, that an effective high-level synthesis process becomes the key in the FPGAs market.

In most recent studies, a particular attention has been given to the results produced by the HLS in terms of the occupied area on the final circuit. Indeed, the area performances aspect can imply different considerations on the overall final product. For example, reducing area can bring to less power consumption or the possibility to implement multiple functionalities on the same board.

Often, in HLS, one of the most common approaches is to maximize and exploit the hardware parallelism as much as possible. Generally, that kind of approach is entirely in opposition to the resource saving policies. Indeed, we can usually refer that problem as a trade-off between latency and area performance.

Another important aspect is about the code motion optimizations and code speculations. Those kinds of techniques also rearrange the statements in the code in order to try to maximize the number of operation concurrently executed.

The methodology proposed in this work, in fact, starts from those code motion steps. As we are going to see in successive chapters, the common code motion strategies and algorithms can be really unbalanced in the sense that

there are no any considerations about the resources usage. Indeed, the primary goal in speculative approaches is to reduce the overall latency of the final circuit.

As we are going to see, the proposed work in this paper aims to be a successive step, built on top of speculations, which tries to apply additional code motion increasing the possibility of resource sharing with an approach as more conservative as possible.

The main contributions proposed in this thesis are:

- Optimize the critical operations distribution in the intermediate code representation by means of heuristic strategies and code motion.
- Balancing code motion based on clustering operations based on their type and "*level of criticality*". This allows reducing the number of needed resources in order to execute the application without impacting on latency performances creating new critical paths.
- Dependencies analysis among operations and code optimizations in order to reduce conflicts among variables. It allows generation of resource sharing without additional *multiplexers*.

In conclusion, the objective is to facilitate the resource sharing without too impacting on the latency performance.

The rest of this thesis is organized as follow. In the next chapter (number 2) we are going to illustrate the theoretical foundations which are considered relevant for a proper understanding of the present work. In chapter 3 we are going to explore the State of the Art of resource sharing and scheduling algorithms which have inspired this thesis work. Chapter 4 will show the proposed methodology, problems formalizations and heuristic approaches developed. In chapter 5 we are going to see how the proposed methodology, implemented in the Panda framework, can effectively increase the resources sharing in some cases. Finally, chapter 6 will briefly show some conclusive considerations and some further suggestions which can improve the methodology itself inspiring some future developments.

Chapter 2

Glossary

In this chapter, some theoretical and preliminary concepts will be illustrated in order to clarify the comprehension of the topics discussed in this thesis.

The glossary section has been divided into three different major sections.

The first section presents an introduction to the graph theory. Indeed, several aspects of the problem will be modelled by means of the usage of graphs.

Successively the second section will discuss some definitions in the compilers context. Since the HLS can be seen as a compilation process, different concepts will be widely taken up.

Finally, the last section will briefly analyse some of the principal aspects in Hardware Synthesis which are fundamental in this thesis work (such as register and module allocation).

2.1 Graph Theory

2.1.1 Graph Definitions

A graph is an ordered pair $G = (V, E)$ where V is a set of *vertices* or *nodes* and E is a set of *edges* or *arcs*. An edge is a 2-element subset of V, that is, $(v_i, v_j) \in E$, where $v_i, v_j \in V$.

Moreover, a graph can be either *directed* or *undirected*.

In a *directed graph* each edge has a direction. Generally, the edge (v_i, v_j) refers to an arc which goes *from* v_i *to* v_j (figure 2.1a).

On the contrary, *undirected graph*'s edges have no orientation. Therefore the edges (v_i, v_j) and (v_j, v_i) are the same (figure 2.1b).

(a) Directed graph. (b) Undirected graph.

Figure 2.1: A very simple graph with two vertices and only one edge.

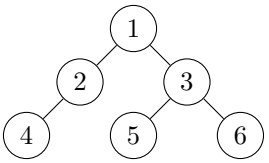


Figure 2.2: An example of tree graph.

2.1.2 Tree Definition

A tree is an *undirected graph* in which two vertices are linked by *exactly one* path.

Often it refers a tree as a *rooted tree*. In a rooted tree, a vertex is designed as a *root*. Conceptually the root node represents the starting node from which every path should begin.

Starting from this point of view, it is possible to make no difference between a tree with directed edges and the same one with undirected edges. Sometimes it may be useful to draw a tree with directed edges just to make the root node and paths toward leaves more explicit.

An example of a tree is shown in figure 2.2 where the node 1 is the root of the tree.

2.1.3 Dominator

Let $G = (V, E)$ be a *directed graph* and $s_0 \in V$ a defined vertex which we indicate as *source node* of the graph.

We can formally define a *dominance relation* among graph's vertices:

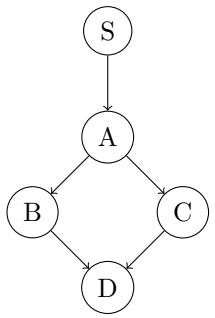
$$D \subseteq V \times V.$$

Let u and v be two vertices in the graph G, then u is said to dominate v (w.r.t source vertex s_0) if all paths from s_0 to v in the graph must pass through vertex u.

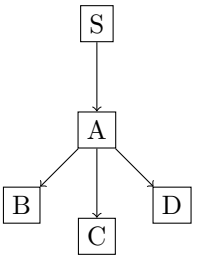
Relation properties: For completeness, we can list the properties in this relation:

- *Reflexivity*:

$$\forall v \in V \quad \big((v, v) \in D \big).$$



(a) A directed graph.



(b) The respective dominator tree (vertex "S" as source).

Figure 2.3: An example of dominance relation.

- *Asymmetry*:

$$\forall v,\, u \in V \quad \big((v,u) \in D \implies (v,u) \notin D \big).$$

- *Transitivity:*

$$\forall v,\, u,\, z \in V \quad \big((v,u) \in D \land (u,z) \in D \implies (v,z) \in D \big)$$

We can deduce that the set of vertices V and the binary relation of dominance D is a *POSET* (Partially Ordered Set). Indeed, not every pair of vertices needs to be comparable.

Be aware that it always is possible to define a *strict* relationship of dominance, the only difference is the *reflexivity* property. In the strict form a vertex does not dominate itself (*irreflexive* property) and then the POSET becomes *weak*.

Despite that difference, two concepts can be easily interchanged. So during later sections, there will not be any specification about the strict or not-strict property.

By reference to Figure 2.3a, let us suppose to select the vertex S as the source node. For instance, it is easy to see the vertex A dominates the vertex B. In the example, each path from S to B must pass through A.

Immediate Dominators: A vertex u is an *immediate dominator* of a *different* other vertex v ($u \neq v$) if u is a dominator of v but it does not dominate any other node that dominates v.

Every vertex in the graph, except the source node, has a *unique* immediate dominator.

In the figure 2.3a, the vertex A is an immediate dominator of the vertex B. Indeed, A, of course, is a dominator of B and it does not dominate S which is the only other vertex which dominates B.

Instead, the vertex S is not an immediate dominator of B because it dominates A which is a dominator of B, as already seen.

Dominators Tree: A very useful concept is the *dominators tree*. That data structure is widely used in compilers scenarios (as we are going to see later).

The dominator tree is built starting from the root selected as the source node of the dominance relation.

Each edge in the tree which connects parent and child node is related with the *immediate* dominance relationship. Indeed, a node's parent in the tree is the immediate dominator in the relation.

We can see an example of dominator tree in the figure 2.3b. The tree is built starting from the directed graph in the figure 2.3a and picking the vertex S as the source node.

Note that the figure 2.3b uses directed edges just to be more clear about the direction from the root node to leaves.

2.1.4 Post-Dominator

The *post-dominance* definition is analogous what we have just seen for dominator in section 2.1.3.

Let $G = (V, E)$ be a *directed graph* and $s_{exit} \in V$ a defined vertex which we indicate as *exit node* of the graph.

We can formally define a *post-dominance relation* among graph's vertices:

$$PD \subseteq V \times V.$$

In particular, let u and v be two vertices in the graph G, then u is said to post-dominate v (w.r.t source vertex s_{exit}) if all paths from v to s_{exit} in the graph must pass through vertex u.

All properties we have seen in dominator relation are still valid in post-dominator relation.

Regard the figure 2.3a, for example, we can suppose to select the vertex D as exit node of the graph. Therefore it is easy to see the vertex A post-dominates the vertex S. Indeed each path from S to D must pass through A.

The same principles about *immediate post-dominator* and *post-dominator tree* can be applied as seen for dominator definitions.

Figure 2.4 shows an example of *post-dominator* tree respect to the directed graph in figure 2.3a. The vertex D is selected as the *exit node*.

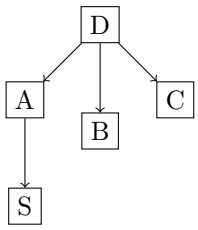


Figure 2.4: The post-dominator tree starting from the directed graph in figure 2.3a and selecting vertex D as the *exit node*.

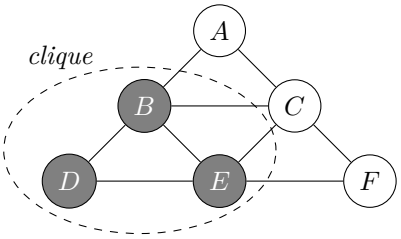


Figure 2.5: An example of a clique on the graph. The vertices in grey compose the clique.

2.1.5 Clique Cover

Clique Definition

In an *undirected graph* $G = (V, E)$ we can define a *clique* C as a subset of vertices (i.e., $C \subseteq V$) where every two distinct vertices are connected by an edge (i.e., *adjacent*).

Figure 2.5 shows an example of a clique. Indeed we have:

$$C = \{B, D, E\} \subseteq V.$$

As we can notice, each pair of vertices C is connected among them.

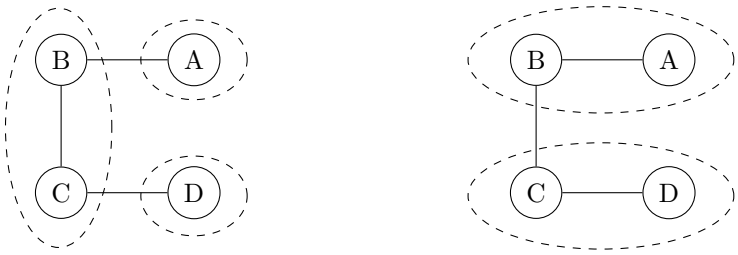
Note that multiple different cliques can be found on the same graph.

Clique Cover Problem

The *clique cover* problem consists of decomposing into cliques a given undirected graph. Therefore those clique partitions are disjointed sub-graphs.

Generally, the clustered solution is not unique, indeed, the original graph can be partitioned in different ways. For example, the most trivial solution is to decompose each vertex into a single partition (since a clique can be composed by a single vertex).

In computer science context, the most interesting problem is to find the partitioned solution which uses as few cliques as possible, that is, the solution



(a) The graph partitioned into three cliques.

(b) The same graph partitioned into only two cliques.

Figure 2.6: The figure shows two different partitioning solutions. The number of cliques is different.

```
1  LABEL S ←
2    A = IN1 + IN2
3    B = IN3 + IN3
4  JUMP S
```

Listing 2.1: Code with an unconditional branch instruction.

with as less partition as possible. It is called the *minimum clique cover*.

In computational complexity theory finding minimum clique cover is *NP-hard*, and its decision version is *NP-complete*.

Figure 2.6 shows two different partitioning solutions. On the left, the original graph (composed by four vertices) has been partitioned into three cliques. On the right, the same graph, instead, has been partitioned with two cliques (note that the solution is also the *minimum*).

2.2 Compilers Definitions

2.2.1 Branch Instructions

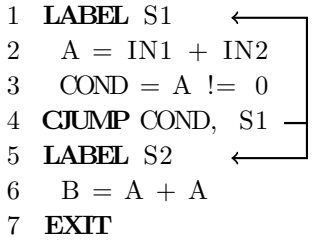
In a computer program, its statements are executed in a sequential order, at least conceptually[1]. A *branch* is a particular instruction which alters the normal sequential flow of a program, by allowing *potential* jumps among different code sections.

For example, let's consider the listing 2.1.

The instruction JUMP changes the default behaviour because it transfers the flow of execution which goes again to the first instruction.

Note that a jump instruction needs a *label* which uniquely identifies a point

[1] We do not consider any out-of-order execution optimization.



Listing 2.2: Code with a conditional branch instruction.

in the program where the execution flow has to be moved.

A branch instruction can be either *unconditional* or *conditional*.

An unconditional branch always produces a change in the execution flow without any condition. Indeed, the JUMP instruction used above is an example of unconditional branch type.

On the other hand, a conditional branch can *potentially* produce a different execution flow depending on some conditions.

We can see an example of a conditional branch in the listing 2.2.

The instruction CJUMP jumps on the first statement if the tested condition is verified, i.e. A != 0, otherwise the execution will continue as usual with the next statement.

2.2.2 Basic Block

When we are analysing the *control flow* in a code, we need to focus on those instructions which change the flow itself.

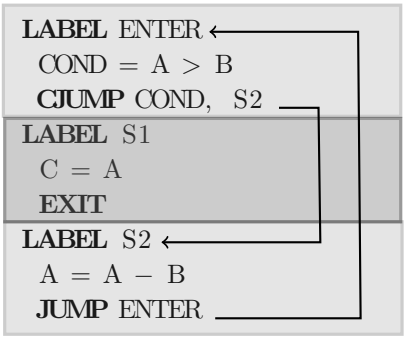
For this purpose, we can lump together any sequence of non-branch instructions into one block. We call that macro-set of instructions: *basic block*.

A *basic block* is a straight-line code sequence that is always entered at the beginning and exited at the end [2]. In other words, a basic block does not contain branch instructions except at the entry (the first statement) and at the exit (the last statement of the block). Moreover, a basic block can have a label only at its beginning. Indeed, a label always identifies the entry point of a basic block.

Let's consider the snippet of code 2.3, we can identify three different basic blocks.

The first basic block starts with the first statement and ends with the first branch instruction: the CJUMP at line 3.

After the branch instruction, a new basic block always begins. So from the LABEL S1 (line 4) another basic block can be identified.



Listing 2.3: A snippet of code where different grey rectangles identify different basic blocks.

Since a LABEL S2 (line 7) must always be the beginning of a new block, the second block ends and the last one starts.

2.2.3 Control Flow Graph

A *Control Flow* (*CFG*) is a graph representation of computation and control flow in the program.

More formally, we can say a CFG is a *directed* graph $G(V_{bb}, E_c)$, where V_{bb} is the set of *basic blocks* and E_c is the set of edges that connect two basic blocks.

Two basic blocks $b_1, b_2 \in V_{bb}$, not necessarily different, are connected with a directed edge $(b_1, b_2) \in E_c$ if at least one of the following conditions holds:

- There is a branch instruction (conditional *or* unconditional jump) which moves the execution flow from the end of b_1 to the beginning of b_2.
- The first statement of b_2 is the next instruction in the standard execution flow after the last statement of b_1. Moreover, the last statement of b_1 does *not* has to be an unconditional jump.

We can see an example of CFG in figure 2.7. Once the code has been split into basic blocks, the set of edges E_c can be easily obtained applying the properties we have just seen. Since there are no *unconditional jumps*, consecutive[2] basic blocks are linked.

In particular, for that example, we can say:

$$(BBi, BBj) \in E_c \quad \forall i, j (i < j) \quad i, j \in \{0, 1, 2, 3\}$$

Moreover, because of the *conditional jumps*, a loop is present in the graph:

$$(BB2, BB2) \in E_c$$

[2] Consevutive in sense of default code execution code.

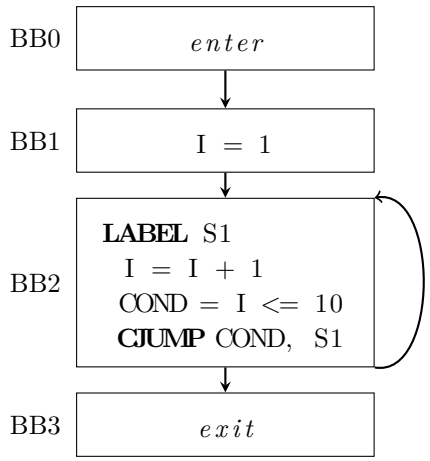


Figure 2.7: An example of *Control Flow Graph* with Basic Blocks.

2.2.4 Data Flow Graph

The *Data Flow Graph* (DFG) is a graphical representation of the data dependencies between operations.

More formally, we can say that a DFG is a *directed graph* $G(V_{op}, E_d)$ where V_{op} is the entire set of operation nodes in the program. Instead, data edges E_d denote the data dependencies between operations.

In other words, the presence of an edge $(s_i, s_j) \in E_d$ implies that the result of the operation s_i will be the input of the operation s_j. This condition also implies that s_i must be executed *before* s_j and the latter cannot start until its dependency has not been completed.

When there is a data dependency relationship among two operations (s_i, s_j), we say that s_i is a *predecessor* of s_j. Consequently, s_j is a *successor* of s_i.

Listing 2.4: Simple code with data dependency.

```
1  x = 10;
2  y = 20;
3  z = x + y;
```

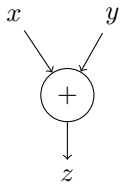


Figure 2.8: Data Flow Graph from code on the left.

Figure 2.8 shows a very simple data flow graph. Indeed, the variable z depends on the value of x and y.

2.2.5 Liveness Analysis

Liveness analysis is a *static* data-flow analysis performed by compilers in order to determine which variables are *in use* at the same time.

That kind of analysis is very important in the compiling process because of the limited number of registers on the destination target architecture.

Indeed, when we write a program with a high-level programming languages, the number of temporaries and variables is potentially unbounded. For example, in the *C* language, there are no formal limits regarding the number of automatic variables we can declare in a block.

Anyway, such programs we write must run on a machine with a bounded number of registers. Therefore, if more variables can fit into the same register the compiler can bind many variables to few registers.

If two variables are "in use" at the same time it's not possible to store their values into a single register. Indeed, the liveness analysis becomes crucial aspect in the compiling process.

We say a variable is *live* if it holds a value that may be needed in the future [2].

In order to perform liveness static analysis on a program, we need to make a specific *control-flow graph* slightly different from what we have seen in the section 2.2.3.

Each statement in the program is represented by a vertex in the graph. Moreover, let s_1 and s_2 be two statements, then there is an edge (s_1, s_2) if statement s_1 can be followed[3] by statement s_2.

An example of analysis is shown in figure 2.9.

Starting from the source code (figure 2.9a), we generate the control-flow graph of statements (figure 2.9b).

In each control flow transition (edges in the graph) we can directly annotate those variables which are currently alive.

The calculation of liveness is done by means of the usage of *USE* and *DEF* sets.

Uses and Defs. Each statement in our program can be associated with two different sets: uses and defs.

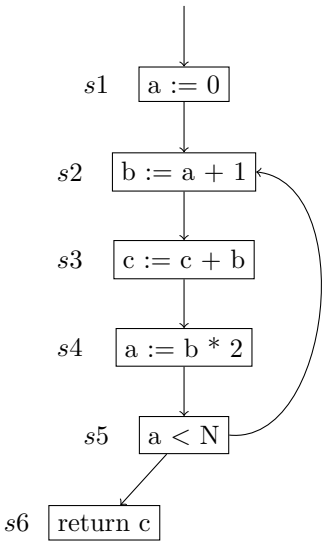
The *USE* set contains all variables used by that statement. Indeed, the statement "b := a + 1", uses the variable `a`.

$$USE = \{a\}.$$

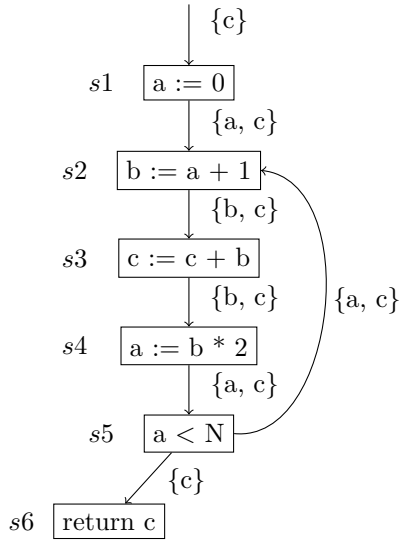
[3]We indicate the following relationship among statements in the sense of the execution control flow of the program.

```
     a := 0;
L1:  b := a + 1;
     c := c + b;
     a := b * 2;
     if a < N goto L1;
     return c;
```

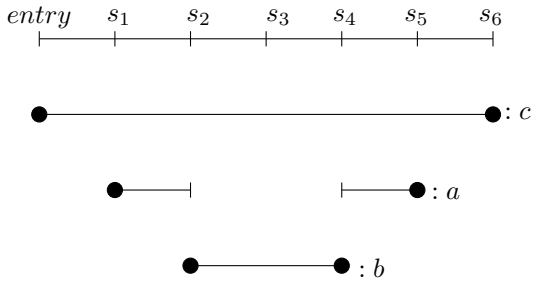
(a) Example source code.



(b) Control Flow Graph.



(c) Control Flow Graph and annotate variable liveness.



(d) Liveness line for three variables.

Figure 2.9: Liveness example analysis.

On the other hand, the *DEF* set contains all variables defined by that statement. For example, the same statement "b := a + 1", defines the variable `b`.

$$DEF = \{b\}.$$

It is often very simple to get those sets just looking at the assignment in the statement. The definition will be in the left-hand side of the assignment itself, and the uses in the right-hand side.

However other times, it is not so immediate. For example the *C* statement "++a", defines and uses the variable `a` (just like: "a := a + 1").

Once all uses and defs sets are computed for each statement, the calculation of the liveness can be solved as a result of those that are called data flow equations.

Data-Flow Equations Let p be a vertex in the control-flow graph (that is, a statement), we can indicate the set of variables $live_{in}(p)$ which are alive on the incoming arcs of the vertex p.

In the same manner, we define $live_{out}(p)$ the variables which are alive on the outgoing arcs from the vertex p.

$$live_{in}(p) = use(p) \cup \big(live_{out}(p)\ def(p)\big),$$
$$live_{out}(p) = \bigcup_{\forall q \in succ(p)} live_{in}(q).$$

The solution of data-flow equations is obtained by iterations.

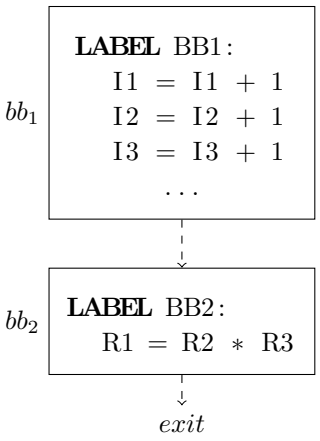
2.2.6 Code Motion

Code motion techniques are common strategies used by compilers in order to exploit the *instruction level parallelism*. Indeed, when a target architecture presents a considerable level of parallelism resources, the analysis of individual basic blocks tend to leave many resources idle. This is especially true in the *FPGA* context, where the parallelism level can be the key to high performances.

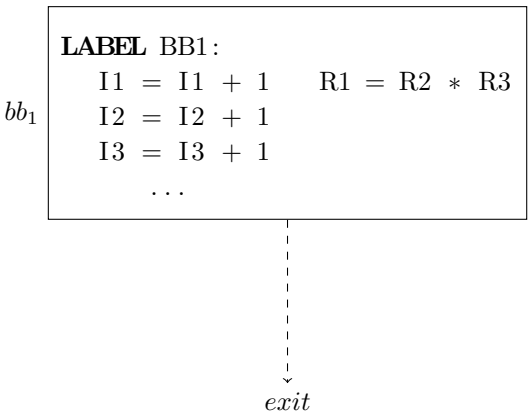
An intuitive example is shown in the figure 2.10.

Let us suppose the target architecture has two different units: one *adder* and one *multiplier*.

The initial situation (figure 2.10a) does not exploit the parallelism resources. Indeed, since two basic block will be scheduled *independently*, first only the adder will be used for the first basic blocks (which contains only addition operations)



(a) Two consecutive basic blocks before code motion.



(b) After code motion, the multiplication has been moved in bb_1 and bb_2 has been deleted because empty.

Figure 2.10: An example of code motion among two basic blocks.

while the multiplier will be idle. After that, the multiplier will be used for the second basic block and the adder will be idle.

It is clear that the resources will not be exploited in parallel and the produced code is not optimized for that architecture.

Anyway, if the multiplication operation is moved in the first basic block (figure 2.10b, the scheduler can execute in parallel two operations, that is, one addition and one multiplication because two modules are available.

In according to [3] there are four different techniques in which the code can be moved among basic blocks.

Movement between Control Equivalent Blocks

This is the simplest and most convenient solution because it does not introduce additional issues or overhead.

The operation is moved between basic blocks which are *control equivalent*.

When a basic block bb_i *dominates* another basic block bb_j and bb_j *post-dominates* bb_i, we say that bb_i and bb_j are control equivalent, meaning that one is executed when and only when the other is.

In this case, no additional operations are ever executed and no compensation code is needed.

The figure 2.11a shows an example of that situation where s_i is moved upward in bb_1 and s_j is moved downward in bb_4.

Indeed, we can notice that blocks bb_1 and bb_4 are control equivalent.

Speculation and Code Duplication

This case happens when:

- *Upward Movement:* the source basic block does not *post-dominates* the destination basic block.
- *Downward Movement:* the source basic block does not *dominate* the destination basic block.

In this case, extra operations may be executed.

In particular, the operation is moved between two basic blocks which are not controlled equivalent. That means for a particular control flow the operation is executed and after discarded because its output would be needed for another different control flow path.

The figure 2.11b shows an example of that situation.

For example, regard to the figure, the operation s_i has been moved from the block bb_2 to the block bb_3. We can say the operation is *speculated*. For what concerns the control flow, bb_1 will always be executed and so the statement s_i. However, the original basic block bb_2 is a conditional branch so it may or may not be executed. In other words, with code motion solution, the operation s_i will be always executed even if its result cannot be used because control flow may not pass through bb_2.

In such cases, it is important that the execution of a speculated operation can be executed in the destination basic block without introducing overhead, that is, it uses only resources that otherwise would be idle.

Code Compensation

This case happens when:

- *Upward Movement:* the source basic block does *post-dominate* the destination basic block but the latter does not *dominates* the source.
- *Downward Movement:* the source basic block does *dominate* the destination basic block but the latter does not *post-dominates* the source.

In this case *compensation* code is needed. The control flow path with the additional code may be slower, so it is important that the other optimized paths are more frequently executed.

The figure 2.11c shows an example of that situation.

Both Disadvantages

This last case is the union of disadvantages of the second and third cases.

It happens when source and destination basic block are not related by any dominance or post-dominance relationship.

2.3 Hardware Synthesis

In many different science and engineering contexts, a system, or a process, can be represented with several *levels of abstractions*. The same principle is worth in hardware description and synthesis.

A circuit can be described in different domains, as well as in different layers. Indeed, when we think to a digital circuit, for example, we could be interested in its functionalities or its structure.

We can ask what kind of algorithm the circuit implements or what problem it solves. Conversely, we could concern with the number of registers used in order to implement that algorithm or where the data have to be stored.

In the figure 2.12 (called *Y-char* in literature[4]) different types of synthesis have been defined.

The axes represent three different domains of description: behavioural, structural and physical. Along each axis, there are different levels of abstraction for that domain. Indeed, when we move far from the centre of the figure, the level becomes more abstract.

A *synthesis* mechanism is a *translation* process which converts a *behaviour description* into a *structural description*.

Practically we start from a more or less *abstract language* which allows us to describe a functionality or an algorithm, then the synthesis process allows the translation of that language into another language which describes how that functionality or algorithm has to be implemented on a specific architecture.

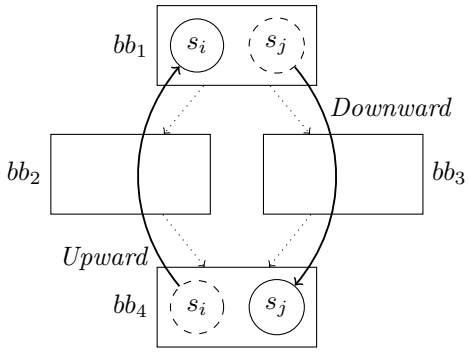
The strength of that approach is that it gives us the possibility of split the problem in several aspects. When we describe the behaviour of our system, its functionalities, we don't care about how the system will be built.

The synthesis process, in fact, is quite similar to a *compilation* process of a programming language such as C or C++ into an assembly language.

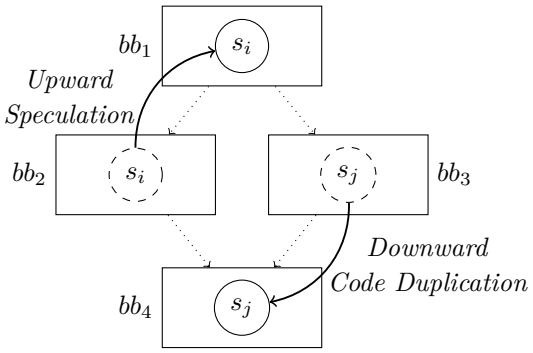
In the C language, for example, we don't have to care about in which register variables will be stored, we just describe the data structure and the execution flow. In that way, we can describe an algorithm, that is, a functionality.

Successively the *compiler* will traduce the C source code into an assembly machine-dependent language which will focus on different aspects of that domain, for example (for a general purpose CPU) the registers allocation.

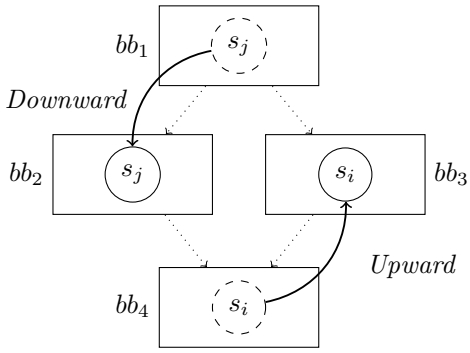
Starting from those concepts, we can define the hardware synthesis as that process in which an algorithm description is translated into a hardware specification which implements the algorithm itself.



(a) Upward and Downward code movement between control equivalent basic blocks.



(b) Upward and Downward code movement where speculation and code duplication are needed.



(c) Upward and Downward code movement where code compensation is needed.

Figure 2.11: The figure shows three different types of code motion.

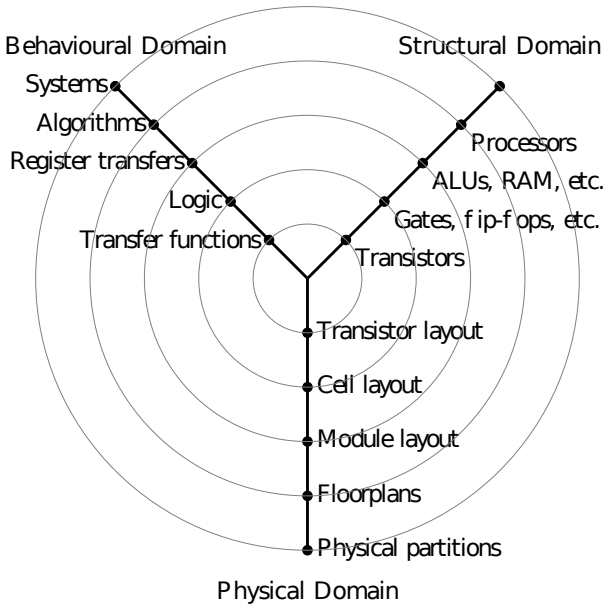


Figure 2.12: Gajski-Kuhn Y-chart.

2.3.1 Register Allocation

In register allocation problem we have to find a suitable mapping among storages values and storage modules, indeed, it's often referred as register *binding* also.

We need to bind each storage values in a proper register in order to store the data information about different clock cycles.

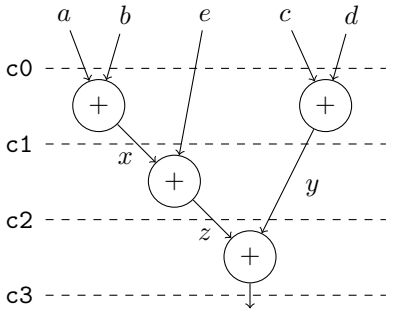
The final goal could be, for example, to minimize the number of storage modules or registers allocated. This aspect is quite crucial because it represents a complex decision problem related with other aspects in the synthesis, as we are going to see later.

Moreover, the allocated number of module storages and the how variables are mapped to them can have a large impact on the entire performances of the circuit.

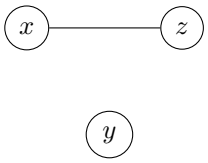
Finally, it also represents an important aspect in the methodologies presented in later sections and the overall theme proposed in this thesis.

General aspect. If we isolate and only focus on the problem of registers allocation, the only important aspect is to minimize the number of registers allocated.

The essential idea is to start from the *liveness* concept of variables. Indeed, in the section 2.2.5, we have preliminarily seen how a compiler perform static analysis on the variables (which *store* data information in the program) in order to minimize the number of registers simultaneously needed.



(a) An example of *Data Flow Graph*. The dashed lines are the cycles intervals $(c_0, c_1, \dots)$.



(b) The compatibility graph taking into account only storage values x, y, and z from the figure 2.13a.

Figure 2.13: A simple example of *compatibility graph* starting from data flow analysis.

Going into further details, we can build a *compatibility graph* among storage values. Such graph synthetically indicates when two storages values are compatibles, that is, they can be bound on the same register.

In [5] the compatibility graph is formally defined:

$$G_s = (V_s,\, W),$$

where the set of vertices V_s are *storages values* and the set of arcs is defined as:

$$W = \left\{ \left(v_i,\, v_j \quad | \quad (\ll w(v_i), P(v_i) \gg \;||\; \ll w(v_j), P(v_j) \gg) = false \right) \right\}.$$

Where $w(v_i)$ and $P(v_i)$ are respectively the last cycles in which the storage values v_i is written and read.

The notation $\ll x_1, y_1 \gg || \ll x_2, y_2 \gg$ is a formal notation to express the overlapping of two intervals, $[x_1, y_1]$ and $[x_2, y_2]$.

In other words, the compatibility graph connects those storage values where their liveness does not overlap.

Figure 2.13 shows an example of compatibility analysis starting from the *data flow graph* and *temporal scheduling information*.

The synthesized function can be easily expressed in the form:

$$out = \Big[\underbrace{\big(\underbrace{(a + b)}_{x} + e \big)}_{z} + \underbrace{(c + d)}_{y} \Big]$$

Exploiting the *hardware parallelism* we can implement it the *datapath* drawn in the figure 2.13a.

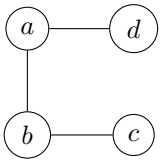


Figure 2.14: A compatibility graph among four storage values.

Anyway, it easy to see that the variables x and y, for example, overlap their liveness because the additions in the first clock cycle are executed in parallel.

On the other hand, the storage values x and z do not overlap because x is used in the clock cycle c_1 for the last time as input, while the variable z is defined as the output of the same cycle.

The reduced compatibility graph is shown in the figure 2.13b. In order to keep the figure more clear the analysis considers only variables x, y, and z. Indeed, compatibility graphs are often plenty of arcs and that makes necessary the usage of heuristics for solving some decision minimization programs, as we are going to see in a while.

Finally, in that simple example, we may deduce that storage values x and z can be assigned to the same register.

In the example just shown, it is immediate to individuate the minimum number of registers because there is only one compatibility.

The minimization problem of register's number can be reduced to as the same problem of Finding a *minimum clique cover* on the compatibility graph.

Complexity consideration. However, compatibility relation does not have *transitivity* property. For example, in the figure 2.14, the vertex a is compatible with b which is compatible with c. However, this does not imply that a and c are compatible, indeed, in the example, they are not.

Therefore, compatibility graphs are not complete graphs. This makes the problem of finding the minimum number of cliques a *NP-hard* problem.

In the figure 2.14, for example, there are different solutions (with different efficiencies) for the binding problem. If we decide to assign the storage values a and b to the same register, then we are forced to assign the remain values d and c to different registers obtaining a total of three registers.

Another better solution is to assign the pair (a, d) and (b, c) to same registers, obtaining a better solution with only two different registers.

Other aspects: So far we focused only on the critical aspect to reduce the number of registers or storage modules needed to preserve storage values informations without considering anything else.

However, it is very important to be aware that register allocation is not an isolated problem. In general, it is not a good idea to solve the minimization problem without correlating information deriving from other stages in the synthesis process.

As introduction example, we can just think that in finding the minimum number of cliques in a graph there is not a unique solution: different solutions can have the same number of cliques.

Although this does not make any difference in the number of registers allocated, to choosing a solution instead of another it can bring to very different hardware descriptions with very different performances.

2.3.2 Module Allocation

Under certain aspects, module allocation is a problem quite similar to what we have seen in the section 2.3.1, for register allocation. Indeed, the general idea is to build a *compatibility graph* in order to understand which resources (registers or modules) can be shared or not.

The module allocation (or module binding) problem is the phase during synthesis process in which each *operation* is assigned to a specific *functional unit*.

In this context, we refer to a module or a functional unit in the same manner, without any distinctions.

Two operations are compatible if both they can be executed on the same functional unit and they have been scheduled in different control cycles.

Starting from that consideration, the compatibility graph is built for each type of operation. Successively, in order to find which operations can be bound to the same module, the problem can be easily reduced in finding the minimum clique covering of the compatibility graph.

Therefore, every clique in the final solution will represent a set of operations which can be bound to the same functional unit.

Some Additional Considerations. The module allocation problem is very complex. The simple reduction to the minimum clique covering cannot be sufficient because it can lead to infeasible solutions. This problem is commonly referred to as the creation of *false loops*.

In order to solve that kind of problem is necessary to preprocess the compatibility graph and erase some compatibilities which, otherwise, would bring to false loop in the final result.

Moreover, it is important to be aware that module and register allocation can strongly depend on each other. This fact can bring further complexity in order to optimize the sharing possibility.

Finally, we have to consider another possible side effect in resource sharing: the possibility of introducing multiplexers. Indeed, when two signals have to reach the same resource a multiplexer has to be added at the input port of the functional unit. Generally, that happens in resource sharing because different inputs use the same module.

From a practical point of view, adding a multiplexer can add an increment in the usage area of the final circuit making the sharing less effective in terms of area performances. However, this represents a small overhead.

Instead, an important problem can happen when the adding of the multiplexer change the latency on the data path. Moreover, we also have to consider the delay produced by the *Finite State Machine* in order to control the multiplexer itself.

In other words, it may happen that resource sharing can introduce a non-negligible overhead which cannot fit in the timing constraints, especially when modules in question are heavy in terms of delay.

In conclusion, the possibility of introducing a multiplexer can significantly decrease the chance to obtain a resource sharing.

2.3.3 Scheduling

One of the critical aspects in the *HLS* (High-Level Synthesis) is the *scheduling* phase. Indeed, at this stage, it is decided in which control step each operation is executed. The scheduling result can considerably affect the final solution in terms of *time performance*, and the final *area* implementation.

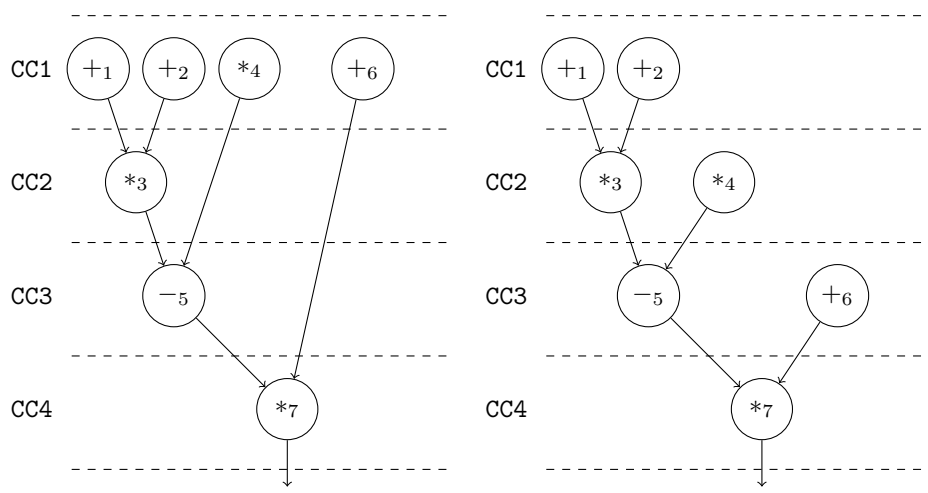
Generally, a scheduling algorithm is usually computed starting from an internal data representation such as the *Control-DataFlow Graph* (CDFG) [6].

The result of a scheduling algorithm is a partition of the control flow in sub-graphs so that each sub-graph is executed in one cycle.

In *HLS* context the control flow is driven by a *Finite-State Machine* (*FSM*) which allows transitions among control cycles. Indeed, we can also say that the FSM is the result of the scheduling phase.

Over the years, many different scheduling techniques have been proposed in order to find feasible solutions which increase the overall performances. However, two important basic scheduling approach deserves to be mentioned because of their relevance and benefit even in more advanced other techniques.

Indeed a scheduling algorithm can be designed in order to deal with different problems or to minimize a particular objective, as we are going to see in the successive chapter.



(a) ASAP. (b) ALAP.

Figure 2.15: The same data-dependencies scheduled with ASAP and ALAP.

ASAP

ASAP scheduling algorithm consists of executing an operation "*as soon as possible*". The algorithm determines the first possible control cycle in which an operation must begin its execution.

Indeed, when an operation has to be executed we have to consider it may require some input from other operations. Therefore an operation requires waiting for completion of its predecessors.

The figure 2.15a shows an example of ASAP scheduling.

ALAP

ALAP scheduling is the opposite of ASAP. Indeed, each operation will be scheduled "*as late as possible*". The algorithm determines the latest possible control cycle in which an operation must begin its execution.

ASAP and ALAP algorithms are fundamental because the union of their information can indicate a kind of mobility of an operation. Therefore each operation can be characterized by a range of cycles in which its execution can begin.

It other words, ASAP and ALAP represent *bounds* cycles in scheduling problem.

2.3.4 PandA Framework and Bambu Tool

PandA [7] is an *open source* project developed at Politecnico di Milano (Italy). The primary objective of the PandA project is to develop a usable framework that will enable the research of new ideas in the *HW-SW Co-Design* field.

One of the principal tools in PandA framework is *Bambu*. Bambu is a free[4] framework aimed at assisting the designer during the high-level synthesis of complex applications.

Bambu allows generating the *HDL description* for a specific RTL implementation starting from a behavioural description of the specification written in *C language*.

At *front-end* level, Bambu exploits a compiler-based interface provided by *GNU Compiler Collection* (GCC). In that way, it is possible to take advantage of the effective *optimizations* techniques and code analysis provided by modern compilers.

As we are going to see in chapter 5, the implementation of the proposed methodology in this thesis work has been integrated into this open-source software by allowing the possibility to produce experimental results in a real context.

Bambu Synthesis Flow

The *synthesis flow* is that process which executes step by steps all methodologies aimed at generating hardware description starting from specifications written in high-level languages. Although the high-level synthesis process shares different concepts with traditional compilation flows, in the high-level synthesis process, the quality of the produced results is much more relevant than the execution time and complexity.

In this regard, Bambu software provides a *highly modular design flow engine* which allows the execution of complex and dynamic synthesis flows. Indeed, it is possible to add and remove synthesis passes at run-time. Moreover, it is possible to insert loops inside the process by allowing the possibility to turn back in the flow and executing a methodology more than one time with different input information.

[4]Free in the sense that it respects the user's freedom.

Chapter 3

State of the Art

In this chapter, the State of the Art will be reviewed. In particular, it will be discussed different works which have inspired the content and the purpose of this thesis.

Whenever possible, a brief analysis will be provided on the resource aware context.

3.1 Balanced Scheduling

Balanced scheduling is a scheduler routine proposed in [8] by Zaretsky et al.

The summary idea is that to *distribute* operations across control cycles uniformly and to reduce *critical timing paths* in the absence of an accurate delay model.

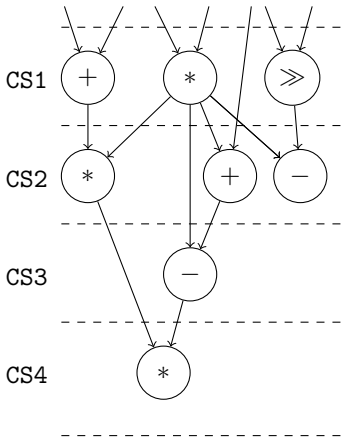
This methodology is compared with results produced by schedulers such as *ASAP* and *ALAP* which tend to distribute the operations in an *unbalanced* result in terms of number of instructions scheduled per clock cycles.

A comparative example is shown in figure 3.1.

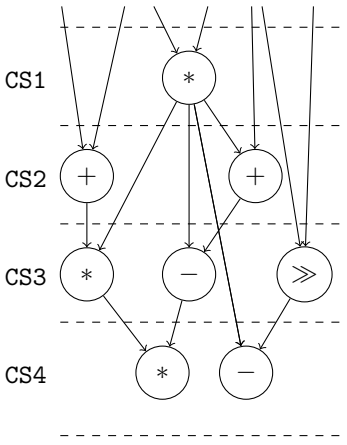
As we can see, in the *ASAP* solution (figure 3.1a) a large number of operations are executed within the first few cycles, followed by fewer operations in the later cycles. Indeed, three operations are executed in both cycles 1 and 2.

On the other hand, the *ALAP* solution (figure 3.1b) fewer operations are executed early on, followed by a large number of operations in the last few cycles.

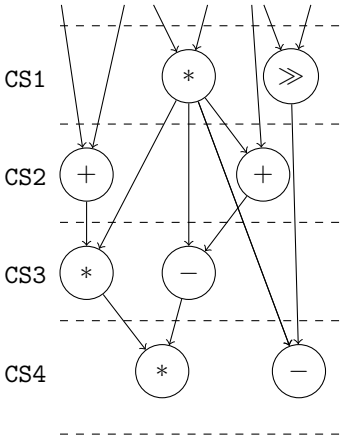
Finally, as expected, the balanced solution (figure 3.1c) produces a distributed number of operation in each control cycle. In particular, for that example, each cycle executes only two different operations.



(a) ASAP.



(b) ALAP.



(c) Balanced Scheduling.

Figure 3.1: Comparison of the same data path with three different scheduling solutions.

3.1.1 Methodology Summary

The balanced scheduling routine starts from the ASAP solution, then that average load factor is computed by the following:

$$avg\_load = \frac{num\_ops}{num\_cycles},$$

where obliviously *num_cycles* is the number of cycles obtained in ASAP result.

For example, with regard to the figure 3.1a, we obtain:

$$avg\_load = \frac{8}{4} = 2.$$

Each basic block is analysed individually.

Indeed, for each basic block, a *dependency analysis* is performed. Such analysis aims to compute the number of dependencies for each operation, that is, the *number of predecessors*.

That information is particularly interesting because, as we are going to see in a while, the methodology tends to postpone those operations which have a bigger number of dependencies. The idea is that the moving of an operation with different predecessors allows major flexibility in the *data dependency constraints*. Therefore more operations obtain more chance to be moved and the algorithm avoids getting stuck in a fixed *suboptimal* solution where no operations can be postponed.

After dependency analysis, since ASAP result tends to schedule most operations within the first cycles, the balancing algorithm starts from the *last cycle* identified by the ASAP solution which probably contains *fewer* operations than the average load factor.

Therefore, starting from the last cycle and going toward the first, if the cycle contains less operation than the average load factor then it will be balanced.

The balancing procedure consists of analysing of previous cycles and looking for some operations with the largest number of dependencies and postpone them in the unbalanced cycle.

3.1.2 Brief Analysis

Balancing Scheduling algorithms shows different improvements in its experimental results. In particular, it aims to increase *timing performance* distributing operation among control cycles in order to reduce critical paths.

However, moved operations are selected in accordance with their dependencies number to increase the mobility possibilities. In other words, no distinction about the usage of resources or the operation types.

The idea of balancing in this kind of schedule is only aware of latency performances.

3.2 SDC Scheduling

System of Difference Constraints (SDC) Scheduling is a new scheduler proposed in [9] by Cong and Zhang.

This scheduler is mathematically modelled as a set of *difference constraints*, that is, a set of dis-equations in the form:

$$x - y \leq b,$$

where x and y are two *integer* variables; b is a constant.

Indeed, the general idea behind the proposed scheduler is that to model the scheduling problem by means of an *Integer Linear Programming* (*ILP*) model.

3.2.1 Model Definition

Variables Definition

Let V_{op} be the set of operation in the *CDFG*. Each operation $v \in V_{op}$ is associated with a *set* of scheduling variables:

$$\big\{sv_i(v) \quad | \quad i \in [0,\, L_v]\big\},$$

where L_v is the *latency* of the operation v.

Each scheduling variable is defined in the set of natural number, that is:

$$\forall v \in V_{op}, \quad \forall i \in [0, L_v] \quad : \quad sv_i(v) \in \mathbb{N} \cup \{0\}$$

Essentially, the value of a scheduling variable captures the relative temporal position (in terms of control state) of an operation.

For example, $sv_0(v') = 5$ means that the operation v' starts (note the subscript 0) at the fifth control step in the final schedule.

Moreover, a *continuity constraint* has to be introduced:

$$\text{If } Lv \geq 1, \quad \forall v \in V_{op}, \quad \forall i \in [1, L_v] \quad : \quad sv_i(v) = sv_{i-1}(v) + 1$$

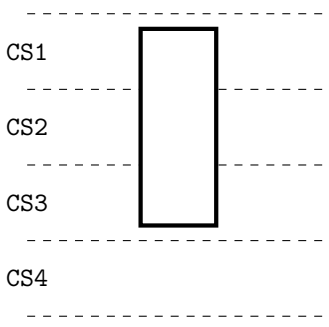
This constraint simply avoids splitting the scheduling of an operation. In other words, if an operation begins in a certain control step, it has to be completed in the immediate successor control steps: it cannot be *interrupted*.

Figure 3.2 shows an example: the sub-figure on the right is not allowed in this type of scheduler.

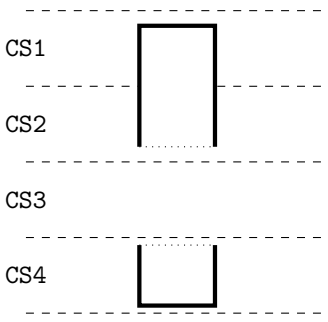
Finally, the last definitions:

- The control step in which an operation v begins:

$$sv_{beg}(v) \equiv sv_0(v).$$



(a) The operation is not interrupted. It begins at the control step 1 and completed at the control step 3.



(b) The operation is interrupted at the control step 2 and resumed at the control stage 4. Note that at stage 3 the operation is not executing.

Figure 3.2: An operation (with a *latency* of 3 cycles) scheduled with and without *preemption*.

- The control step in which an operation v ends:

$$sv_{end}(v) \equiv sv_{L_v}(v).$$

- An artificial node for each basic block bb which represents the starting operation in a basic block:

$$ssrc(bb).$$

  We named it *super-source node*.

- An artificial node for each basic block bb which represents the ending operation in a basic block:

$$ssnk(bb).$$

  We named it *super-sink node*.

Constraints

Using the scheduling variables, we can model all needed constraint in order to obtain a *feasible* schedule.

Data Dependency. Those constraints keep the valid data dependency relationships among the operations.

$$\forall e(v_i, v_j) \in E_d \quad : \quad sv_{end}(v_i) - sv_{beg}(v_j) \leq 0$$

Essentially, if data dependency among two operations v_i and v_j exists then v_i must be completed before v_j starting.

Indeed, the formula can be rewritten in the form:

$$sv_{end}(v_i) \leq sv_{beg}(v_j)$$

Control Dependency. This constraint is about the dependencies among basic blocks. In other words, some basic blocks must to be scheduled before others.

This can be achieved with the dis-equations:

$$\forall e_c(bb_i, bb_j) \in E_c \quad : \quad sv_{end}(ssnk(bb_i)) - sv_{beg}(ssrc(bb_j)) \leq 0.$$

Timing Constraints. In the hardware specifications, it can happen to deal with constraints related to the timing.

Sometimes, for example, it is needed to *synchronize* two different hardware logics in order to allow data communication. Indeed, it often occurs when we need to deal with *I/O* components.

For those reasons, it may be very useful to have the possibility to force the scheduling to meet some timing criteria.

In SDC scheduling it is possible to model timing constraint with different constraints.

- Minimum timing among two operations:

  $$sv_{beg}(v_i) - sv_{beg}(v_j) \leq -l_{ij}.$$

  It allows expressing the minimum number of cycles that must elapse between one operation and the other.

- Maximum timing among two operations:

  $$sv_{beg}(v_j) - sv_{beg}(v_i) \leq u_{ij}.$$

  On the other hand, it specifies the maximum number of cycles between two different operations. Combining this constraint with the previous one, it is possible to establish the exact number of cycles between two operations.

- Latency on CDFG:

  $$sv_{end}\big(ssnk(bb_j)\big) - sv_{beg}\big(ssrc(bb_i)\big) \leq T_{lat}.$$

  It can be used to force the maximum latency between an ending basic block and the start of another one.

However, it is possible to design another kind of constraint in order to model another type of needs, in [9] other types have been proposed.

Resource Constraints. Since the scope of this work is especially focused on the resource allocation, it is useful to examine how the resources limit is faced by this kind of proposed schedule.

Practically, an analysis over all basic blocks is performed. All nodes in the *DFG* of basic blocks are *topologically* examined. If the analysis notices a conflict over a limited type of resource, then the following constraint will be added to the scheduling formulation:

$$sv_{beg}(v_i^\pi) - sv_{beg}(v_j^\pi) \leq -Latency(v_i^\pi).$$

This constraint forces the execution of two different operations (which use the same resource π) in *sequence*. Indeed, the operation v_j can start only after all the operation v_i as been executed.

Making this analysis it is possible to avoid scheduling operations of the same type in overlapped cycles.

Objective Functions

Once all constraints have been defined, the schedule solution can be achieved by using an objective function and solving the *ILP* model.

Different objective functions can be taken into account, depending on the form of optimization one wants to obtain.

Just to clarify with an example, the *ALAP* and *ASAP* scheduling can be obtained with the following objective functions:

- ALAP

  $$min \sum_{v \in V_{op}} sv_{beg}(v).$$

  The starting time of each operation is *minimized*.

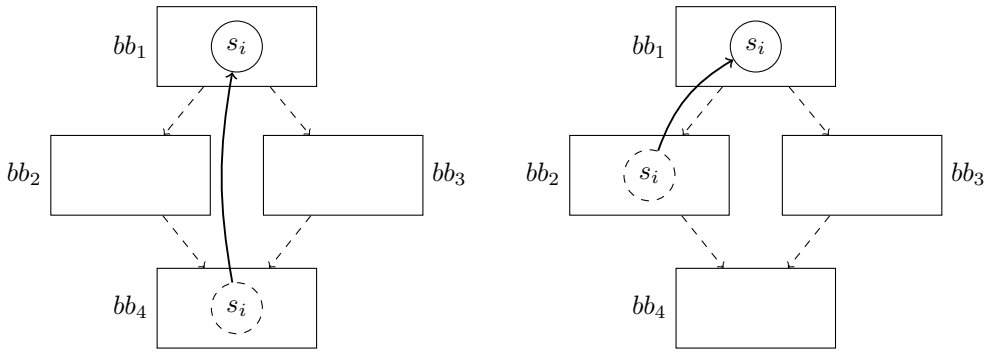
- ASAP

  $$max \sum_{v \in V_{op}} sv_{beg}(v).$$

  The starting time of each operation is *maximized*.

3.2.2 Brief Analysis

As we have seen in the previous section, SDC scheduling is able to deal resource constraints and avoid to schedule same types of operations in the same cycle when no more modules can be bound. However, this scheduling does not directly provide an effective solution focused on the *resource sharing*. The resource constraint is something which can be designed as a *precondition* for the synthesis.



(a) The operation has been moved from basic block bb_4 to the basic block bb_1.

(b) The operation has been moved from basic block bb_2 to the basic block bb_1.

Figure 3.3: The figure shows different typologies of code motions on *Control Flow Graph*. On the left a non-speculative movement because bb_4 and bb_1 are on the same control flow trace. On the right a speculative movement because bb_4 and bb_2 are on different control flow traces. Indeed, bb_2 follows a conditional branch.

Moreover, the approach of SDC scheduling does not support *code motion* and, thus, the possibility of obtaining an optimal solution in terms of resource sharing is generally lower. In other words, the *solution space* of the SDC scheduling model is limited by the intermediate code representation given as input.

3.3 Speculative SDC Scheduling

Speculative SDC Scheduling is an extension of SDC Scheduling (section 3.2) aimed at supporting code motion proposed in [10] by Lattuada and Ferrandi.

Code motion are classified in two different class (figure 3.3):

- *Non-speculative:* when an operation is moved from a basic block to another one which is in the same trace. An example is shown in the figure 3.3a.
- *Speculative:* when an operation is moved in two basic blocks which are not in the same trance. In other words, the operation is anticipated with respect to the conditional branch. An example is shown in the figure 3.3b.

The proposed methodology recalls the *ILP model* we have already seen in the SDC scheduling in the section 3.2.1. However, it is worth emphasizing some differences.

In the *original* SDC scheduling all operations can be considered in the model formulation, however, control dependencies constraints do not allow operations

speculation because by using a super source node (*ssrc*) which represents the starting point in a basic block.

Instead, Speculative SDC scheduling has to be independently applied only on each *intra loop* of the function, where the entire function is the most external loop.

Therefore the significant difference in Speculative SDC scheduling is the *lack* of the super source node (*ssrc*) and the absence of control dependency constraints. Because of that difference, the operations can be even scheduled before of what it would be the beginning of its originating basic block. Indeed we obtain speculation and code motion between basic blocks.

Moreover, because of *side effects* (e.g., store operations) some operations have to be explicitly marked in a special set (SE) which contains all those operations that cannot be speculated.

Therefore an additional constraint is inserted into the model in order to handle that kind of operations:

$$\forall\, v_i \in OP_k \left(v_i \in SE \wedge Cond(v_i) \neq \emptyset \quad \Longrightarrow \quad sv_{end}\big(Cond(v_i)\big) - sv_{begin}(v) \leq -1 \right).$$

where:

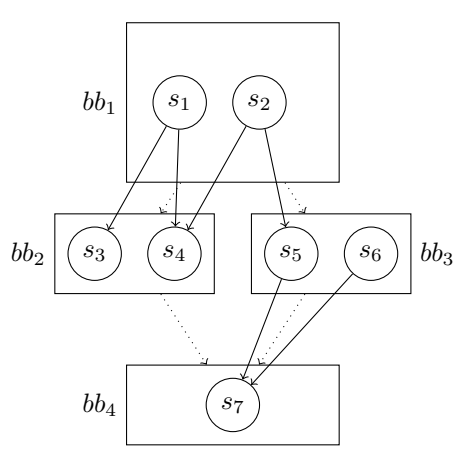
- OP_k is the set of operation in the k-th loop. It is analogous to the set V_{op} (extended to all operations) in the original SDC scheduling.
- SE is the set of those operations which cannot be speculated.
- $Cond(v_i)$ is the conditional construction (e.g., `if` statement) which controls the execution of the operation v_i.
- $sv_{begin}(v_i)$ is the control step in which an operation v_i begins, the definition is the same we have seen in the original SDC scheduling.
- $sv_{end}(v_i)$ is the control step in which an operation v_i ends, the definition is the same we have seen in the original SDC scheduling.

That constraint prevents an operation which cannot be speculated (that is, it belongs to the set SE) to be scheduled before its conditional construct. Indeed, there has to be at least one cycle of distance among them.

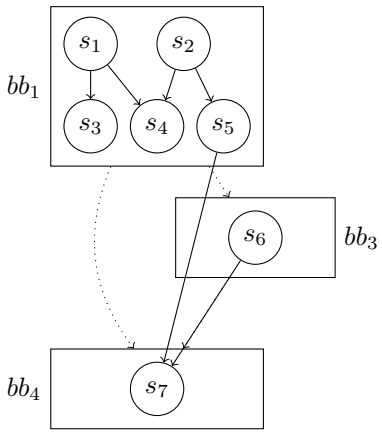
Finally, the result produced by Speculative SDC scheduling is successively used in order to identify which operations can be moved from a basic block to another in order to improve timing performances.

3.3.1 Brief Analysis

As we are going to see in the successive chapter, the methodology proposed in this thesis work starts from the code motion concept.



(a) *Control Data Flow Graph* before code movement.



(b) *Control Data Flow Graph* after code movement. The basic block bb_2 has been removed because it is empty.

Figure 3.4: Code motion example.

The algorithms proposed in [10] aim to perform code motion and code speculations in order to increase the overall performance in the final scheduling during the synthesis. Indeed final experimental results show a real improvement in timing performance in terms of both clock frequency and the number of control cycles.

The relevant principal aspect in Speculative SDC scheduling and its related heuristic code motion algorithms is the fact that there is no a really focus on the resources sharing. The code motion is applied without taking into account the type of statements moved among basic blocks.

It may happen that a basic block is almost emptied because most of its statements are speculated and moved in dominator basic blocks. However some operations cannot be moved (because of side effects for instance) and some basic blocks cannot be completely deleted. Therefore the algorithm tends to move as most as possible statements toward the *root* in the dominator tree leaving the basic blocks almost empty. This can bring to an *unbalanced* intermediate representation result where most of the body function is concentrated in the first part of the *control flow*.

Figure 3.4 shows an example of that situation.

Before the code motions, the distribution of statement in basic blocks is quite uniform. The number of statements contained in each basic block can be expressed as the following vector:

$$x^b = [2, 2, 2, 1]; \qquad \sigma^2_{x^b} = 0.25.$$

Instead, after code motion the distribution of statements is:

$$x^a = [5, 1, 1]; \qquad \sigma^2_{x^a} = 5.33.$$

We can observe a situation in which almost all statements have been moved in the first basic block. Moreover, an important observation is the fact that the basic block bb_2 is removed because all its statements have been moved. On the other hand, the basic block bb_3 still has one single statement.

That unbalanced situation may be not very effective regard to resource sharing because many operations have been assigned to the same basic block and they will be probably scheduled in the same control cycle.

3.4 Pattern-Based for FPGA Resource Reduction

In the work [11] by J. Cong and W. Jiang, a different approach has been proposed.

Usually, during the high-level synthesis process, the algorithms of modules and registers allocation build a compatibility graph over a certain type of operation. In other words, pairs of operations of the same type will be analysed in order to understand whether they can be shared on the same resource or not. That kind of approach is, in a certain sense, quite restrictive because the analysis is done only on individually among operations without considering other interesting dependencies information.

Instead, the method proposed by Cong and Jiang aims to identify recurrent patterns in the data-path. Practically, we obtain a higher level of abstraction regarding the resource sharing because an entire group of different modules can belong to the same pattern.

When a pattern is found it is possible to avoid the complete repetition of all its modules preserving the data path coherence.

In order to allow pattern identification, different parts of the data flow graph will be analysed and compared with a set of predefined patterns. Each pattern is characterized by a hash function with describes in a concise way the structure of the pattern in terms of operations types and interconnections.

An example of a pattern at scheduling stage is shown in figure 3.5.

3.5 Port Assignment for Interconnect Reduction

The port assignment problem concerns the connection between registers and functional unit ports. In [12], Cong et. al. propose a fast heuristic approach

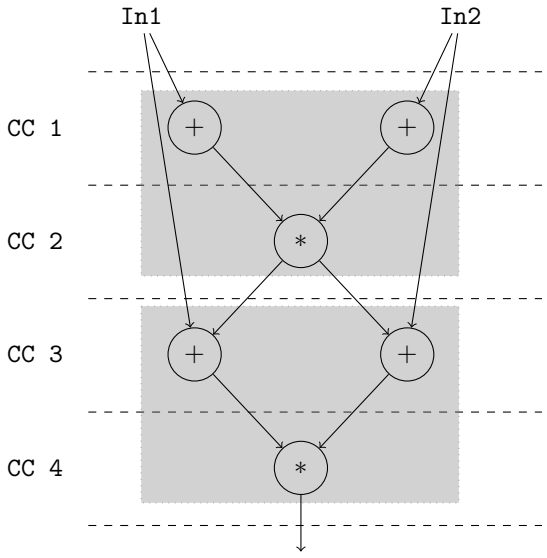


Figure 3.5: Example of pattern recognition. The grey rectangles show the same repeat pattern of operations.

.

which can solve the *NP-complete* assignment problem and reduce the number of interconnections needed for *commutative* operations.

Moreover, the interconnections between registers and functional units can represent one of the biggest parts in terms of power consumption [13].

The general idea in that work is to start from a conflict graph among registers which feed an input port (or both) of a functional unit. In other words, the heuristic algorithm minimizes the number of registers which have to be connected to both ports of the functional unit.

The port assignment problem can become quite relevant in the area performance context. Indeed, it can happen that when only one register is assigned to one port of a functional unit, there is no need of multiplexer and that situation is very favourable to resource sharing. Indeed, the lack of multiplexers allows trivial sharing of the same module because there is no more timing overhead of Finite State Machine in order to select the correct input.

3.6 Register Binding and Port Assignment for Multiplexer Optimization

In [14], Deming Chen and Jason Cong propose a *k-cofamily* algorithm, a *register binding* strategy which guarantees optimality in terms of the number of registers used while reducing the interconnection area. This approach, thus, allows

reducing the number of needed multiplexers also.

The algorithm takes as input the result of *scheduling* and *module binding* phase and it builds a *compatibility graph*. In that graph, each vertex represents a variable and an edge is added between two variables if only if they are compatible, that is, their *liveness do not overlap*.

Moreover, the graph is enriched by *weights* added for each edge. Those weights describe the cost of binding the two variables to the same register. If two variables can be bound to the same register without increasing the number of multiplexers, then the weight will be bigger.

The heuristic approach consists into find k *disjoint* paths on this compatibility graph, such that the union of all the paths covers all the vertices of the graph. Finally, all vertices which belong to the same path can be bound on the same register.

The minimization approach consists of finding the *minimum cost flow* covering all nodes. In that way, the interconnection cost will be consequently minimized.

Chapter 4

Proposed Methodology

This chapter presents the core of this thesis work.

The idea behind the proposed methodology is to start from *code motions* and *code speculations* which can significantly increase the overall performances of the final result.

As we have seen in section 3.3.1, those algorithms can be very *unbalanced* and generally they do not care about resource sharing. Instead, they push as much as possible towards massive parallelism which it usually degrades area performances.

On the other hand, the proposed methodology aims to restore a more balanced situation where *critical* operations are well separated each other.

Moreover, the approach tries to be as more conservative as possible in order to increase the area performances without affecting the overall latency of the circuit. Indeed, some critical compatible operations will be distributed among basic blocks in order to increase sharing resources possibilities without introducing *multiplexers*.

The structure of this chapter is organized as follows. First, some preliminary *definitions* will be provided in order to formalize the problem. Then, the same problem will be described in terms of *Integer Linear Programming* (ILP) model formulation. Such description highlights, in an elegant way, the constraints and objectives of the situation. Finally, the *heuristic balancing code motion* will be presented and different problematic will be explored providing a suitable solution.

4.1 Balancing In Bambu Synthesis Flow

The high modularity and flexibility of the engine flow implemented in Bambu[7] compiler allows to invalidate previous steps and thus create cycles in the syn-

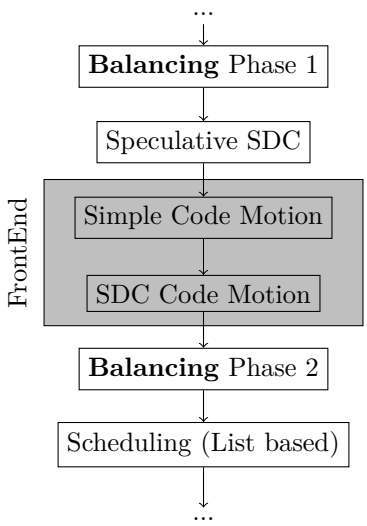


Figure 4.1: Linearised Synthesis flow in Bambu implementation.

thesis graph.

The implementation of the proposed methodology exploits this engine. Indeed, the balancing algorithm is implemented in one single node in the flow which is executed two different times.

As we are going to see in this chapter, the first stage of the algorithm is done in order to gather information about the synthesized control flow graph before code motion transformations.

Exploiting that information the algorithm is able to reconstruct the history of all speculations and code motions done and to compute some other different solutions.

The figure 4.1 synthetically shows how the synthesis flow is unfolded. The first phase of the *Balancing algorithm* is executed before *front-end* steps of *Code Motion*.

Only after all speculations and code motion have been performed, the second phase of Balancing is executed, that is, the core of the algorithm.

4.2 Balanced Code Motion

4.2.1 Preliminary Information

Let S be the set of all *statements* of function which has to be synthesized:

$$S = \{s_1, s_2, s_3, \ldots, s_n\}.$$

Similarly we can define the se of all *basic blocks* in the control flow graph:

$$B = \{bb_1, bb_2, bb_3, \ldots, bb_m\}.$$

First Balancing execution (phase 1). Actually, in the proposed methodology those set are not fixed. Practically all information about statements and their membership in basic blocks are collected in both executions of the Balancing step.

More formally, we can say that in the first execution of the Balancing step we obtain S^b and B^b which represent the statements and basic blocks sets *before* any code motion. The number of statements and basic blocks are respectively n^b and m^b.

Moreover, we define:

$$M^b : S^b \to B^b,$$

the function indicates for each statement the respective belonging basic block.

The first step execution just collects that information, thus, it does not change or modify the code or graphs.

Second Balancing execution (phase 2). After code motions have been performed, the second execution of the algorithm is executed. Again, the same information about the set of statements and basic blocks are gathered.

Therefore we define the sets: S^a and B^a with respective cardinalities n^a and m^a.

As previously seen, the membership function is again computed: M^a.

The first note to underline is the necessity to differentiate the sets of statements and basic blocks among the two execution of balancing. Indeed, It may happen that some statement or basic blocks are *removed* or *added* in the code motion algorithms.

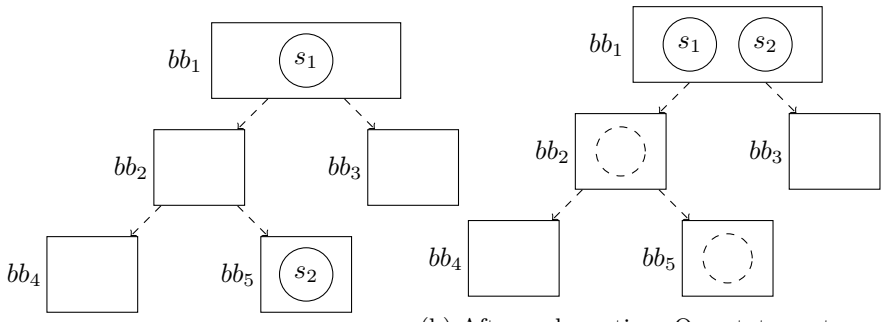
Generally, code speculations can move a lot of instructions and clean out an entire basic block and so it will be optimized and completely removed.

On the other hand, statements can be removed, for example, branch instruction. Others can be even added in order to keep data dependencies after code motion, like conditional assignments.

We can easily formalize that, *generally*:

$$S^b \neq S^a; \qquad B^b \neq B^a.$$

It should easy deduce that the computed membership function is different in the two executions of the balancing algorithm. Obviously, code motion purpose is to adjust and speculate instructions among basic blocks and, thus, change their basic block memberships.



(a) Before any code motion.

(b) After code motion. One statement has been moved.

Figure 4.2: The *dominator tree* before and after the statement s_2 has been moved.

Once all those information have been computed and stored, the algorithm core can begin. Indeed, the following sections describe steps of the balancing algorithm which are performed in the second execution of the synthesis flow (phase 2).

4.2.2 Statements Mobility

The membership function M, in the form before (M^b) and after (M^a) code motion, is somehow a representation of a state.

Analysing the differences among two states (before and after) it is possible to compute what we can define as statement *mobility*.

The figure 4.2 is self-explanatory.

The statement s_2 belongs to the basic block bb_5 before any code motion. We can write:

$$M^b(s_2) = bb_5.$$

After code motion, when balancing step is executed for the second time, the statement s_2 has been moved in the basic block bb_1, that is:

$$M^a(s_2) = bb_1$$

Exploiting that information and given the basic blocks *dominator tree*, it is possible to compute all basic blocks where an operation can be moved again.

We define the set of all possible basic blocks where a statement can be moved as mobility set. Formally:

$$\varphi : S^a \rightarrow \mathcal{P}(B^a),$$

where $\mathcal{P}$ is the power set of B^a.

Note that φ function does not need superscript because it does not make any sense to compute the mobility before code motions.

The mobility set of a statement is computed analysing the basic blocks dominator tree.

Starting from the basic block where the statement has been moved ($M^a(s)$), we need to compute the path in the tree toward the basic block where the statement was before movement ($M^b(s)$).

For instance, the figure 4.2 the path from bb_1 to bb_5 is simply:

$$bb_1 \rightarrow bb_2 \rightarrow bb_5.$$

Therefore, the mobility set of s_2 is the following:

$$\varphi(s_2) = \{bb_1, bb_2, bb_5\}.$$

It is important to consider that φ function is defined over the set of basic blocks *after* code motion ($\mathcal{P}(B^a)$). It does mean that some basic blocks could be erased or created and we need to consider that in the mobility set computation.

In the example of figure 4.2, let us suppose the basic block bb_2 has been deleted:

$$B^b \setminus B^a = \{bb_2\}.$$

The the new mobility set of the statement s_2 will be:

$$\varphi(s_2) = \{bb_1, bb_2, bb_5\} \setminus \{bb_2\} = \{bb_1, bb_5\}.$$

If the code motion introduces new basic blocks, then the dominator tree structure may have been modified, but the principle is the same.

In conclusion, we can say that mobility set represents a *space of possible solution*, in terms of placements, for a statement.

Listing 4.1: Mobility Set Computation Algorithm

```
1  compute_mobility(bb_before, bb_after) {
2    mobility := ∅;
3    DOMS := dominators_of(bb_before);
4
5    if (bb_before has not been deleted) {
6      mobility := mobility ∪ {bb_before};
7    }
8
9    for (each b in DOMS) {
10     if (b has not been deleted) {
11       mobility := mobility ∪ {b};
12     }
13
14     if (b == bb_after) {
15       break;
16     }
17   }
18
19   return mobility;
20 }
```

Mobility for created statements. The mobility definition we have seen so far cannot be directly applied to those statements which have been created after code motion steps.

Indeed, for those statement, there is no concept of a path in the dominator tree because there is no destination basic block.

Let S^c be the set of all new statements created after code motions steps, that is:

$$S^c = S^a \setminus S^b.$$

We see that:

$$\forall s \in S^c \ (\ M^b(s) \text{ is not defined}\ ).$$

The definition of mobility set can be easily extended for those statements just defining it as the single basic block where the statement itself has been created.

$$\forall s \in S^c \ \big(\varphi(s) = M^a(s)\big). \tag{4.1}$$

The algorithm presented in listing 4.1, is not executed on created statements (S^c) but the mobility function is just computed as the singleton membership function.

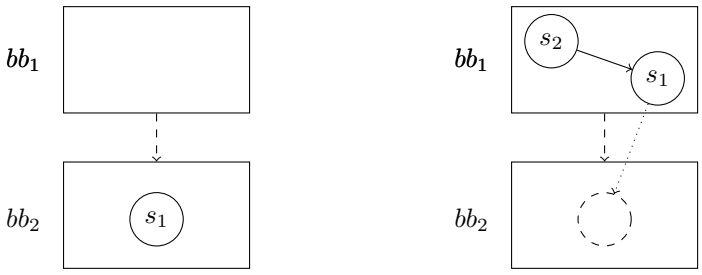
Listing 4.2: Assign mobility for a statement

```
1  assign_mobility(statement) {
2    mobility := ∅;
3    bb_before := M^b(s);// Maybe not defined
4    bb_after := M^a(s); // Alywas defined
5    if (statement ∈ S^c) {
6      mobility := bb_after;
7    } else {
8      mobility := compute_mobility(bb_before, bb_after);
9    }
10   return mobility;
11 }
```

Increasing mobility for created statements. As we have already seen, code motion and speculations can introduce new statements in the code representation. Those new operations introduced in the data flow can be very critical in the proposed methodology because of *data dependency*.

As we are going to see better in the successive sections, the mobility set of a statement can be reduced (in terms of possible basic blocks) because of other statements mobilities. Indeed, an operation which has a limited mobility can force all their *predecessors*[1].

[1] Predecessors in the data flow.



(a) Dominator tree with two basic blocks and one single statement *before* code motion.

(b) Dominator tree *after* code motion. A new statement s_2 has been created with data dependency with s_1.

Figure 4.3: A possible situation in dominator tree before and after code motion.

For that reason, statements with limited mobility can have a huge impact on the overall results of the algorithm.

In this case, all created statements (S^c) have reduced mobility, more precisely their mobility set has *one single element*.

In accordance with the expression 4.1, we can say that:

$$\forall s \in S^c \ \big(|\varphi(s)| = 1 \big),$$

since the image of $M^a(s)$ function is *one single* basic block.

In other words, all statements which belong to the set S^c are potentially critical on algorithm result. Indeed, the proposed methodology implements a strategy in order to *increase the mobility* for all those statements.

The general idea behind the mobility increment is the fact that a statement should have at the least the same mobility of its predecessors.

Let Γ be the set of *immediate predecessors* (in terms of data flow) of a statement, thus:

$$\Gamma : S \to \mathcal{P}(S).$$

Then we can extend the mobility for statements in S^c following:

$$\forall s \in S^c \quad \forall s' \in \Gamma(s) \quad \Big(\varphi(s) \supseteq \varphi(s') \Big). \tag{4.2}$$

In other words, we can intend that created statements can inherit the mobility of their predecessors.

Starting from that, we can insert additional mobilities for those statements, just evaluating mobilities of their predecessors.

With regard to the figure 4.3, we can see how code motion steps moved the statement s_1 from the basic block bb_2 to the dominator bb_1. Moreover, a new statement s_2 has been created, which is a predecessor in the data flow of s_1.

Summarily, following the mobility definition seen so far:

$$\varphi(s_1) = \{bb_1, bb_2\} \qquad \varphi(s_2) = \{bb_1\}.$$

The predecessor dependency is expressed by:

$$\Gamma(s_1) = \{s_2\}.$$

The formula 4.2 allows us to write:

$$\varphi(s_2) \supseteq \varphi(s_1).$$

So we can extend the mobility of s_2 simply adding mobility of s_1 ($\varphi(s_1)$):

$$\varphi(s_2) = \{bb_1\} \cup \{bb_2\}.$$

Listing 4.3: Increasing mobility for S^c statements

```
1  function mobility_increment_new_stmts () {
2    new_stmts := S^c;
3    new_stmts := topological_sort(new_stmts);
4
5    for (s in new_stmts) {
6      mobility := get_mobility(s);
7      preds := immediate_predecessors_of(s);
8
9      for (p in preds) {
10       mob_preds := get_mobility(s);
11       mobility := mobility ∪ mob_preds;
12     }
13   }
14 }
```

The listing 4.3 shows the pseudo algorithm which performs the mobility increment for each statement in S^c.

An interesting aspect is at line 2, the set S^c is actually topologically[2] sorted. That operation is necessary because it could be a data dependency among two or more created statements.

That is, it *may* happen that:

$$\exists\, s_i,\, s_j \in S^c\big(\, s_j \in \Gamma(s_i)\,\big).$$

Therefore, the topological analysis is fundamental in order to assign correct mobilities and avoid conflicts among created statements themselves.

4.2.3 Statement clusters

As we have already seen previously, the proposed methodology aims to balance operations among the various basic blocks in the control flow. The final purpose

[2] in terms of data dependencies

is that to allow multiple possibilities of module sharing and, thus, reduce the final target area and the number of resources used to implement the functional behaviour.

In this regard, we can say that the balancing of some kind of statements can be useless or even harmful.

Since module sharing is a complex problem the attempt of balancing some "light" operations (such as *shift*, or even *additions*) may affect the balancing of other heavier operations.

Some operations can be implemented and synthesized on a target architecture without the usage of many resources, while their sharing may introduce a more complex *data path* with the addition of *multiplexers* (which can request much more resources).

For those reasons, in the proposed methodology we only focus on those operations defined as *critical*.

In particular, the implementation in *Bambu* compiler has been targeted on *multiplication* operations.

In *FPGA* context multiplication implementations usually request *DSP* blocks on the board which may be a quite rare resource.

Therefore an important step is to *cluster* the statement in accordance with the kind of operation they have associated with.

The cluster should group statements which have a high probability to be bound on the same *functional unit*.

Formally the cluster is a *partition* K of statement set, that is:

$$S^a = \bigcup_{k \in K} S_k,$$

where S_k is a different cluster.

4.2.4 Balancing cluster

In the proposed methodology, the balancing algorithm stars from a *cluster* of statements (e.g. all statements which implements a multiplication). Let us call that cluster the set S_k.

Extended cluster

In order to keep data dependencies consistency, it is necessary to compute all possible successors (*not only immediate*) for each statement in the cluster.

Let ς be the set which associates *immediate successors* of a given statement:

$$\varsigma : S \to \mathcal{P}(S).$$

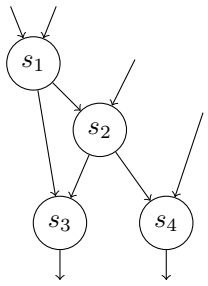


Figure 4.4: Example Data flow graph with four statements. Edges represent data dependencies.

We can define the *closure* of ς which represent *all* successors of a given statement. As usual in computer science, we are going to use the *Kleene star operator* notation:

$$\varsigma^* : S \to \mathcal{P}(S),$$

where:

$$\forall\, s_i,\, s_j \in S\Big(s_j \in \varsigma^*(s_i) \iff s_j \in \varsigma(s_i) \ \vee\ \exists s_z \in S\big(s_z \in \varsigma(s_i) \ \wedge\ s_j \in \varsigma(s_z)\big)\Big),$$

or recursively:

$$\varsigma^*(s) = \bigcup_{s' \in \varsigma(s)} \Big[\{s'\} \cup \varsigma^*(s')\Big].$$

With regard to the figure 4.4, it easy to compute the *immediate successors* from the *data flow graph*.

Indeed, for example, s_1 is directly connected to s_2, s_3 but not s_4.

That is:

$$\varsigma(s_1) = \{s_2,\, s_3\}.$$

Its closure is simply the *reachability* concept on a graph: s_4 is a successor (*not immediate*) of s_1. That's because s_4 is an immediate successor of s_2.

So:

$$\varsigma^*(s_1) = \varsigma(s_1) \cup \varsigma^*(s_2) \cup \varsigma^*(s_3) = \{s_2,\, s_3,\, s_4\}.$$

Thus, starting from the original cluster we compute an *extended cluster* S_k^E which contains even all successors statements.

More formally:

$$S_k^E = \bigcup_{s \in S_k} \Big[\{s\} \cup \varsigma^*(s)\Big]$$

Problem ILP Formalization

Starting from the *mobility sets*, the general idea is to assign each statement from the *original cluster* (S_k) in a proper basic block and try to *minimize* the number of statements which implement the same operation that belong the same basic block.

If two operations are collocated in different basic blocks, they will be surely scheduled in different *control cycles* and the probability of obtaining a sharing becomes higher.

On the other hand, two operations collocated in same basic block (especially if they are independent each other) may be scheduled in *parallel* or in the same control cycle making the sharing not possible.

In this paragraph, we are going to analyse the problem starting from an *ILP model* representation.

- **Data**

  - The number of statements in the extended cluster:

    $$N = |S_k^E|.$$

  - *Not-strict* Dominance relationship:

    $$\Delta(bb_i, bb_j) = \begin{cases} 1 & \text{if } bb_i \text{ dominates } bb_j \text{ or } bb_i = bb_j; \\ 0 & \text{otherwise.} \end{cases}$$

    $$\Delta : B^a \times B^a \to \{0, 1\}.$$

  - *Complement* of mobility function:

    $$\varphi(s)^c = B^a \setminus \varphi(s), \qquad \forall\, s \in S_k^E.$$

  - Equality function among basic blocks:

    $$\delta(bb_i, bb_j) = \begin{cases} 1 & \text{if } bb_i = bb_j; \\ 0 & \text{otherwise.} \end{cases}$$

    $$\delta : B^a \times B^a \to \{0, 1\}.$$

    It can be defined as *identity matrix* also:

    $$\delta = I^{|B^a| \times |B^a|}$$

  - Verification function on the original cluster:

    $$\rho(s_i, s_j) = \begin{cases} 1 & \text{if } s_i \in S_k \text{ and } s_j \in S_k; \\ 0 & \text{otherwise.} \end{cases}$$

$$\rho : S_k^E \times S_k^E \to \{0, 1\}.$$

This function simply returns one when the two arguments both belong to the original cluster.

- **Decision Variables**

$$x_i \in B^a, \qquad \forall\, i \in S_k^E.$$

For each statement i in the *extended cluster*, we want to decide in which basic block assign it.

- **Constraints**
We introduce two different constraints.
  - Data dependencies:

$$\Delta(x_i, x_j) = 1 \qquad \forall\, i,\, j \in S_k^E \left(i \in \Gamma^*(j)\right).$$

This formulation generates a series of constraints for each pair of statements which are dependent each other.

Indeed, given two statements i and j where i is a *predecessor* of j, then the basic block assigned for i must be a *dominator* of the basic block where j, instead, has been assigned. Note that x_i and x_j can be the same basic block.

  - Force feasible assignments:

$$x_i \neq \nu \qquad \forall\, i \in S_k^E \qquad \forall\, \nu \in \varphi^c(i).$$

Those constraints, instead, allow only assignments in according to the mobility of a statement.

- **Objective Function**
We want to *minimize* the number of statements of the *original cluster* which have been assigned to the same basic block. That is:

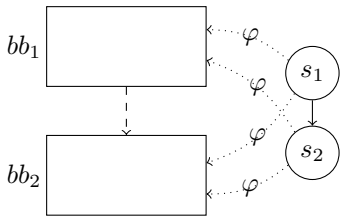
$$min\left[z = \sum_{i=1}^{N-1} \sum_{j=i+1}^{N} \delta(x_i, x_j)\, \rho(i, j)\right].$$

Heuristic balancing

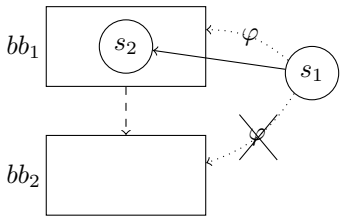
The following paragraph describes one the *fundamental* aspect in the proposed methodology.

As we have seen in the *ILP* formulation, the problem consists of an assignment: we have to decide which is the *most suitable* basic block for each statement of the *extended* cluster.

The *pseudo* algorithm is presented in the listing 4.4.



(a) At the beginning, s_1 and s_2 can be placed both in bb_1 and bb_2. Dotted arcs represent where the statement may be assigned.



(b) After s_2 has been placed in bb_1, the mobility set of s_2 has to be reduced in order to keep data dependency consistency.

Figure 4.5: Mobility reduction example because of data dependency.

Mathematically the assignment decision can be modelled as a *total* function:

$$\varkappa : S_k^E \to B^a.$$

That function is represented by the *map* "`assignment_map`" at line 2 of the algorithm.

The first step in the heuristic balancing algorithm is to assign all those statements which have a *single mobility*.

$$\forall s \in S_k^E \left(|\varphi(s)| = 1 \implies \varkappa(s) = \varphi(s) \right)$$

This operation is done in algorithm's lines 4-9.

After trivial constraints, the successive assignments will be done in *reverse topological order*[3]. Therefore the operations without successors will be assigned first, and so on.

When a statement is selected, its mobility may not be coherent with previous assignments and so we need to reduce the mobility set.

An example situation is shown in figure 4.5. As we can see in figure 4.5a, there are two dependent statements which can be assigned to both basic blocks:

$$\varphi(s_1) = \varphi(s_2) = \{bb_1, bb_2\}; \qquad \varsigma^*(s_1) = \{s_2\}.$$

Since the heuristic assignment is computed in reverse topological order, s_2 will be placed before s_1.

Let us suppose s_2 will be assigned to the basic block bb_1, that is:

$$\varkappa(s_2) = bb_1.$$

Then, it is easy to see that s_1 cannot be assigned to bb_2 anymore. Indeed, bb_2 is *dominated* by bb_1, so the control flow of the program will execute bb_1 and *after* bb_2.

[3]In terms of data dependencies.

If s_1 was assigned to the bb_2, it would be executed after s_2 and this will break data dependency: s_1 is *predecessor* and must to be executed *before*.

The mobility reduction is shown at lines 20-27 in the algorithm. Note that two dependent statements can be still assigned to the same basic block.

After the mobility set for a statement has been properly reduced, the algorithm chooses the most suitable basic block among the remaining.

The heuristic approach proceeds differently depending on whether the statement belongs to the original cluster or not.

In case the statement does not belong to the original cluster (that is, it is not really an operation to balance), then the idea is to assign it to the *less dominant* basic block.

In a set of basic blocks, the less dominant are those ones that do not dominate any other in the set. Since the dominance relation is asymmetric, that definition is always well-defined. Moreover, it is easy to demonstrate that in a mobility set there will always be exactly one less dominant basic block.

That is:

$$\forall\, s \in S^a \left[\exists!\, bb_i \in \varphi(s) \ \left(\forall bb_j \in \varphi(s) \big(bb_i \neq bb_j \quad \wedge \quad bb_i \text{ does not dominate } bb_j \big) \right) \right].$$

That's because code motion only makes sense along paths in the dominator tree in our methodology.

The heuristic approach consists in assigning successors to the less dominator basic block. The reason of that is to avoid reducing mobilities of other statements as much as possible.

On the other hand, in case the statement which has to be assigned it belongs to the original cluster, it will be select the basic block which contains less already assigned statements of the *original cluster* itself.

Let $s \in S_k$ be the statement to balance, then:

$$\underset{bb \in \varphi(s)}{\arg\min} \left| \big\{ s_i \mid \varkappa(s_i) = bb \ \wedge \ s_i \in S_k \big\} \right|.$$

In that way, we are trying to balance the statements among basic blocks. Note that we care only about the number of statements which belong to the original cluster (not extended).

Listing 4.4: Heuristic Balancing

```
1  function heuristic_balancing() {
2    assignment_map := [];
3
4    // Assign statement with mobility 1
5    for (s in S_k^E) {
6      if (|φ(s)| == 1) {
7        assignment_map[s] = φ(s);
8      }
9    }
10
11   // Until all statements have not been assigned
12   while (assignment_map.size != |S_k^E|) {
13     for (s in S_k^E) {
14       // if all successors have been already assigned
15       if (∀s' ∈ ς*(s) (∃assignment_map[s'])) {
16
17         // Get the mobility set
18         mob := φ(s);
19
20         // Reduce mobility in according to successors
21         for (s' in ς*(s)) {
22           for (m in mob) {
23             if (assignment_map[s'] dominates m) {
24                 mob := mob \ m;
25             }
26           }
27         }
28
29         b := {};
30         // Select the best basic block to assign
31         if (s ∈ S_k) {
32           // if s is in the original cluster
33           b := min[m ∈ mob : contains less stmt of S_k];
34         } else {
35           // if s is only a successors of the original cluster
36           b := less_dominator(mob);
37         }
38
39         assignment_map[s] = b;
40       }
41     }
42   }
43
44   return assignment_map;
45 }
```

4.2.5 Moving critical predecessors

The only balancing of statements among basic blocks may not be so effective in order to obtain module sharing.

As already discussed in previous chapters, one critical aspect of the sharing

problem is about the *liveness* of storage values.

In order to understand the problematic, we can refer to the figures 4.7 and 4.8.

In figure 4.7, for instance, there are two multiplications which have to be balanced over three basic blocks. In particular, one multiplication is moved from *basic block 2* to the dominated *basic block 3*. In that way, there aren't multiplications in the same basic block anymore and the distribution of multiplications over basic blocks is perfectly balanced.

Although the solution aims to increase the possibility of multiplications sharing, the final synthesis will probably produce two different multipliers without making any sharing.

The reason of that can be shown in figure 4.8.

For sake of simplicity, let us suppose each basic block (from the example in figure 4.7) is scheduled in one single control cycle, the same concept can be easily generalized without any further complex changes.

Our objective is to balance the two multiplications and increase the possibility of obtaining a sharing.

If two multiplications are scheduled in the same control cycle because they are in the same basic block, then they will be obviously bound on two different modules (figure 4.8a).

When our balancing methodology is applied, one multiplication is moved into another basic block and consequently into a different control cycle. In that case, regard to the figure 4.8b, the multiplications will be still bound on different modules (no sharing) because of *liveness of their operands*.

Indeed, as we can see, the two multiplications are driven by four addition operations which are all scheduled and completed in the first control cycle. Therefore, the multiplications inputs must be all assigned in *different registers* (liveness of their storage values overlap).

Because of different registers, the sharing would require a *fan-in* with two *multiplexers* in order to select the proper register. That kind of situation is not so useful in reducing area because despite the fact one multiplier is saved, two multiplexers will be instantiated.

The figure 4.6 give us an idea of the result data path.

During synthesis process, the module allocation phase could decide to still make no resource sharing and avoid multiplexers.

That's why our balancing methodology should take into account even some *"critical" predecessors* of those operations in the original cluster.

Before to proceed, let us take a look at the figure 4.8c. In case predecessors of the moved multiplication are also postponed, then multiplications' inputs won't overlap their liveness anymore.

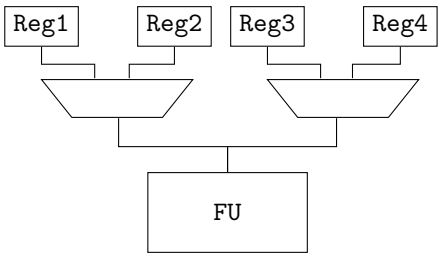
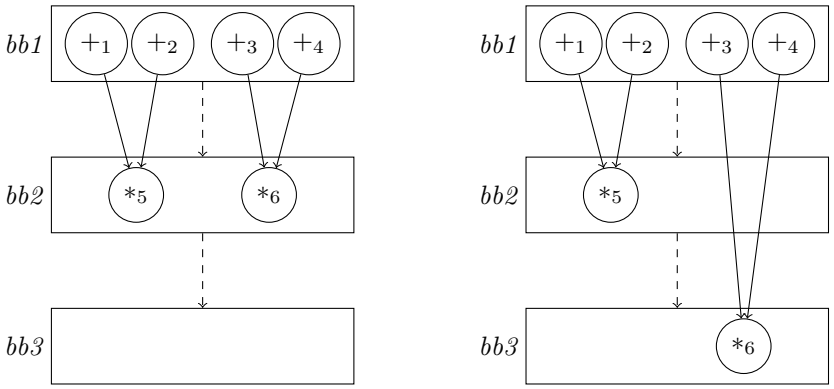


Figure 4.6: Example of sharing FU among different registers. Multiplexers are needed to select the proper register.



(a) Control Data Flow Graph *before* balancing.

(b) Control Data Flow Graph *after* balancing.

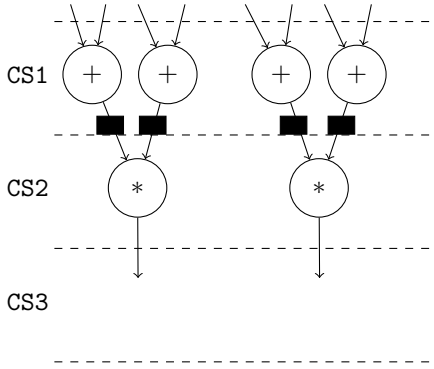
Figure 4.7: Control Data Flow Graph with four additions and two multiplications.

Indeed, the general idea is to extend our methodology to the predecessor operations in order to avoid liveness overlapping and allow resource sharing without any additional multiplexers.

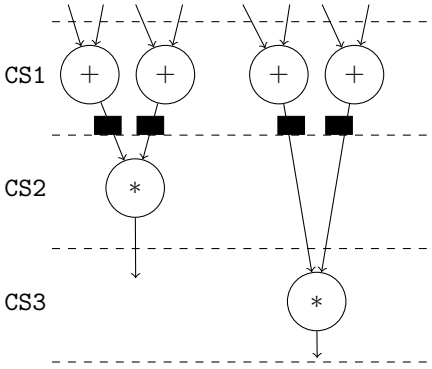
In particular, since our methodology operates at *code motion level*, the predecessors can be moved among basic blocks (and not at scheduling level among control cycles as we have seen in the example). However, there is still a strong correlation between basic blocks and control cycles.

Extended Mobility. In order to obtain effective resource sharing, we have just seen that it can be very useful to move predecessors operation in different basic blocks.

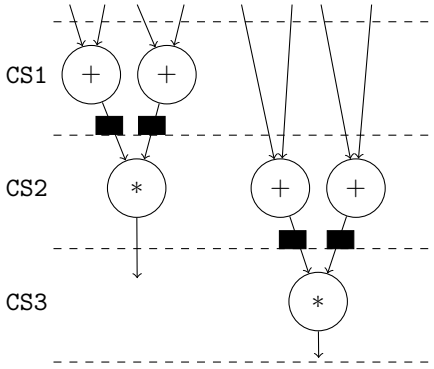
So far our methodology is based on the mobility concept obtained from the computed difference of two states: *before* and *after* code motion. With that definition the mobility of an operation, it totally depends on the fact it has been moved by code motion steps in the front-end analysis.



(a) Both multiplications have been scheduled in *cycle* 2. Each store value needs a register.



(b) Multiplications have been scheduled in different cycles, but stored values still need 4 registers because of liveness overlap.



(c) Predecessors of second multiplication have been postponed. Now storage values liveness do not overlap. It is possible to store operands into only two registers.

Figure 4.8: Example of liveness aspect in balancing methodology. The directed edges represent operation data dependencies and black rectangles represent registers for storage values among different control cycles.

It may easily happen that a predecessor we want to move has a *single mobility* (it cannot actually be moved) in according to the definition of mobility function. Indeed, the possibility of moving predecessors is quite *critical* and it requires a *less conservative* definition of mobility itself. We call it *extended mobility.*

Since extended mobility is not related to the concept of before or after code motion, there is no need to differentiate the membership function for a statement. Indeed, we are going to use the notation $M(s)$ to indicate the basic block where *currently* the statement s is assigned.

$$M : S \to B.$$

We can formally define the extended mobility function for a statement:

$$\varphi^E : S \to \mathcal{P}(B).$$

where:

$$\varphi^E(s) = \left\{ bb \in B \mid bb \notin DOMS\big(M(s)\big) \wedge \forall s' \in \varsigma(s) \left(bb \in DOMS\big(M(s')\big) \right) \right\} \cup \left\{ M(s) \right\}.$$

Intuitively, $DOMS(bb)$ function returns the set of all *strict* dominators of basic block bb.

As we can see in the first part of conjunction, the extended mobility allows movements only in dominated[4] basic blocks.

On the other hand, second conjunction part allows data dependency consistency.

A pseudo code algorithm which computes the extended mobility for a given statement s is presented in the listing 4.5.

[4]Dominated by the basic block where the statement is currently assigned.

Listing 4.5: Compute Extended Mobility

```
1  function compute_extended_mobility(s) {
2    // M(.) is the current basic block membership
3    bb_current :=  M(s);
4    ext_mobility_set := {bb_current};
5    successors := successors_of(s);
6
7    for (bb in all_basic_blocks) {
8      if (bb not dominates bb_current) {
9
10       compatible := true;
11       for (succ in successors && compatible == true) {
12         bb_succ := M(succ).
13         if (bb_succ != bb && bb not dominates bb_succ) {
14           compatible := false;
15         }
16       } // for all successors
17
18       if (compatible == true) {
19         ext_mobility_set := ext_mobility_set ∪ {bb};
20       }
21     } // if
22   } // for all basic blocks
23
24   return ext_mobility_set;
25 }
```

Heuristic Predecessor Movement. With the definition of extended mobility, it is possible to make a heuristic approach in order to move those predecessors and increase the sharing possibilities.

The idea is quite simple, we try to move the immediate predecessors in the *less dominant* basic blocks in order to decrease the liveness overlapping among storage values.

We have already defined the less dominant concept in the section 4.2.4. Practically in a set of basic blocks, the less dominant is the one which does not dominate any others in the set. For a *generic* set, there may be more than one element which enjoys that property. However, in a set built starting from a path on the dominator tree (as usually happen in code motion) *only one* basic block will be the less dominant.

The heuristic approach follows the general idea behind the technique called "*as late as possible*" (ALAP). We postpone the operations in order to minimize the number of cycles in which operands variables are alive.

Listing 4.6: Heuristic Movement Critical Predecessors

```
1  function heuristic_moving_predecessors(cluster) {
2    assignment_map := [];
3
4    for (s in cluster) {
5      preds := immediate_predecessors(s);
6
7      for (p in preds) {
8        mobility := compute_extended_mobility(p);
9        b := less_dominator(mobility);
10       assignment_map[p] := b;
11     }
12   }
13
14   return assignment_map;
15 }
```

Cycles optimization.

As we have just seen, the predecessors of a statement in the cluster can be moved in less dominant basic blocks in order to postpone their execution. Although this strategy can effectively reduce the overlapping liveness among storage values, it can decrease the timing performances.

The reason of that is quite simple. First of all, we have to consider that the proposed methodology aims to reduce the area occupation by promoting the binding of different operations on the same module (resource sharing). In particular, as we are going to see later, it can be applied to *multiplications expressions* which can have a huge impact on the resource utilizations.

Therefore, it often happens that the statements (operations) which we are balancing also have a significant impact on timing performance, that is, they may require several temporal time units in order to produce the result.

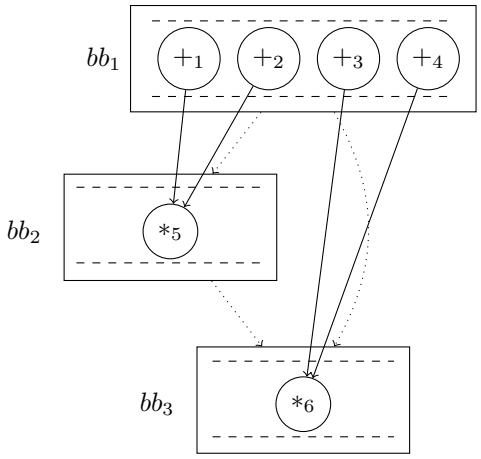
In other words, the statements, which we are trying to balance, can occupy an entire control cycle or a bit less at the scheduling stage.

Moreover, since those statements are subjected to movements towards less dominant basic blocks, they won't have any predecessor in that block and they will be probably scheduled in the first control cycles of the block.

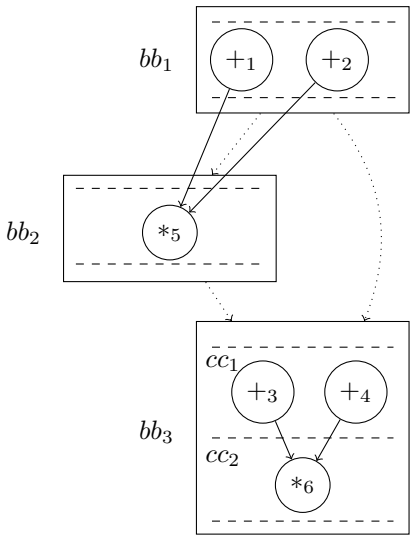
When critical predecessors movements are applied, those immediate dependencies are moved toward less dominant basic blocks also. Indeed, it may happen that the immediate predecessor will be assigned to the same block where balanced statements have been previously moved.

Therefore since statements in the original cluster can indeed fill an entire control cycle, the dependency in the same basic block may lead to an increment of the cycles number for that block.

Figure 4.9 shows an immediate example of that kind of situation.



(a) Before predecessor movements. Each basic block is scheduled in one single control cycle.



(b) After predecessors movement. The basic block bb_3 needs to be scheduled in two control cycles.

Figure 4.9: The figures shows a possible situation obtained because of the movement of predecessors of the operation s_6. The dashed lines inside the basic blocks separate different control cycles for that block.

Before predecessors movement (figure 4.9a) the operation s_6 in the basic block bb_3 has its predecessors (s_3 and s_4) in the basic block bb_1.

Each basic block is indeed scheduled in *one single* control cycle, for a total of 3 cycles.

Because of liveness overlapping among additions operations, the heuristic algorithm will move the predecessors s_3 and s_4 in the less dominated basic block. In particular, for that example, bb_1 dominates bb_2 and bb_3 but statements s_3 and s_4 can only be moved in bb_3 because of data dependency with s_6.

More formally:

$$\varphi^E(s_3) = \varphi^E(s_4) = \{bb_1, bb_3\},$$

where obviously bb_3 is the less dominant because bb_1 *dominates* bb_3.

When s_3 and s_4 are moved in the same basic block of their successor s_6 it is not possible to schedule the basic block bb_3 in a single control cycle anymore. An addition and a multiplication cannot be *chained* in a single control cycle because they will not fit[5] into it. We can see that in figure 4.9b.

Because bb_3 is now scheduled in two control cycles, the final result will require 4 cycles, that is, one cycle more than what we had before predecessors movement.

[5]We suppose the multiplication operation will almost take an entire control cycle

In order to avoid that problem, it is possible to apply an optimization introducing a new *artificial code statement* which breaks the variables liveness. We can, in fact, introduce a new *phi* (or *merge*) operator which reassigns the needed values to new variables. In that way, there is no more need to move immediate predecessors.

Listing 4.7: Code representation from figure 4.9a.

```
1  LABEL bb1:
2     v1 = in1 + in2;
3     v2 = in3 + in4;
4     v3 = in5 + in6;
5     v4 = in7 + in8;
6     CJUMP condition, bb3;
7  LABEL bb2:
8     v5 = v1 * v2;
9  LABEL bb3:
10    v6 = v3 * v4;
```

Listing 4.8: PHIs optimizations.

```
1  LABEL bb1:
2     v1 = in1 + in2;
3     v2 = in3 + in4;
4     v3 = in5 + in6;
5     v4 = in7 + in8;
6     CJUMP condition, bb3;
7  LABEL bb2:
8     v5 = v1 * v2;
9  LABEL bb3:
10    v3_temp = PHI(v3:bb1, v3:bb2);
11    v4_temp = PHI(v4:bb1, v4:bb2);
12    v6 = v1_temp * v2_temp;
```

For instance, the listing 4.7 shows a code representation of the situation we have discussed so far. As we have already noted, the liveness of input variables for the two multiplications overlap.

Without actually moving immediate predecessors, we can just *break* the input variables lifetime making a copy of their value.

In that way, as we can see in the listing 4.8, the new input variables for the $v6$ multiplication are $v3\_temp$ and $v4\_temp$. Therefore, two multiplications inputs are no more in conflict and they can be bound on the same registers obtain a favourable situation for module sharing.

Chapter 5

Experimental Results

This chapter will show an overview of the experimental results obtaining implemented the methodology described in the previous chapter.

In particular, the first section will clarify the experimental infrastructure in which the results have been obtained.

Subsequently, the successive part will present in tabular form the most relevant data with some brief comments.

5.1 Experimental Setup

5.1.1 HLS Tool

The high-level synthesis process has been conducted by mean of the usage of Bambu software, the HLS tool provided by Panda [7] open source framework developed at the Politecnico di Milano.

Bambu tool is able to produce a Verilog description starting from a high-level representation written in C languages. Successively the synthesis process toward the implementation on specific board is completed by Vivado tool [15].

In particular, the following tools versions have been used:

- Panda 0.9.5;
- Vivado v2017.2 (64-bit).

5.1.2 Front-end Tool

In order to parse the input C source code, Bambu exploits a compiler-based interface to interact with the GNU Compiler Collection (GCC). In the exper-

imental setup, the compiler version 4.9 has been used in order to produce the results.

5.1.3 Target Device

The final target device chosen is the FPGA board *Xilinx Z-7020* which is the board used in the "quality of result" section in the Bambu publications.

5.1.4 Bambu Options

In the following lines, the compiling and synthesis options used in the experimental phase are listed.

- `-compiler=I386_GCC49`
  Force the selection of the proper *GCC* version.
- `-evaluation`
  Automatically start the RTL-synthesis and simulation of produced Verilog.
- `-fwhole-program`
  It is a GCC-like option which enables optimizations for the entire source code.
- `-fno-delete-null-pointer-checks`
- `-no-iob`
- `-aligned-access`
- `-clock-period=15`
  Specify the period of the clock signal
- `-speculative-sdc-scheduling`
  It enables the speculative sdc scheduling.
- `-device=xc7z020-1clg484-VVD`
  It selects the proper target device.
- `-print-dot`
  It prints internal representations, e.g., Control or Data Flow Graph.
- `-v4`
  It increases the output verbosity.

5.1.5 Benchmark Sources

The *benchmark* suite which has been chosen in the experimental setup is *CHStone* [16]. The CHStone benchmark suite selected programs of various application domains.

5.1.6 Experimental Environment

Evaluation and synthesis processes have been run on a *Linux* (64 bit) environment with a processor Intel(R) Xeon(R) CPU E5-2620 v2 @ 2.10GHz (24 cores).

5.2 Obtained Results

As we have seen in chapter 4, the balancing code motion algorithm work starting from a cluster of common operations. Indeed, the cluster is populated by those "*critical*" operations which the algorithm will find to balance and increase the possibility of resource sharing.

In this experimental setup, the grouping is done simply analysing the operation type and selecting multiplication expressions.

As we are going to see in the final chapter of this thesis, the proposed methodology may be applied even in a different context with different types of operations.

Despite that, the experimental results have been conducted only on a single type of operation, that is, the multiplication. However, it can still be a preliminary approach in order to demonstrate the effectiveness of the proposed methodology in certain contexts.

Obviously, as we will see, that choice can affect the goodness of the results because of some benchmarks do not present a favourable context in terms of the number of multiplications.

5.2.1 Multiplication Distribution

In this section, we are going to see how the distribution of multiplications operations among basic blocks changes before and after the balancing code motion.

Table 5.1 shows the number of multiplication in each basic block before and after the balancing algorithm in the *ADPCM* benchmark.

As we can expect, the multiplications distribution appears to be effectively balanced after the balancing code motion. Figure 5.1 plots a graphical representation.

In particular, we can compute the *standard deviation* as a balancing index.

Before the balancing process we obtain the following value:

$$\sigma_{before} = 1.35.$$

Instead, after the balancing process we obtain:

$$\sigma_{after} = 0.58.$$

BasicBlock ID	No. Mult.s Before Balancing	No. Mult.s After Balancing
8	4	1
9	2	2
10	0	1
11	0	1
12	1	1
13	1	1
14	0	1
20	2	2
27	2	2
29	1	1
46	1	1
47	2	1
58	0	1
60	1	1
78	6	3
79	1	2
80	2	2
88	1	1
92	2	2
106	1	2
107	2	2
115	1	1
119	0	1
133	2	2

Table 5.1: Multiplication Distribution among Basic Blocks *before* and *after* balancing code motion.

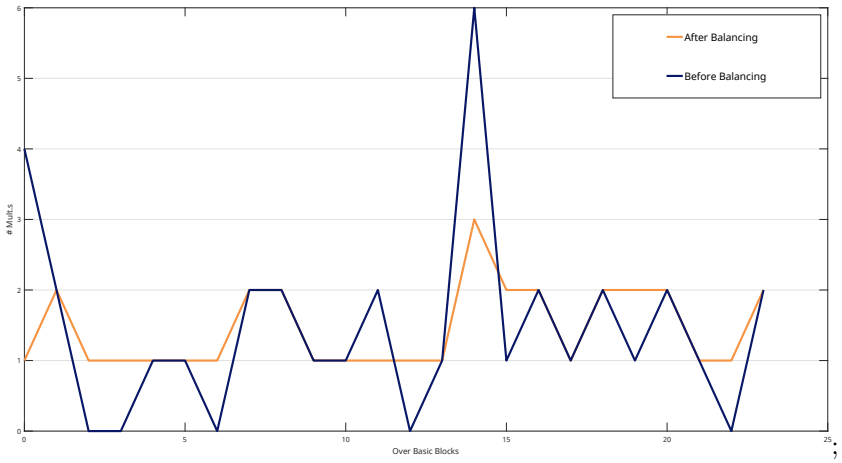


Figure 5.1: Multiplication Distribution over Basic Blocks.

That value is a measure which quantifies the amount of variation or dispersion of the multiplication among basic blocks. Indeed, a lower value describes a more balanced distribution (less variation).

5.2.2 Bambu Golden Reference

The following table shows the current quality of results published on the panda website. All information is accessible from here: `https://panda.dei.polimi.it/?page_id=678`.

Benchmark	No. Cycles	LUTs	Slices	Reg.	DSPs	BRAMs	Freq.	Slack
adpcm	15202	8925	2893	5611	60	10	77.39	2.08
dfdiv	1815	2608	870	1743	18	0	77.69	2.13
dfsin	44883	11355	3405	5464	41	0	69.25	0.56
gsm	2303	4146	1289	2572	39	3	69.98	0.71

Table 5.2: CHStone results from Bambu version 0.95.

5.2.3 Benchmarks with Balancing Code Motion

In the following table are reported the relevant output produced by the evaluation applying the methodology proposed in this thesis.

Benchmark	No. Cycles	LUTs	Slices	Reg.	DSPs	BRAMs	Freq.	Slack
adpcm	15200	8437	2752	5664	*36*	11	69.88	0.69
dfdiv	1815	2614	887	1743	18	0	77.32	2.07
dfsin	44883	11373	3458	5406	41	0	67.27	0.14
gsm	2303	4255	1436	2671	*30*	3	71.04	0.92

Table 5.3: CHStone results from Bambu applying the balancing code motion methodology.

5.2.4 Resource Allocation Details

The following tables show some comparisons about the number of module allocated by the high-synthesis process. The second column is the number of modules instantiated without the balancing code motion proposed in this thesis. Instead, the third column shows the number applying the proposed methodology.

Resource Type	Without Balancing	With Balancing
MUX_GATE	267	326
Registers_SE	379	384
Registers_STD	80	75
mult_expr_FU	35	22
widen_mult_expr_FU	6	6

Table 5.4: Number of modules allocated in *ADPCM* benchmark with and without Balancing.

Resource Type	Without Balancing	With Balancing
MUX_GATE	46	47
Registers_SE	74	80
Registers_STD	18	18
mult_expr_FU	0	0
widen_mult_expr_FU	6	6

Table 5.5: Number of modules allocated in *DFDIV* benchmark with and without Balancing.

Resource Type	Without Balancing	With Balancing
MUX_GATE	165	168
Registers_SE	256	260
Registers_STD	110	110
mult_expr_FU	1	1
widen_mult_expr_FU	14	14

Table 5.6: Number of modules allocated in *DFSIN* benchmark with and without Balancing.

Resource Type	Without Balancing	With Balancing
MUX_GATE	158	171
Registers_SE	149	148
Registers_STD	80	78
mult_expr_FU	7	6
widen_mult_expr_FU	35	36

Table 5.7: Number of modules allocated in *GSM* benchmark with and without Balancing.

5.3 Results Consideration

As we have just seen in the previous pages, the proposed methodology can effectively reduce the amount of needed resources without having a too negative effect on the latency performances.

In particular, the conducted experiments have been focused only on the reduction of multipliers modules, increasing the sharing of different multiplication operations on the same functional unit.

The reason for targeting multipliers modules arises from the fact that those functional units can significantly affect the overall performance of the circuit both in terms of delay and occupied area. Indeed, multipliers have a relatively huge impact on the timing performance because of their latency. Moreover, a multiplier is usually implemented using DSP resources which usually are less present in a FPGA architecture.

The possibility of using less critical resources can bring positive effects even in the RTL process, simplifying, for example, the placing phase of modules.

Therefore, the balancing of multipliers represents an optimal candidate in order to demonstrate the effectiveness of the proposed methodology.

As we can imagine, the movement and the postponing of multiplications operations can lead to a degradation of latency performance because of the creation of new *critical paths* on the circuit. The balancing code motion, in fact, is designed in order to handle this problem and mitigate it, focusing on the resource sharing.

In conclusion, the experimental results can directly show us the increment of sharing in a real context, because of the reduction of *DSP* resources usage as result of the evaluation process (RTL synthesis).

5.3.1 Balancing Aspects

As we have seen in the previous section (5.2.1), the balancing code motion strategy can effectively distribute the operations among basic blocks in a more uniform way.

That situation is surely more in favour in order to obtain module sharing respect to an unbalanced situation in which various multiplications are executed in parallel in the same basic block.

Moreover, often the motion of multiplications operations does not affect the latency performances because they will still be scheduled in an existent control cycle without increment the final number of cycles. In other words, the multiplications can be balanced without the creation of further critical paths.

However, the effectiveness of balancing strongly depends on the mobilities of operations computed starting from previous code motion and speculations

stages. Since the proposed methodology aims to be as more conservative as possible in order to avoid the degradation of latency performances, some balancing solutions could not be considered by the heuristic model.

5.3.2 Sharing Aspects

The results produced in the experimental section shows us the effective increment of the number of multipliers module shared for different operations.

However, the proposed methodology does not always produce an increment of area performances. In some presented benchmarks, in fact, the number of *DPS resources* does not change.

As we have already briefly explained, the balancing code motion implemented exploits a conservative strategy starting from speculations done in previous steps during the synthesis process. This strong dependence can *limit* the code *mobility* computed in the algorithm for multiplications operations and, thus, the possibility of moving the operations themselves.

However, this conservative approach allows avoiding issues in the timing aspects.

Indeed, for those benchmarks in which the balancing algorithm does not improve, the results are practically the what has been shown in the Bambu software without the implementation of balancing code motion.

Chapter 6

Conclusions and Future Works

In this chapter, we briefly summarize the work presented in this thesis, commenting the experimental results given in the previous chapter and proposing some future developments.

6.1 Main Pros in the Methodology

6.1.1 Low Latency Impact

As we have already discussed, the moving of multiplication statement could significantly impact on the latency performances. Generally, as in other contexts, high-level synthesis process is very subjected to a trade-off between latency and area performances.

During the design process, in fact, often one aspect is fixed as constraint and tools try to minimize the other one.

The methodology proposed in this thesis work has a low impact level on the latency performances. Indeed, even if some multiplications are postponed, they will be still moved into an existent[1] basic block without creating another one. This approach minimizes the possibility to create new critical paths.

Finally, the conclusive approach in the methodology allows extending the analysis to predecessors of multiplications. The optimizations produced in the last section of chapter 4 provide an effective approach in order to solve the problem of variable lifetime overlaps.

[1] After code motions and speculations.

6.1.2 Time Complexity Considerations

The initial data acquisition phase is performed considering all statements in the input body function. If we assume n the number of all statements, the asymptotic time complexity grows linearly, that is, $\mathcal{O}(n)$.

Moreover the analysis of *dominance* relationship among basic blocks, it can be simply done visiting the dominator tree. Assuming m the number of basic blocks, the time complexity is always linear: $\mathcal{O}(m)$.

The analysis of successors and predecessors in data dependency context is still linear in the number of statements, we still obtain $\mathcal{O}(n)$.

On the other hands, the balancing algorithm is performed for each statement in the extended cluster ($|S_k^E| = q$). The reducing mobility strategy can take upon $\mathcal{O}(q \cdot m)$ for a total of $\mathcal{O}(m \cdot q^2)$ which is the most critical section.

The algorithms proposed essentially run on graph representations and they simply scan them *linearly*. Moreover, it is very likely that the number of analysed basic blocks is quite small respect the number of statements.

6.2 Drawbacks in the Methodology

6.2.1 Strong Dependency

As already mentioned, the methodology is characterized by a strong *dependence* from the speculations and code motion made in previous steps in the synthesis process.

Although that dependence provides a *conservative* approach which avoids performances degradation, it can also introduce some limits in the balancing code motion. For that reason, sometimes the solutions space could not include the *optimal* solution in term of *uniformity* of statements distribution.

In other words, the possibility of re-arrange "*heavy*" operations depends on whether those operations have been moved in previous steps or not.

6.2.2 Highly Dedicated Approach

The methodology has been designed and tested with a *single* type of operation (i.e., *multiplications*).

In the previous chapter, it has been already explained the reason of that. Anyway, as shown in a particular benchmark, the lack or the minimum availability of such type of operations can compromise, obviously, the final improvement obtained.

Despite that, as we are going to see in few lines later, the methodology may be easily extended in order to operate with more different clusters of operations.

6.3 Future Developments

6.3.1 Aggressive Code Motion Balancing

A possible further approach would be to reduce the dependence of the proposed balancing algorithm from previous code motions steps. In other words, it could be possible to design a more "*aggressive*" heuristic which computes the mobility of a statement without taking into account the position of that statement before of code speculations.

Actually, this kind of strategy has been already proposed in this work even if in a more specific context.

In fact, in order to break the liveness overlapping of input variables, we have computed what this work calls *extended mobility* (see section 4.2.5).

Therefore, this specific approach may be extended in the overall heuristic algorithm probably increasing the mobility assigned to those statements in the cluster which have to be balanced.

As always, this approach represents a kind of *trade-off* between area and latency performances. Indeed, the mobilities increment can bring to select some solutions which degrade the overall latency of the circuit.

A possible strategy, as seen in different cases, is to let the designer pick the more suitable strategy in accordance with his/her demands. Alternatively, an automatic algorithm could execute both strategies and then compute the best solution which minimizes the overall performances.

6.3.2 Cyclic Code Motion on Operation Types

So far the methodology approach has been applied to a *single* specific type of operations (i.e., multiplications). A possible alternative strategy could be to apply *repeatedly* the balancing code motion to *different types* of operations.

The idea would be to start from a *more critical* resource (e.g. multiplications) and proceed towards less critical operations. Indeed, the proposed methodology already presents the concept of operations *cluster* (see section 4.2.3).

Once all non-trivial statements have been grouped in different clusters, the algorithm will be first applied to the more relevant (in terms of possible area occupied) set. Successively, all operations balanced have to be *fixed and constrained* in the next iteration where a different, and less relevant, set of operations will be balanced.

6.3.3 Memory Operations

A small consideration can be done about *memory* operations. Those operations can be very problematic in term of efficiency. Indeed, in some target architecture, the memory is a shared module which cannot behave easily in a *parallel way*. In other words, operations of storing data in the memory cannot be executed in aggressive parallel fashion.

In this regard, the balancing code motion strategy can represent a valid solution in these terms. The distribution of memory access in a more uniform way can improve the overall latency of the circuit. Therefore, the proposed methodology could be easily applied in that context and obtain better result even in terms of latency.

List of Figures

Bibliography

[1] Xilinx Inc. Vivado design suite user guide: Partial reconfiguration. `http://www.xilinx.com`, 2015.

[2] Andrew W. Appel. *Modern Compiler Implementation in C: Basic Techniques*. Cambridge University Press, New York, NY, USA, 1997.

[3] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

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

[5] Leon Stok. Data path synthesis. 18:1–71, 12 1994.

[6] D. D. Gajski and L. Ramachandran. Introduction to high-level synthesis. *IEEE Design Test of Computers*, 11(4):44–54, Winter 1994.

[7] Politecnico di Milano. Panda framework. `http://panda.dei.polimi.it`, 2014.

[8] D. C. Zaretsky, G. Mittal, R. P. Dick, and P. Banerjee. Balanced scheduling and operation chaining in high-level synthesis for fpga designs. In *8th International Symposium on Quality Electronic Design (ISQED'07)*, pages 595–601, March 2007.

[9] J. Cong and Zhiru Zhang. An efficient and versatile scheduling algorithm based on sdc formulation. In *2006 43rd ACM/IEEE Design Automation Conference*, pages 433–438, July 2006.

[10] Marco Lattuada and Fabrizio Ferrandi. Code transformations based on speculative sdc scheduling. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, ICCAD '15, pages 71–77, Piscataway, NJ, USA, 2015. IEEE Press.

[11] Jason Cong and Wei Jiang. Pattern-based behavior synthesis for fpga resource reduction. In *Proceedings of the 16th International ACM/SIGDA Symposium on Field Programmable Gate Arrays*, FPGA '08, pages 107–116, New York, NY, USA, 2008. ACM.

[12] Hao Cong, Song Chen, and T. Yoshimura. Port assignment for interconnect reduction in high-level synthesis. In *Proceedings of Technical Program of 2012 VLSI Design, Automation and Test*, pages 1–4, April 2012.

[13] Dake Liu and C. Svensson. Power consumption estimation in cmos vlsi chips. *IEEE Journal of Solid-State Circuits*, 29(6):663–670, Jun 1994.

[14] D. Chen and J. Cong. Register binding and port assignment for multiplexer optimization. In *ASP-DAC 2004: Asia and South Pacific Design Automation Conference 2004 (IEEE Cat. No.04EX753)*, pages 68–73, Jan 2004.

[15] Xilinx Inc. Vivado. `http://www.xilinx.com`.

[16] Chstone benchmark suite. `http://www.ertl.jp/chstone/`.