Chapter 6

OPTIMIZATION FOR RECONFIGURABLE SYSTEMS USING HIERARCHICAL ABSTRACTION

Elaheh Bozorgzadeh
UCLA Computer Science Department
elib@cs.ucla.edu

Adam Kaplan
UCLA Computer Science Department
kaplan@cs.ucla.edu

Ryan Kastner
UCLA Computer Science Department
kastner@cs.ucla.edu

Seda Ogrenci Memik
UCLA Computer Science Department
seda@cs.ucla.edu

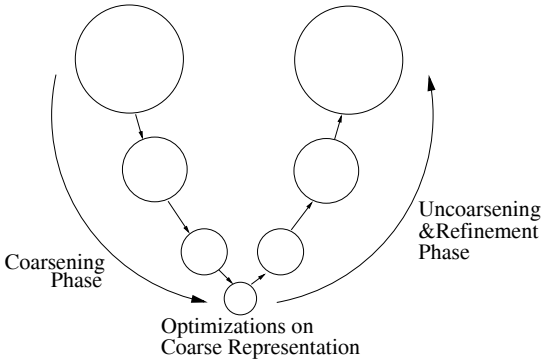
Majid Sarrafzadeh
UCLA Computer Science Department
majid@cs.ucla.edu

1. Introduction

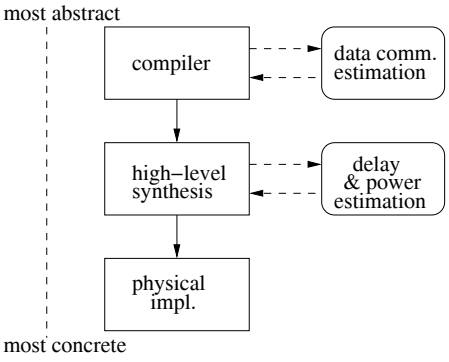
In the previous chapters, we have seen various multilevel optimization techniques used to solve a variety of complex problems. In this chapter we discuss techniques to synthesize VLSI systems from a high level description. From system genesis as a high-level description to its final physical layout, the problem of hardware synthesis is too nebulous to be handled in a single stage optimization process. As a result, CAD flow is divided into three major stages: Compiler Optimization, High-level Synthesis, and Physical Design. The traditional flow of the stages departs from the exact concept of multilevel optimization as addressed in this book. Therefore, in this chapter we focus on another way to abstract complexity in VLSI systems: *hierarchical abstraction* .

Hierarchical abstraction is a top-down, modular breakdown of the problem, where each stage represents a different level of abstraction with specific optimization objectives. For clarity, the generic concept of multilevel optimization is illustrated in Figure 6.1(a), along with two different realizations of hierarchical abstraction in Figure 6.1(b) and (c). The difference between the two hierarchical flows in the figure is in the way the interaction between the stages is realized. In Figure 6.1(b), the interaction is applied through modeling and estimation of parameters which physically manifest themselves in later stages. Estimation engines in each stage are shown in the figure as well. There has been extensive research in high-level synthesis that considers the effect of early optimization techniques on subsequent stages. Such techniques either utilize the early estimation or use forward-looking objective functions [Wong et al., 2002]. The estimation parameter could be routability/area [Xu and Kurdahi, 1997; Dougherty and Thomas, 2000], power [Macii et al., 1998; Khouri et al., 1998], etc. In Figure 6.1(c), the interaction is realized via feedback and cross-level communication between the stages [Park et al., 1999; Bringmann and Rosenstiel, 1997]. The formulation of the whole VLSICAD flow in the multilevel optimization framework is still an open problem. In this chapter, we specifically address the synthesis flow for reconfigurable systems.

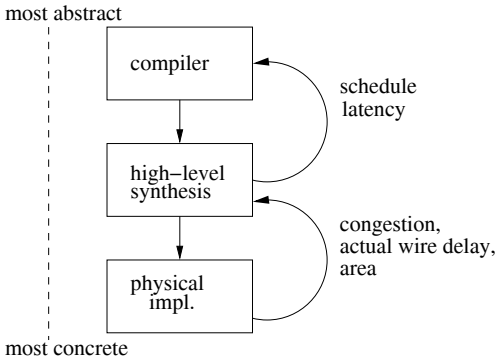
Programmability and reconfigurability are essential ingredients in emerging embedded systems. Reconfiguration can be realized in different forms. Field Programmable Gate Arrays (FPGAs) are one of the most popular reconfigurable devices. FPGAs consist of programmable building blocks called logic blocks or Configurable Logic Blocks (CLBs), which are placed on the FPGA chip either on a two-dimensional array or in rows. Between each row and column routing channels are located. The interconnect architecture is also programmable. This is enabled through



a) Generic Representation of
Multilevel Optimization



b) Traditional Hierarchical Abstraction



c) Hierarchical Abstraction With
Cross−level Interaction

Figure 6.1. Different VLSICAD Flow Methodologies

switches located between wire segments along the routing channels. A switch is essentially a transistor that can take on or off states, hence enabling a connection or an open circuit between two wire segments. In [Hauck, 1998; Brown et al., 1992] more information on FPGAs and other programmable architectures is available.

New embedded systems require a larger amount of flexibility as compared to their predecessors. Rapidly changing markets demand embedded system solutions that can be modified easily. Reconfigurability in embedded systems provides the necessary adaptability. Devices can be reconfigured during runtime (often referred to as dynamic reconfiguration), or on a per application basis (static reconfiguration). The time of reconfiguration, along with how much of the device is reconfigured, can be managed by a reconfiguration manager (either on or off-chip) or even possibly by the synthesis tools that create circuit functionality.

The addition of reconfigurability to embedded devices introduces new and unique problems. Each stage of the development of embedded systems is affected by reconfigurability [Schaumont et al., 2001]. Throughout the design flow, we seek to make the best use of this flexible platform, such as exploiting parallelism and low-cost design customization. However, flexibility in reconfigurable systems comes at the expense of degraded design quality. For instance, in FPGAs, programmable interconnect realized through switches limits wire performance. Additionally, data communication should obey the prefabricated programmable interconnect structure in the system. In short, reconfiguration allows great design flexibility, but at a performance cost. Thus, the limitations of FPGA technology must be well understood and modeled so that we can push reconfigurable systems to their highest potential performance.

Current models of computation (MOCs) are based on traditional Von Neumann architectures, and as such they lack the ability to model the emerging reconfigurable world of computing. Traditional optimization techniques cannot exploit reconfigurability to its full potential. Hence, the entire design flow must be revisited and revised from this new perspective.

Due to the complexity of the overall system design, the global optimization problem is divided into multiple optimization problems at different levels of abstraction. At each stage of design, different optimization techniques must be applied. Integrating reconfigurability into system design requires us to incorporate reconfiguration-centric optimization techniques into this hierarchical framework.

For a given reconfigurable platform, the overall goal of the design flow is to map an application (specified in a high level programming language

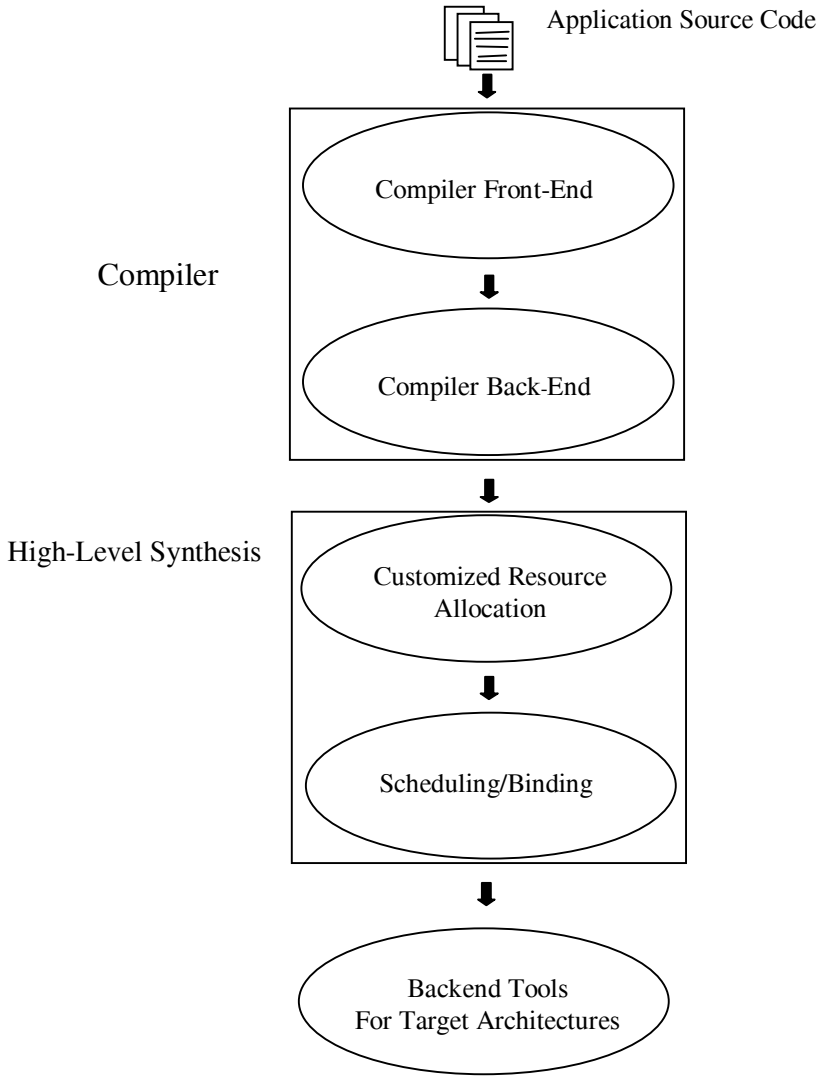


Figure 6.2. **Overall Design Flow for Reconfigurable Computing Systems.**

such as C or Fortran) to hardware. Each of the levels of system design for a representative FPGA-like platform is illustrated in Figure 6.2.

There are three major steps in mapping an application onto hardware: the compiler stage, the high-level synthesis stage, and back-end (logical/physical) implementation stage. Within the compiler stage, the compiler front-end translates the high-level source code of an application into an intermediate representation (IR). The hardware compiler back-end (translation of this IR into a hardware description) is where hardware-aware optimizations need to be applied. Data communication minimization is one of the essential optimization problems at this stage. The compiler back-end then sends a control dataflow graph (CDFG) to the high-level synthesis stage. Customized resource allocation provides the set of resources to be used by subsequent scheduling and binding tasks. In this stage optimization techniques can be utilized to find the set of resources best suited to the needs of the specific input CDFG. This customized resource set includes various implementations of functional modules, such as IP blocks and soft/hard macros. During scheduling and binding, operations are assigned to control steps and bound to available resources obeying the resource constraints. At this stage efficient utilization of customized resources while minimizing the overall latency of the schedule is an important optimization objective. Finally, a back-end implementation stage maps the application on the target architecture. This stage is highly architecture dependent and is mostly provided by device vendors.

In the following, three optimization problems at three different levels of the design flow will be presented in detail. We provide formulations for each individual optimization problem. Based on theoretical analysis on the hardness of these problems, we identify NP-Complete problems and optimally solvable problems. We propose efficient heuristics and optimal algorithms for respective problems.

The organization of the rest of this chapter is as follows: in Section 2, data communication minimization problem on a given CDFG at the compiler level is presented. In Sections 3 and 4 we describe two optimization problems at the high-level synthesis stage; customized resource allocation and simultaneous scheduling and binding driven by efficient trade-off between utilization of customized resources and exploitation of available parallelism. Conclusions are given in Section 5.

2. Compilation: Data Communication Minimization

Compilation can be generalized as a process that translates some form of high-level program specification (e.g. an accounting application written in C++) into an equivalent lower-level form (e.g. an executable binary program). The high-level specification is usually referred to as the source code, and the lower-level form (or the machine architecture that it executes on) is referred to as the target. A typical compiler is divided into a front-end segment and a back-end segment. The front-end parses the high-level source code and transforms it into an intermediate representation (IR). The back-end typically takes this IR and targets it toward a particular architecture (such as an Intel x86 or a MIPS processor). Traditionally, a compiler is a *software compiler*, meaning that its back-end produces some low-level code to be executed upon a target processor. In this work, we refer to *hardware compilers*, whose back-ends produce a hardware description which can subsequently be synthesized into actual hardware. In hardware compilation, the input program specification is used as a recipe for the gate-level logic that we wish to synthesize. Although we use the term hardware compiler to represent a compiler that builds hardware, the compiler itself is a software product.

At the compiler back-end stage of synthesis, we focus on the problem of minimizing data communication between modules in the resulting floorplan. We define data communication as a collective abstraction comprised of two components: a) reads and writes from RAM and b) interconnect required between hardware modules. Therefore we say that the total data communication in a circuit is the addition of data communication resulting from RAM accesses with the data communication resulting from wires between modules in the floorplan. Minimizing communication can benefit the final design by reducing memory access delay (by reducing the number of RAM operations) and reducing interconnect (and thus increasing wire performance). However, this objective must be achieved without concrete foreknowledge of the modules that will be synthesized, for the generation and placement of these modules will occur after compilation. Essentially, we wish for the compiler to make informed decisions about future modules via a software-level model of the inter-module communication. Such a model can be realized in a control dataflow graph (CDFG), which can easily be synthesized from the compiler's IR. In this discussion we reintroduce Static Single Assignment (SSA), a classic compiler technique, and demonstrate its effectiveness in the realm of optimizing hardware compilation. Specifically, we will show

that performing SSA on a program in CDFG form will reduce data communication in the resulting hardware description. Furthermore, we show that SSA in its original form is not optimal in terms of data communication and give an optimal algorithmic extension to minimize the amount of data communication. In the following subsection, we present a formal definition of the data communication minimization problem. Then we present our modified SSA algorithm, and prove that it is both a correct and minimal solution to the data communication problem.

2.1 Problem Definition

2.1.1 Control Data Flow Graphs. We focus on the control dataflow graph (CDFG) as a model of computation (MOC) for the IR of our compiler. The CDFG offers several advantages over other models of computation. Most compilers have an IR that can easily be transformed into a CDFG. Therefore, this allows us to use the back-end of a compiler to generate code for a variety of processors. Furthermore, the techniques of data flow analysis (e.g. reaching definitions, live variables, constant propagation, etc.) can be applied directly to CDFGs. Additionally, many high-level programming languages (Fortran, C/C++) can be compiled into CDFGs with slight modifications to pre-existing compilers: a pass converting a typical high-level IR into control flow graphs and subsequently CDFGs is possible with minimal modification. Most importantly, we believe that the CDFG can be mapped to a variety of different microarchitectures. All of these reasons make the CDFG a good MOC for investigating the performance of mapping different parts of the application across a wide variety of SOC components. A CDFG consists of a set of control nodes N_{cfg} and control edges E_{cfg}. The *control nodes* are a set of basic blocks. Each control node holds a number of instructions or computations that execute atomically. The *control edges* model the control flow relationships between the control nodes. The control nodes and control edges form a directed graph $G_{cfg}(N_{cfg}, E_{cfg})$. Each control node contains a set of operations. The data flow relationships between the operations in a particular control node can be viewed as a sequential list of instructions I or a data flow graph $G_{dfg}(V_{dfg}, E_{dfg})$. The conversion from I to G_{dfg} , and vice-versa, is trivial. Data edges between any pair of control nodes represent general data communication ; meaning that a piece of data is used by both nodes. As mentioned earlier, the data may be communicated directly via a wire, or written to and read from RAM. An example CDFG is shown in Figure 6.3.

2.1.2 Minimization of Data Communication. In this section, we examine the problem of mapping an application onto a micro-

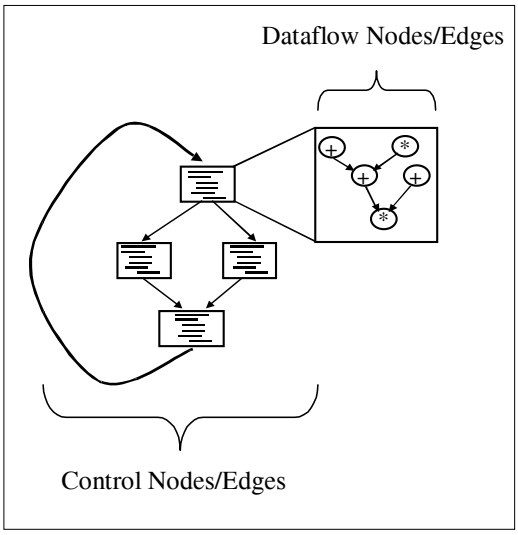


Figure 6.3. A Control Data Flow Graph

architecture such that inter-module data communication is minimized. Each control flow node (or basic block) of a CDFG can be thought of as an abstraction for real modules that will be placed in the floorplan after compilation. A control flow node consists of a set of inputs and a set of outputs. After the computations are completed, control is transferred to another control flow node. We must add a mechanism to direct the control flow, i.e., a controller. The actual mapping of control nodes and controllers to hardware modules can be realized in two different schemes: a distributed control scheme and a centralized control scheme.

Distributed control has several different entities that control the path of execution. Each control node has a local controller that determines the next control node in the execution sequence. Therefore, there are direct connections between control nodes. Every control node is equipped with an *execute* port that tells it when to begin execution. Additionally, each control node has a set of *control flow indicator (CFI)* ports. There is a CFI port for each of the different control nodes that may follow this node in execution. Equivalently, there is a CFI for each control edge emanating from a control node. A CFI port connects to the execute port of other control nodes.

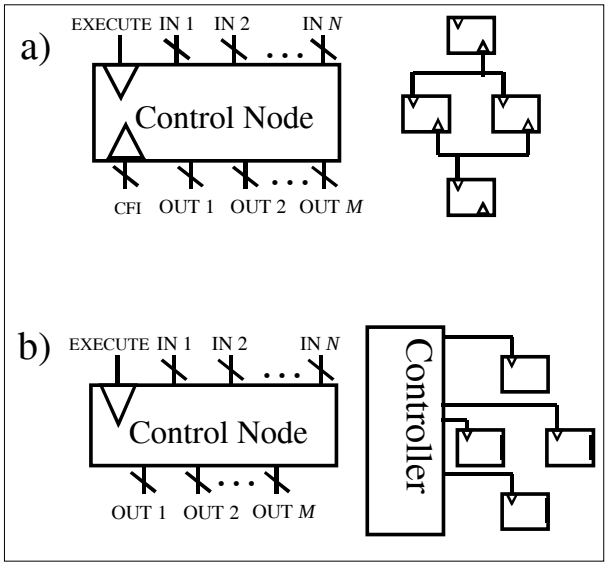


Figure 6.4. a) Distributed Control b) Centralized Control

Figure 6.4 a) illustrates a simple example of distributed control. Centralized control has one controller that determines the control node or nodes that execute at any given instant. As with distributed control, each control node has an execute port that initiates the execution of the data flow graph embedded in the control node. Unlike distributed control, every execute port of a control node is connected to the controller. Centralized control closely resembles the separation of control flow and data flow assumed by most high-level synthesis engines. Figure 6.4 b) gives an example of centralized control.

In order to determine the data exchange between nodes in a CDFG, we establish the relationship between where data is generated and where data is used for calculation. The specific place where data is generated is called its *definition point*. A specific place where data is used in computation is called a *use point*. The data generated at a particular definition point may be used in multiple places. Likewise, a particular use point may correspond to a number of different definition points; the control flow dictates the actual definition point at any particular moment. If data generated in one control node is used in a computation in a second control node, these two control nodes must have a mechanism to transfer

the data between them. A distributed data communication scheme has a direct connection between the two control nodes (i.e. one node controls the other's execution through a signal). If a centralized data communication scheme were used, the first control node would transfer the data to memory and the second control node would access the memory for that data. Therefore, in a centralized scheme, minimizing the inter-node communication has a direct impact on the number of memory accesses, and in a distributed scheme, the interconnect between the control nodes is reduced. In practice, many hybrid models exist, in which some of the data is communicated directly between nodes, and other data is written to RAM. Nevertheless, in each control scheme real performance boosts can be realized through communication optimization. Thus, regardless of the scheme that we use, we should try to minimize the amount of inter-node data communication . Therefore, our problem is the following: given a CDFG representation of an application, we wish to perform a transformation on this graph that will result in minimal data communication (data edges) between nodes. This transformation must be performed in a correct manner, that is: it must maintain the correctness of the application being compiled.

2.2 Algorithm

In this section we provide an efficient and optimal algorithm for minimizing data communication in a CDFG, via a transformation called Static Single Assignment. Our pictorial representation of the transformed CDFG will subsequently include data edges between control nodes (in addition to the control edges of the original CDFG), emphasizing the potential reduction in data communication.

2.2.1 Static Single Assignment. We can determine the relationship between the use and definition points through static single assignment [Cytron et al., 1989; Briggs et al., 1998]. Static Single Assignment (SSA) renames variables with multiple definitions into distinct variables — one for each definition point. Traditionally, a variable is a named symbol that represents a storage location, abstracting the underlying storage mechanism for values. In typical imperative source code, a variable may take on several values over its lifetime (for instance, an incremental loop counter variable is assigned its original value plus a constant every time the loop is iterated). We define a *name* as a symbol (usually a character string) that represents the contents of a storage location (e.g. register, memory). A name is unspecific to SSA. In non-SSA code, a name represents a storage location but we may not know the exact location; the precise location of the name depends on the control flow

of the program. We call a name in non-SSA code a *location* (sometimes referred to as l-value), as it abstracts the potential storage location for a set of values. SSA eliminates this confusion, as each name represents a value that is generated at exactly one definition point. Therefore, there is a mapping from every name to a single associated value. This is intuitively beneficial for hardware synthesis, as we wish for every name to specifically represent a single signal traveling along a wire. Therefore, we say that SSA maps every name to a single value, by allowing only one single assignment to each name in the program. Of course, in the presence of control constructs such as loops or branches, a name might have to represent different values depending on the control path that a program takes during its execution. In order to maintain proper program functionality, we must add ϕ-nodes into the CDFG. ϕ-nodes are needed when a particular use of a name is defined at multiple points. A ϕ-node is essentially a selector that takes a set of possible values and selects the particular one that corresponds to the execution path that has been taken. ϕ-nodes can be viewed as an operation of the control node. They can be implemented using a multiplexer.

Figure 6.5 illustrates the conversion to SSA. SSA is accomplished in two steps: first we add ϕ-nodes and then we rename the variables at their definition and use points. There are several methods for determining the location of the ϕ-nodes. The naive algorithm would insert a ϕ-node at each merging point for each original name used in the CDFG. A more intelligent algorithm - called the minimal algorithm - inserts a ϕ-node at the *iterated dominance frontier* of each original name [Cytron et al., 1989]. The iterated dominance frontier is the set of nodes in the timeline of the program where two (or more) values of a variable merge. The semi-pruned algorithm builds a smaller SSA form than the minimal algorithm, in that it creates fewer ϕ-nodes. It determines if a variable is local to a basic block and only inserts ϕ-nodes for non-local variables [Briggs et al., 1998]. The pruned algorithm further reduces the number of ϕ-nodes by only inserting ϕ-nodes at the iterated dominance frontier of variables that are live at that time [Cytron et al., 1991]. After the position of the ϕ-nodes is determined, there is a pass where the variables are renamed. The minimal method requires $O(|E_{cfg}|+|N_{cfg}|^2)$ time for the calculation of the iterated dominance frontier. The iterated dominance frontier and *liveness analysis* must be computed during the pruned algorithm. Liveness analysis is the identification of the ranges in code (denoted *live ranges*) over which a variable is used. A variable's live range starts with its definition and culminates with the last expression in which it is used. Typically we say that a variable's live range covers a set of basic blocks (or control nodes in our representation). There are linear

or near linear time liveness analysis algorithms [Graham and Wegman, 1976; Karn and Ullman, 1991; Kennedy, 1981]. Therefore, the pruned method has the same asymptotic runtime as the minimal method.

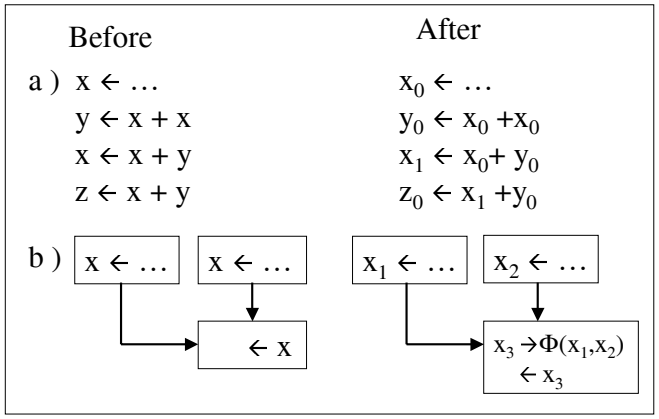


Figure 6.5. a) Conversion of Straight-line Code to SSA b) SSA Conversion with Control Flow

2.2.2 Minimizing Data Communication with SSA. SSA allows us to minimize the inter-node communication. The various algorithms used to create SSA all attempt to accurately model the actual need for data communication between the control nodes. For example, if we use the pruned algorithm for SSA, we eliminate false data communication by using liveness analysis, which eliminates passing data that will never be used again. SSA allows us to minimize the data communication, but it introduces ϕ-nodes to the graph. We must add a mechanism that handles the ϕ-nodes. This can be accomplished by adding an operation that implements the functionality of a ϕ-node. A multiplexer provides the needed functionality. The input names are the inputs to the multiplexer. An additional control line must be added for each multiplexer to determine that the correct input name is selected. A fundamental limitation of using SSA in a hardware compiler is the use of the iterated dominance frontier for determining the positioning of the ϕ-nodes. Typically, compilers use SSA for its property of a single definition point. The representation of a variable as a single value aides classical optimizations, such as dead-code identification (which eliminates pieces of code that will never be executed from the intermediate representation). We are using it in another way - as a representation to minimize the data communication between hardware components (CFG nodes). In

this case, the positioning of ϕ-nodes at the iterated dominance frontier does not always optimize the data communication. We must consider spatial properties in addition to the temporal properties of the CDFG when determining the position of the ϕ-nodes.

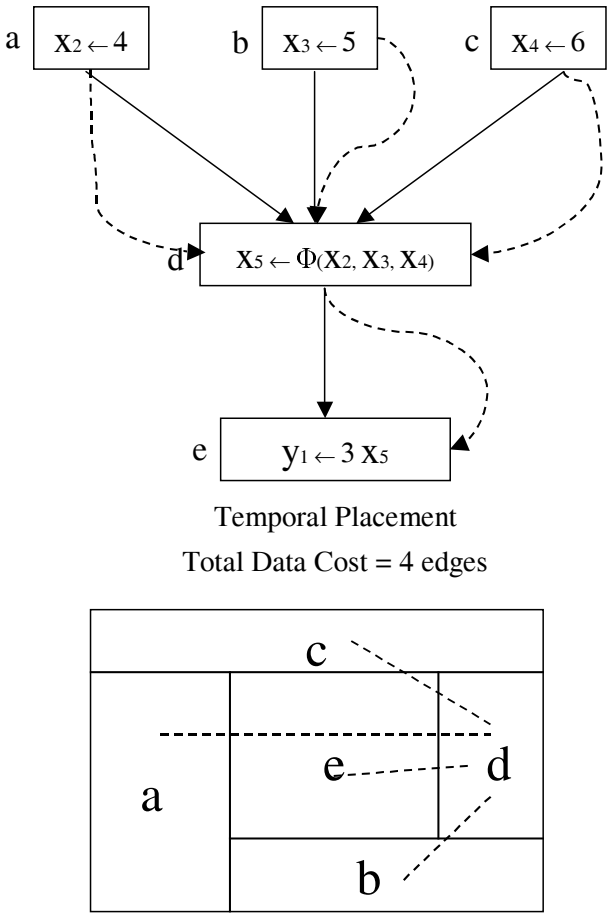


Figure 6.6. SSA form and the corresponding floorplan (dotted edges represent data communication, and solid edges represent control). Data communication = 4 units.

We illustrate our point with a simple example. Figure 6.6 exhibits traditional SSA[1] form as well as the corresponding floorplan, containing control nodes a through e. The ϕ-node is placed in control node d. In the traditional SSA scheme, the data values x_2, x_3, and x_4 (from nodes a, b, and c) are used in node d, but only in the ϕ-node. Then, the data x_5 is used in node e. Therefore, there must be a communication connection from node a to node d, node b to node d and node c to node d, as well as a connection from node d to node e - a total of 4 communication links. In Figure 6.7, the ϕ-node is distributed to node e. Then, we only need a communication connection from nodes a, b, and c to node e, a total of 3 communication links. From this example, we can see that traditional ϕ-node placement is not always optimal in terms of data communication. This arises because ϕ-nodes are traditionally placed in a temporal manner. When considering hardware compilation , we must think spatially as well as temporally. By moving the position of the ϕ-nodes, it is possible to achieve a better layout of our hardware design. In order to reduce the data communication, we must consider the number of uses of the value that a ϕ-node defines as well as the number of values that the ϕ-node takes as an input.

2.2.3 An Algorithm for Spatially Distributing ϕ-nodes.

The first step of spatially distributing ϕ-nodes is determining which ϕ-nodes should be moved. We assume that we are given the correct temporal positioning of the ϕ-nodes according to some SSA algorithm (e.g. minimal, semi-pruned, pruned). The movement of a ϕ-node depends on two factors. The first factor is the number of values that the ϕ-node must choose between. We call this the number of ϕ-node source values s. The second factor is the number of uses of the value that the ϕ-node defines. We call the defined value the ϕ-node destination value *dest*. We denote the number of blocks in which this value is used by d. Taking Figure 6.6 as an example, the ϕ-node source values are x_2, x_3, and x_4 whereas the ϕ-node destination value is x_5. Determining s is simple: we just need to count the number of source values in the ϕ-node. Finding the number of nodes in which the destination value is used is more difficult. We can use def-use chains [Muchnick, 1997], which can be calculated during SSA. The relationship between the amount of communication links C_T needed for a ϕ-node in temporal SSA and the number of communication

[1]We use the terms "traditional SSA" and "temporal SSA" interchangeably to mean the SSA introduced by Cytron et al. [Cytron et al., 1989].

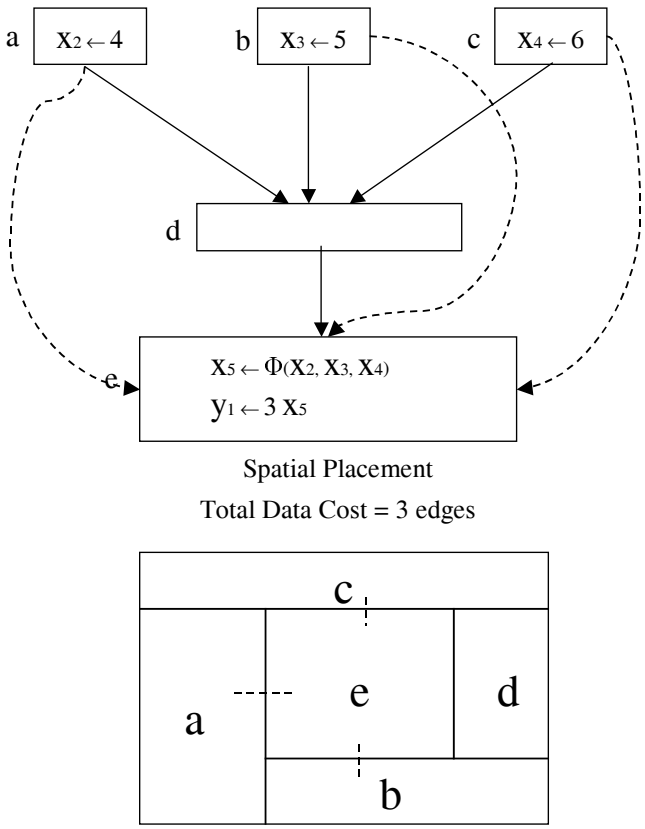


Figure 6.7. SSA form with the ϕ-node spatially distributed, as well as the corresponding floorplan. Data communication = 3 units.

links C_S in spatial SSA is:

$$C_T = s + d \tag{6.1}$$

$$C_S = s \times d \tag{6.2}$$

Using these relationships, we can easily determine if spatially moving a ϕ-node will decrease the total amount of inter-node data communication. If C_S is less than C_T, then moving the ϕ-node is beneficial. Otherwise, we should keep the ϕ-node in its current location. (It is equivalent to say that we should move the ϕ-node when there is only one use of the destination value, since C_S is only less than C_T if d is equal to 1. s must be greater than 1 or else a ϕ-node would be unnecessary.) After we have decided on which ϕ-nodes we should move, we must determine the control node(s) to which we should move the ϕ-node. This step is rather easy, as we move the ϕ-node from its original location to all control nodes that have a use of the destination value of that ϕ-node. It is possible that by moving the ϕ-node, we increase the total number of ϕ-nodes in the design. But, we are decreasing the total amount of inter-node data communication. Therefore, the amount of data communication is not directly dependent on number of ϕ-nodes.

```
SPATIAL SSA ALGORITHM (G(N_cfg , E_cfg))
1.  Perform_SSA (G)
2.  Calculate_def_use_chains(G)
3.  Remove_back_edges(G)
4.  Topological_sort(G)
5.  for each node n∈ N_cfg
6.      s ← |Φ.source|
7.      d ← |def_use_chain(Φ.dest)|
8.      if s × d < s + d
9.          move_to_spatial_locations(Φ)
10. restore_back_edges(G)
11. return G
```

Figure 6.8. Spatial SSA Algorithm

It is possible that a use point of the definition value of ϕ-node ϕ_1 is another ϕ-node ϕ_2. If we wish to move ϕ_1, we add the source values of ϕ_1

into the source values of ϕ_2; obviously, this action changes the number of source values of ϕ_2. In order to account for such changes in source values, we must consider moving the ϕ-nodes in a topologically sorted manner based on the CDFG control edges. Of course, any back control edges (resulting from loops) must be removed in order to have valid topological sorting. We can not move ϕ-nodes across back edges, as this can induce dependencies between the source value and the destination value of previous iterations i.e. we can get a situation where $b_1 \leftarrow \phi(b_1, ...)$. The source value b_1 was produced in a previous iteration by that same ϕ-node. The complete algorithm for spatially distributing ϕ-node to minimize data communication is outlined in Figure 6.8.

Theorem 1 *Given an initially correct placement of a ϕ-node, the functionality of the program remains valid after moving the ϕ-node to the basic block(s) of all the use point(s) of the ϕ-node's destination value.*

Proof: There are two cases to consider. The first case is when the use point is a normal computation. The second case is when a use point is a ϕ-node itself. We consider the former case first. When we move the ϕ-node from its initial basic block, we move it to the basic blocks of every use point of the ϕ-node's destination value *dest*. Therefore, every use of the *dest* can still choose from the same source values. Hence, if the ϕ-node source values were initially correct, the use points of *dest* remain the same after the movement. We must also ensure that moving the ϕ-node does not create some other use point that uses the same name but has a different value. The ϕ-node will not move past another ϕ-node that has the same name because by construction of correct initial SSA, that ϕ-node must have *dest* as one of its source values. The proof of the second case follows similar lines to that of the first one. The only difference is that instead of moving the initial ϕ-node ϕ_i to that basic block, we add the source values to the ϕ-node ϕ_u that uses *dest*. If we move ϕ_i before ϕ_u, then the functionality of the program is correct by the same reasoning of the first part of proof. Assuming that the temporal SSA algorithm has only one ϕ-node per basic block per name, we can add the source values of ϕ_i to ϕ_u while maintaining the correct program functionality. □

Theorem 2 *Given a correct initial placement of ϕ-nodes, the spatial SSA algorithm maintains the correct functionality of the program.*

Proof: The algorithm considers the ϕ-nodes in a topologically sorted manner. As a consequence of Theorem 1, the movement of a single ϕ-node will not disturb the functionality of the program hence the ϕ-node will not move past another value definition point with the same

name. Since we are considering the ϕ-nodes in forward topologically sorted order, the movement of any ϕ-node will never move past a ϕ-node which has yet to be considered for movement. Also, a ϕ-node can never move backwards across an edge (remember we remove back edges). Therefore, the algorithm will never move a value definition point past another value definition point with the same name. Hence every use preserves the same definition after the algorithm completes. This maintains the functionality of the program. □

Theorem 3 *Given a floorplan where all wire lengths are unit length, the Spatial SSA Algorithm provides optimal data communication.*

Proof: In this proof we will distinguish between a *ϕ function* (which is an expression that maps a set of source values to a destination value) and a *ϕ-node* (which is a control node at which a ϕ function is placed). The source values of any given ϕ function exist in individual control nodes, and the cardinality of this set of nodes is the same as s, the cardinality of the set of source values. Likewise, the use points of any ϕ function's *dest* are individual control nodes, and their cardinality is d. The number of control nodes at which a given ϕ function is placed will be referred to as n. (n may also be referred to as the cardinality of the set of ϕ-nodes for a ϕ function, by the definition above.) The amount of data communication that this algorithm can reduce is restricted to the number of data edges coming into each ϕ-node from its sources and the number of data edges connecting each ϕ-node to its uses. (The other data communication is already minimized, since SSA variables are actual data values. Therefore, SSA variables passed between control blocks are actual pieces of data that must be moved.) If a ϕ-node is coalesced with its use point, then the number of out degree edges leaving the ϕ-node for those use points is equal to zero (since the ϕ function is in the same node with its uses). The total number of data communication points entering and exiting the ϕ-nodes of a given ϕ function can be represented by a cost equation:

$$C = \sum_{i=1}^{n} (in_i + out_i) \tag{6.3}$$

where *in* is the number of inbound data edges to each ϕ-node from source values and *out* is the number of outbound data edges from each ϕ-node to uses of the destination value. In a floorplan where each edge has unit cost, this equation represents the total cost of this ϕ function in the graph. In order to maintain correctness in a CDFG, every source value of a ϕ function must be coming into all ϕ nodes defining this function.

(This is the only data that needs to enter a ϕ-node.) Therefore, for all minimal cost cases, we can say that $in = s$ for every ϕ node and the data communication cost of the ϕ function can be restated as

$$C = n \times s + \sum_{i=1}^{n} (out_i) \tag{6.4}$$

since s is constant. This leaves us with two values we can minimize: n (the number of total nodes defining a given ϕ function) and out (the number of data edges connecting the ϕ-node to its uses), since s cannot be reduced (for the sake of correctness). The most minimal cost we can have is when $n = 1$ or $out = 0$. ($n \geq 1$, because at least one node must define the ϕ function. $out = 0$ is possible, as stated earlier.) In the case that $out = 0$, the ϕ function will be coalesced with every use point of that function. That means that the total number n of nodes defining this function will equal d (the number of use points of the ϕ function). Therefore,

$$C = n \times s + \sum_{i=1}^{n} (out_i) = n \times s = d \times s = \mathbf{s} \times \mathbf{d} \tag{6.5}$$

(corresponding to spatial placement) In the case that $n = 1$, that means that there is only one node defining a given ϕ function. This means that either a) there is a directed edge from this node to every use point or b) there is only one use point and this node has been coalesced with it. In the case of part a, the total number of directed edges leaving the one ϕ node is equal to d (the number of use points) therefore

$$C = 1 \times s + \sum_{i=1}^{n} (out_i) = s + out = \mathbf{s} + \mathbf{d} \tag{6.6}$$

(corresponding to temporal placement) Part b is a special case of $C = s \times d$ ($n = 1$, $out = 0$). Therefore, we can minimize the total in/out degree of the ϕ-node(s) by selecting the smaller of the two equations ($C = s + d$, $C = s \times d$). This selection corresponds to either choosing temporal placement (in the case of $s + d < s \times d$) or choosing spatial placement (if $s + d > s \times d$). This minimization of the degree of the ϕ-node(s) leads to minimal data communication in the CDFG. □

Theorem 4 *The asymptotic complexity of the Spatial SSA Algorithm is the same as the asymptotic complexity of the pruned SSA algorithm:* $O(|E_{cfg}| + |N_{cfg}|^2)$.

Proof: Def-use chain calculation, topological sort, and removal and restoration of back edges are all linear time graph operations (at most

a complexity of $O(|E_{cfg}| + |N_{cfg}|))$. Likewise the loop of the Spatial SSA Algorithm is $O(|N_{cfg}|^2)$ in the worst case, as the loop executes $O(|N_{cfg}|)$ times, and each time there is a potential to move a ϕ-node to its spatial position (a worst-case $O(|N_{cfg}|)$) operation). Therefore the complexity of the entire Spatial SSA Algorithm can be reduced to the asymptotic complexity of performing SSA, its most complex operation. We have previously shown that the complexity of performing SSA using the pruned SSA algorithm is $O(|E_{cfg}| + |N_{cfg}|^2)$. □

In this discussion, we examined the use of SSA to minimize the amount of data communication between control nodes. We showed a shortcoming of SSA when it is applied to minimizing data communication. The temporal positioning of the ϕ-node is not optimal in terms of data communication. We formulated an efficient algorithm to spatially distribute the ϕ-node to minimize the amount of data communication. Additionally, we proved that if all data communication wire-lengths are of unit cost, the Spatial SSA Algorithm provides optimal data communication.

3. Customized Resource Allocation

A customized resource is a module which is optimized to realize one or more particular functions. Each operation in a dataflow graph can be realized using a customized block or reconfigurable logic units. This is called a *customized block candidate* (or module candidate). There is a gain and a cost associated with each customized block candidate. The gain is the increased performance of the customized block, faster runtime compilation, etc. The performance can be delay of computation on the resource, power consumption, etc. The cost comes from non-flexibility, since not every operation with any functionality can be realized on such resources. Utilization of a customized block is an important target, first due to utilization of silicon and second due to the effort to custom design a module on the platform. It is not cost-effective to have customized modules that either are not used in many applications or do not yield a significant gain. Hence, the objective is a trade-off between the associated cost and gain with each module candidate.

Profiling the data path of applications is a good way to extract such customized block candidates [Cadambi and Goldstein, 1999]. Profiling is mostly applied to DAG representation of systems. For example, we extract control data flow graphs (CDFG) from a compiler as described in the previous section. Any customized candidate can be represented as a sub-graph. Sub-graph matching can be applied to extract the customized blocks in the data path of each application, to study their criticality in

performance or any other objective function such as power consumption, etc.

Other than the associated gain and cost for each customized block candidate, there is an overlap cost between the candidates in applications. Figure 6.9 shows an example in which different module candidates (called *patterns*) have overlap. In Figure 6.9, different extracted modules (or patterns) on the data flow graphs are shown. Candidate 1 has been observed two times in Figure 6.9. There are overlaps between customized blocks 1, 2, and 3. If customized block 1 and customized block 2 overlap in application C, only one of them can be used by application C. If there are customized resources associated with both customized block candidates 1 and 2 in the platform, both resources would not be fully utilized when application C is implemented. This dependency between candidates are considered in our model and the DAG representation of the problem.

The problem is similar to resource allocation problem on a scheduled data flow graph in high level synthesis. Resource allocation problem is resolved by solving graph coloring problem in a conflict graph. However, this solution cannot be applied to our defined problem. There are two main differences. First is that overlap does not mean that two resources cannot be chosen to be embedded on the chip. However, it leads to less utilization. Basically the overlap between two patterns implies that during scheduling and binding in highlevel synthesis, only one of the two customized blocks associated with the patterns i and j will be used. The other problem is due to the decision problem on the number of instances of each candidates required in the target architecture. It is not easy to handle this in the conflict graph. Therefore the problem focused in this work is referred as *customized block allocation* problem.

The problem of customized block allocation is more challenging when the reconfigurable platform is shared among a set of applications or there are variations of an application to be implemented on the platform [Keutzer et al., 2000]. Different sets of customized block candidates are demanded by different applications. The goal is to find the set of candidates to have a custom-designed physical resources on the platform. *Demand* corresponds to the associated performance gain for each customized block in an application. The constraint can be the limit on cost of customization. The customized blocks are embedded on the part of the system which is not reconfigured from application to application.

In this section we formulate a generalized customized resource allocation problem, transform it into an optimization problem in a directed acyclic graph, and propose an algorithm to solve the problem. Different variations of customized resource allocation problem can be solved in

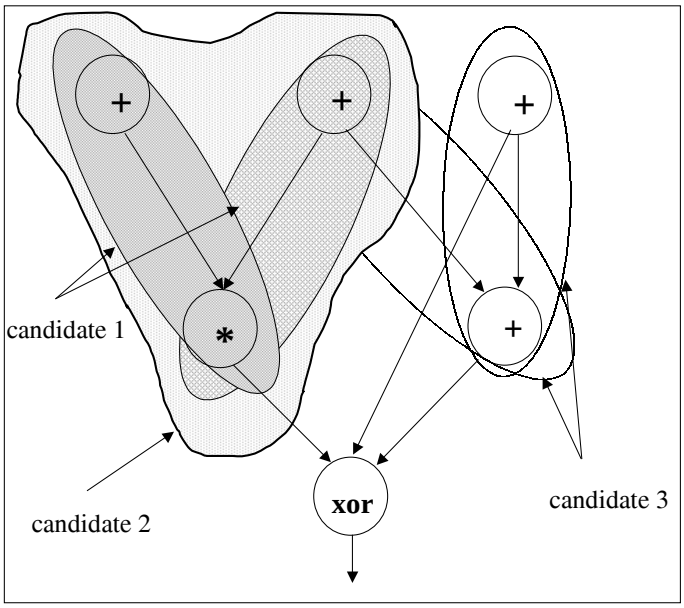


Figure 6.9. **Different Customized Block Candidates on a Data Flow Graph.**

this graph. Our gain and cost model for customization is general, as is the objective function. Each application requires a different number of resources for each operation. This is taken into account in the graph representation as well.

We assume that the following data for each application are provided as an input to customized block selection problem:

1 There is a performance *gain* associated with each customized block.

2 The area assigned for embedded basic blocks on the platform is restricted. In order to obtain maximum utilization, all customized block candidates cannot be chosen to be embedded.

3 The gain of each customized block depends on the given frequency of occurrences of each customized block candidate in data flow graphs (DFGs). The frequency of each candidate in a dataflow graph does not imply the number of physical resources required for those operations in the platform. The upper bound on the required number of resources for each customized block candidate can be obtained by applying an ASAP scheduler on the dataflow graphs. The ASAP scheduler schedules the data flow graphs and returns the maximum number of instances of each customized block used in scheduled DFGs.

4 Overlap between the instances of customized block candidates on the dataflow graphs are given after applying profiling and subgraph matching. The number of times two different sub-graphs overlap in a dataflow graph implies that both sub-graphs cannot be matched on the dataflow graph at the same time. Therefore on mapping the application to the platform, only one resource realizing either of the subgraph will be used.

3.1 Gain and Overlap Model

Assume that there is overlap between two customized blocks in a given data flow graph. Assume g_i and g_j are the gains associated with customized block i and customized block j. During scheduling and binding in highlevel synthesis, only of the two customized blocks associated with the patterns i and j will be used . We assume that any of the two customized blocks b_i and b_j can be used with probabilities p_i and p_j, respectively. The total gain is not simply the summation of both gains, but rather a weighted average as follows:

$$g_{total}(i,j) = p_i g_i + p_j g_j, \tag{6.7}$$

where $p_i + p_j = 1$. Equation 6.7 can be rephrased as follows:

$$g_{total}(i,j) = g_i + g_j - overlap_{unit}(i,j), \tag{6.8}$$

where

$$overlap_{unit}(i,j) = (1-p_i)g_i + (1-p_j)g_j. \tag{6.9}$$

In Equation 6.8, $overlap_{unit}$ is the overlap between single instance of pattern i and pattern j. Here we assume that in case of overlap each of the customized blocks can be selected with equal probability ($p_i = p_j = 0.5$). In addition, the frequency of occurrences of each customized block has to be considered while computing the gain of the customized block for an application. Assume the number of times customized blocks i and j have been observed in an application are occ_i and occ_j, respectively. c_{ij} is the number of times customized blocks i and j have overlap in a given application. Assume d_i and d_j are the number of customized resources associated with patterns i and j demanded by an application. If there are r_i and r_j number of embedded resources for customized block i and j, the overlap and gain functions can be approximately computed as follow:

$$Overlap(i,j) = \frac{r_i}{d_i} \cdot \frac{r_j}{d_j} \cdot c_{ij} \cdot overlap_{unit}(i,j), \tag{6.10}$$

$$gain_i = \frac{r_i}{d_i} \cdot occ_i \cdot g_i, \tag{6.11}$$

$$g_{total}(i,j) = gain_i + gain_j - Overlap(i,j), \tag{6.12}$$

where $r_i (0 \leq r_i \leq d_i)$ and and r_j $(0 \leq r_j \leq d_j)$ are the number of available resources associated with customized block candidates i and j. Equation 6.12 can be extended for all applications by summing the gain function over all applications:

$$Gain = \sum_{k=0}^{app} (\sum_{i=0}^{n} f(i,k) \cdot occ_{i,k} \cdot g_i - \frac{1}{2} \sum_{j=0, i \neq j}^{n} g(i,j,k) \cdot c_{ijk} \cdot overlap_{unit}(i,j,k)) \tag{6.13}$$

where

$$f(i,k) = \begin{cases} \frac{r_{i,k}}{d_{i,k}} & r_{i,k} \leq d_{i,k} \text{ and } d_{i,k} \neq 0 \\ 0 & d_{i,k} = 0 \\ 1 & r_{ik} > d_{ik} \end{cases}$$

, and

$$g(i,j,k) = \begin{cases} \frac{r_{i,k}}{d_{i,k}} \cdot \frac{r_{j,k}}{d_{j,k}} & r_{i,k} \leq d_{i,k} \text{ and } r_{j,k} \leq d_{j,k} \\ \frac{r_{i,k}}{d_{i,k}} & r_{i,k} \leq d_{i,k} \text{ and } r_{j,k} > d_{j,k} \\ \frac{r_{j,k}}{d_{j,k}} & r_{j,k} \leq d_{j,k} \text{ and } r_{i,k} > d_{i,k} \end{cases}$$

The subscript k in the coefficients and variables of Equation 6.13 shows their corresponding value in application k, $k = \{1, 2, ..., app\}$.

3.2 Problem Formulation

Customized Block selection problem can be formulated as follows:

- Given a set of customized block candidates, $P = \{p_1, p_2, ..., p_n\}$ with corresponding
  1. Gain set $G = \{g_1, g_2, ..., g_n\}$,
  2. Area set $A = \{a_1, a_2, ..., a_n\}$,
  3. Occurrence of each customized block i in each application j (occ_{ij}), and
  4. Demand Set in each application j for each customized block i (d_{ij}),
- The objective is to choose $R = \{r_1, r_2, ...r_n\}$ such that the total gain function as formulated in (6.13) is maximized.
- Subject to: $0 \leq r_i \leq max(d_{ij})$ and $\sum_{i=0}^{n}(r_i \times a_i) \leq A_{max}$, where $j = 0, 1, ..., k$ and $i = 0, 1, ..., n$ and where A_{max} is the maximum area assigned for embedded customized resources in the platform.

The objective function is a quadratic function. If only one application is considered, the problem can be solved by a quadratic programming solver. However, considering multiple applications causes the coefficients $f(i,k)$ and $g(i,j,k)$ to get non-smooth. Therefore piece-wise quadratic programming might be applied to solve the aforementioned optimization problem.

In the next sub-section, we introduce the *overlap graph*. We show that any instance of customized block selection problem can be represented by the overlap graph.

3.3 Overlap Graph

The overlap graph is an undirected weighted graph $G = (V, E)$. The weight of an edge can be negative. Each customized block candidate is represented by a node in graph G. The label of the node is the area associated with customized block candidate (original nodes). An edge between two nodes associated with the two patterns corresponds to overlap between the two patterns. Since overlap is a cost, the weight is the negative of the value returned by function $overlap_{unit}(i, j)$ in Equation 6.10. To each node associated with patterns in the graph (or called *original* node), a dummy node is connected. There is a dummy node associated with each original node in the graph. The dummy node is connected to its original node with an edge. The weight of the edge is the gain of the pattern associated with the original node (Equation 6.11). The labels of the dummy nodes are zero. Figure 6.10 shows an overlap graph. Customized Block selection problem is transformed into a problem of extracting an induced subgraph of G such that the summation over weights of edges inside the subgraph is maximized subject to summation over labels of nodes inside the subgraph does not exceed a given limit (see Figure 6.10). Lemma 3.1 mentions the properties required for an induced subgraph to be a feasible solution of the corresponding customized block selection problem. The first property is the area constraint. The other property says that the correct number of dummy nodes has to exist in the subgraph.

Lemma 3.1 *Summation over the weights of the edges in an induced subgraph of the overlap graph corresponding to customized block selection problem returns the objective value for the customized block selection problem if the conditions below are satisfied:*

- *The summation over the labels of the nodes inside the subgraph is less than a given limit, which is equivalent to area limit assigned for embedded customized blocks in the system. This is referred as area constraint.*
- *The dummy node associated with any original node is included in the subgraph iff the original node is itself inside the subgraph. This is referred as the gain constraint.*

Proof: It can be easily proven by contradiction. If any of the two conditions mentioned in Lemma 3.1 is removed, the summation of weights of edges inside the subgraph is not the objective value for customized block selection problem. □

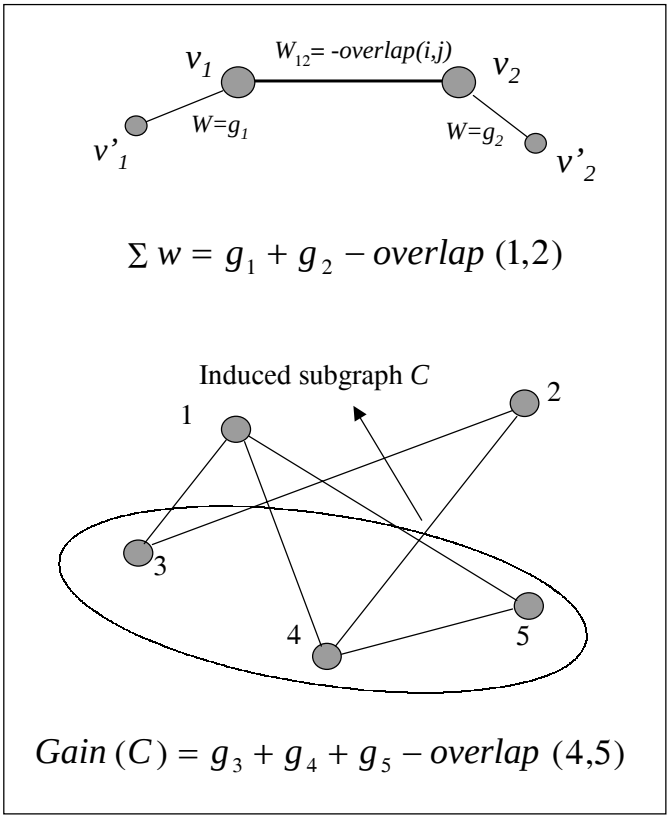


Figure 6.10. **Overlap Graph.**

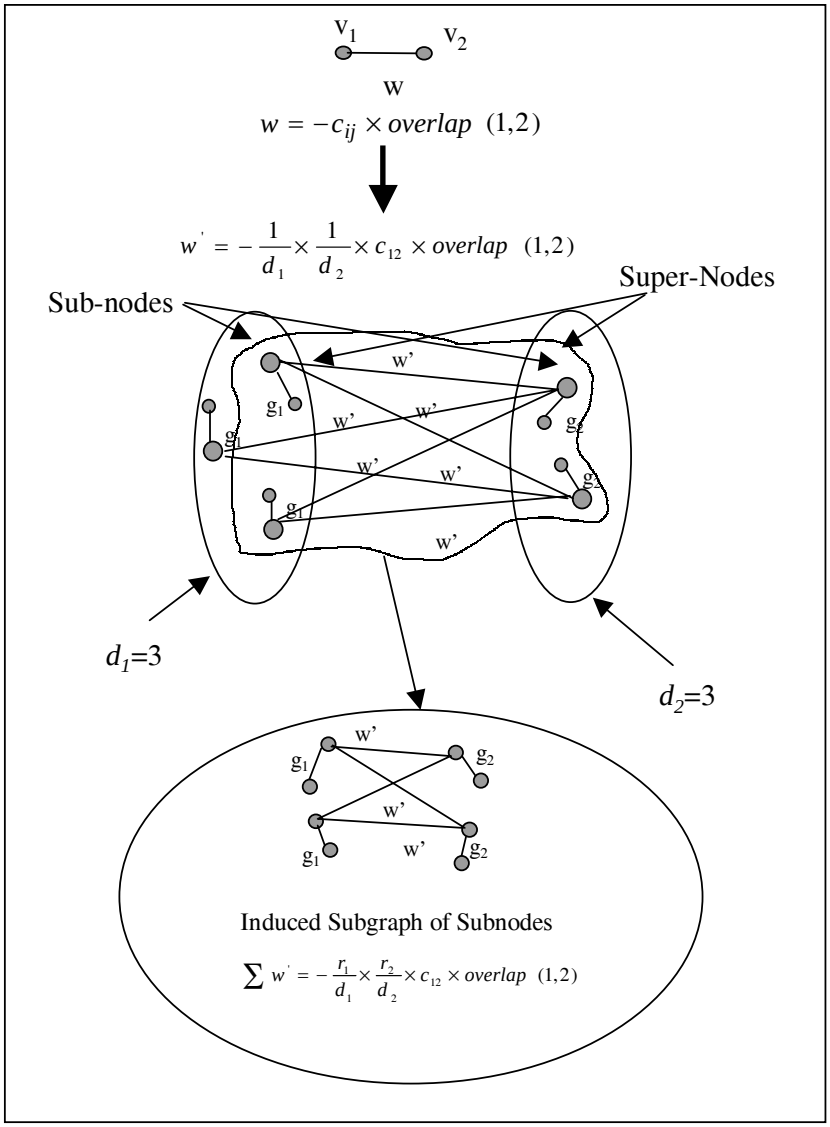


Figure 6.11. **Supernodes and Subnodes in Overlap Graph**

We modify the graph in order to be able to handle the multiple instances of customized blocks and multiple application demands. Figure 6.11 shows two nodes of overlap graph with corresponding labels and weight of the edge between two end nodes. Assume v_1 represents customized block 1 and v_2 represents customized block 2. There are two resources for customized block 1 ($d_1 = 2$) and three resources for customized block 2. The edge between the two nodes shows the overlap between the customized blocks. According to Equation 6.12, the weight of the edge is:

$$w_{i,j} = -Overlap(i,j) \tag{6.14}$$

$$w'_{i,j} = -\frac{1}{d_i} \cdot \frac{1}{d_j} \cdot c_{ij} \times overlap_{unit}(i,j) \tag{6.15}$$

$$w_{i,j} = r_i \cdot r_j \cdot w'_{i,j} \tag{6.16}$$

$$w'_{d_i} = \frac{1}{d_i} \cdot occ_i \cdot g_i \tag{6.17}$$

$$w_i^{dummy} = r_i \cdot w'_{d_i} \tag{6.18}$$

The weight of an edge depends on the number of instances selected from nodes v_1 and v_2. In the overlap graph, node v_1 and node v_2 are divided into d_1 and d_2 sub-nodes, respectively, shown in Figure 6.11. There is an edge with weight w' between any two instances of node v_1 and v_2. We refer to w' as *unit overlap weight*. *Supernode* is a node representing all the node in the graph G representing instances of the same pattern or candidate and their corresponding dummy nodes. In graph G, there is a node associated with each instance of any patterns. Relative to supernodes, the original nodes and dummy nodes of graph G are referred as *subnodes*. Supernode is a set of subnodes. As shown in Equation 6.15 and 6.16, overlap between two nodes can be defined in terms of unit overlap weight between the two supernodes and number of subnodes of the two overlapping supernodes in overlap graph. Equations 6.17 and 6.18 similarly show that the individual gain of each supernode can be expressed in terms of the number of subnodes and the unit gain. The subgraph shown in Figure 6.11 includes two instances of v_1 and

two instances of v_2. The total summation over the weights of the edges is equivalent to the objective function *Gain* for the customized block selection problem. The label of each subnode is the area of the instance.

The overlap graph representation can be easily extended to handle multiple applications. In this case, the number of occurrences and demands for each customized block candidate is different. We consider multiple edges between supernodes. Each edge corresponds to the overlap between the two end supernodes of the edge in one application. Similarly, there are multiple edges between subnodes for each supernode. In addition, we define multiple occurrences for each supernode. The number of subnodes of a supernode is as many as the maximum number of resources demanded by all applications for the corresponding customized block candidate. The number of subnodes for supernode i among applications $k = 1, ..., app$ is $max(d_{i,k})$ over k.

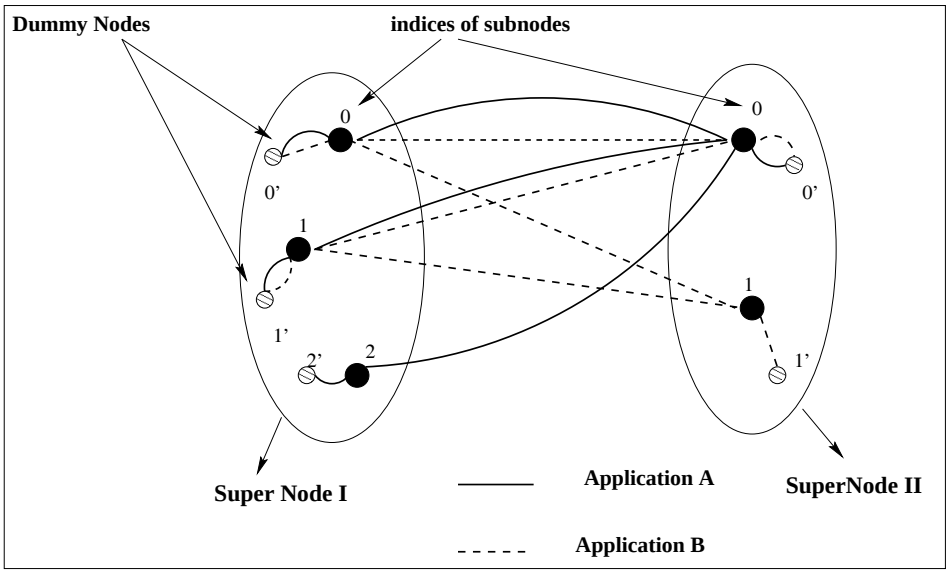


Figure 6.12. **Overlap MultiGraph**

As shown in Figure 6.12, each subnode corresponds to an instance of a supernode. Assume that application A demands 3 resources for customized block node I in overlap graph. Application B demands two instances of supernode I. Supernode I has three subnodes (original subnodes). Therefore, it consists of three subnodes connected through edges to supernode II. The first set of edges is according to overlaps

between node I and supernode II in application A. The overlap graph is a multigraph , i.e. a graph with more than one edge between two nodes in the graph. Since two instances of node I were demanded by application A, there exists an edge between the first two instances of node II and the first two instances of supernode II. In order to handle this issue, we must index the subnodes of each supernode. Index i of customized block I corresponds to the i^{th} resource of customized block I.

3.4 Customized Block Selection Algorithm

In this section, we propose an algorithm for the customized block selection problem formulated in Section 3.2. The problem of finding an induced subgraph in a graph is equivalent to clustering a subset of nodes in the graph. In the customized block selection problem, we generate a single cluster on overlap graph.

Theorem 5 *The problem of generating a single cluster on an overlap graph including edges with negative weights such that the total summation of the weights of the edges inside the cluster (induced subgraph) is maximized is NP-Complete.*

Proof: Assume that there is an overlap graph with n supernodes. Assume that there is only one instance of each supernode. Therefore, there is one subnode for each supernode. The gain of each supernode in customized block selection problem is 1 and the weights of the edges between the supernodes are all $-n$. We can transform this graph into a graph G' in linear time. In G', each node represents a supernode, the label of each node is set to 1 and weights of all edges are equal to $-n$. The problem to find an induced subgraph S in this graph such that the summation of edges between the the nodes inside the cluster S is maximum. It can be easily seen that this problem is as hard as finding maximum independent set in undirected graph G' [Garey and Johnson, 1999]. □

According to Theorem 5, the problem is an intractable problem. Therefore, a heuristic has to be developed to solve the problem. We propose a simple and fast heuristic to solve this problem. We support a sequential clustering approach, in which one node at a time is chosen to be added to the cluster. Clustering algorithm has to satisfy both the area constraint and the gain constraint. The area constraint can be easily handled. When a node is added to the cluster, it is checked if the area constraint is satisfied. If the label of the node added to the total labels of nodes in cluster is not less than a given limit, the node is not a feasible choice and cannot be added to the cluster. In order to satisfy

the gain constraint, the clustering algorithm has to deal with two cases. While constructing the cluster, we avoid choosing any dummy node before selecting the node associated with the dummy node. Therefore, no redundant dummy node is added.

In sequential clustering , nodes are added to the cluster iteratively. We have to choose a node among a set of nodes outside the cluster referred as *candidate* nodes. We choose each candidate based on the local closeness to optimality. Each candidate node is the one increasing the objective value the most. However, it might not be feasible. For feasibility, the node must satisfy the two constraints in Lemma 3.1. Otherwise the next best candidate is chosen. *potential* function is used to judge and define the ranking of the candidates.

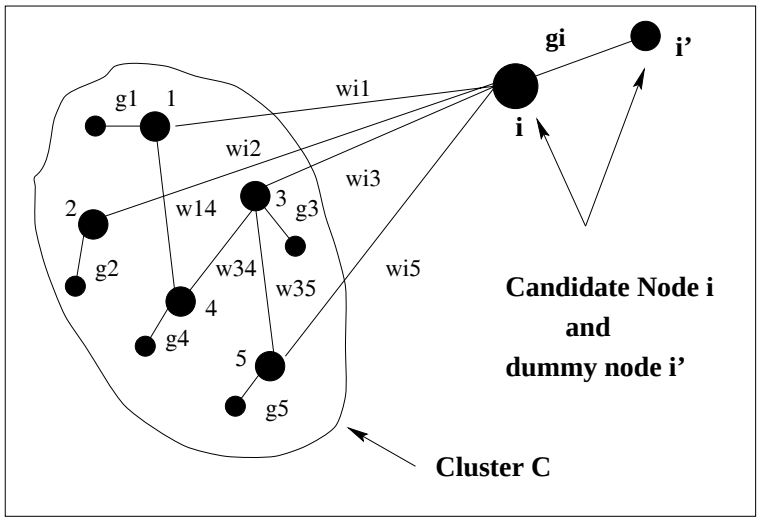


Figure 6.13. **Candidate Node i Being Added to Cluster C.**

Figure 6.13 shows candidate node (node i) being added to a cluster C. Node i' is the dummy node associated with node i in overlap graph. The objective value for the current cluster is the summation over the weights of the edges inside the cluster. If node i is added to cluster C, the change in the value of the objective function is the total weights of the edges connecting node i to the nodes inside the cluster.

We define the potential function $p(i, C)$ for each candidate i in Equation 6.19. Since there is an area constraint, we consider utilization in our potential function by dividing by the area of the embedded customized block candidate, i.e., the label of the node in overlap graph.

$$p(i, C) = \frac{g_i + \sum_{j \in C} w_{ij}}{l_i} \tag{6.19}$$

Lemma 3.2 *A required condition for a candidate node to be added to a cluster in sequential clustering is that* $\sum_{j \in C} w_{ij} + g_i > 0$*, if the current nodes in cluster are assumed not to be pruned later.*

Proof: If a node with negative potential is added to cluster, it decreases the gain of the cluster itself. The gain of that node cannot be positive, since the nodes already inside clusters are fixed. □

```
CUSTOMIZED BLOCK SELECTION ALGORITHM (MultiGraph G,subgraph C)

1.    for each supernode v in G
2.        Compute_Potential_Gain(v, index(v))
3.        Insert_Candidate_Queue(C, index(v))
4.    endif
5.    while Priority_Queue_Not_Empty(C)
6.        candidate <-- Queue.front
7.        if Area_Constraint(candidate) && Potential_Candidate(C,candidate)

8.            Update_Neighbors(candidate, G)
9.            Update_SuperNode(candidate)
10.           Increase_SuperNode_index(candidate, C)
11.       endif
12.       Remove_From_Queue(candidate, C)
13.       Update_Gain_Queue(C)
14.   endwhile
```

Figure 6.14. **Pseudocode of Customized Block Selection Algorithm.**

This condition avoids selecting a dummy node if the original node associated with the dummy node is not already in a cluster. The associated dummy node is connected to the graph only through the corresponding original node. Hence, if the node is inside the cluster, the dummy node has a positive potential function value towards the cluster. Otherwise the potential function value for such dummy node is zero. Therefore this condition satisfies the gain constraint mentioned in Lemma 3.1.

The potential gain of each node i inside cluster C cannot become negative after adding the candidate. The pseudocode of Customized Block Resource Allocation Algorithm is shown in Figure 6.14.

Lemma 3.3 *The complexity of the Customized_Block_Selection_Algorithm is* $O(r_{max}^2 \times n^2 \times k)$*, where* n *is the number of supernodes,* r_{max} *is the*

maximum number of subnodes belonging to supernodes in overlap multi-graph , and k is the number of applications.

Proof: In this algorithm each node is visited once. Each time a node is selected, the number of nodes to be updated is equal to the edge degree of the node. Therefore, the clustering algorithm process takes at most as long as visiting all the multi-edges on the overlap multi-graph, i.e. $O(E)$. The number of nodes, N_s, is equal to $\sum_{i=0}^{n}(r_i \times n_i)$, where r_i is the number of subnodes for supernode n_i. The complexity of the algorithm is $O(N_s^2)$. It can easily be shown that $O(N_s^2) \leq O((r_{max}^2 \times n^2 \times k)$. □

4. Simultaneous Scheduling and Binding: Customized Resource Utilization/Latency Minimization Trade-off

In this section we present a simultaneous scheduling and binding algorithm for reconfigurable architectures. Customized resource allocation which precedes synthesis, provides a set of specialized resources. The operations in a given application are scheduled such that these available resources are utilized in the best way to improve performance. The architecture is customized with these resources for a particular family of applications, or a specialized library containing the resources is provided to the synthesis stage. Hence, when implementing one of those applications, the timing critical operations common to this family are expected to be executable by them. Our scheduler is aware of the customization of the target architecture. It maintains a balance between two extreme cases. On one hand, the customized resources should be used as frequently as possible. On the other hand, offloading all operations onto a limited number of customized resources might lead to the underutilization of available parallelism. Functional units for any desired operation type can be instantiated using reconfigurable logic. For operations that cannot be performed by the specialized resources, this is a necessity. For other operations this can be done in order to exploit parallelism in the schedule, i.e., operations that can be executed in parallel can be asigned to customized resources as well as to reconfigurable resources if there are not enough customized resources. However, as mentioned earlier, the available blocks are highly preferred for those operations. It is the task of the scheduler to evaluate the trade-offs in such situations.

4.1 Problem Formulation

Given a data flow graph (DFG) as input, the problem of simultaneous scheduling and binding is assigning a clock cycle as a start time

and a resource to each operation in the DFG. The input is a DAG $G = (V, E)$, where $V\}is the set of operations in the DFG$ and $E \subseteq V \times V$. Each operation op has a type $OpType_op$ from the set of operation types $Types = \{OpType_1, ..., OpType_{Types}\}$.

Resources available for the schedule are provided from a library of pre-designed and pre-characterized modules $L = \{(ModuleType_i, Metric_i) \mid Type_i \subset Types, Metric_i = \{area_i, delay_i, ..., power_i, implementationType_i, availability_i, reconfiguration_i\}\}$. The set $Metric_i$ contains several performance metrics related to the functional units. The $implementationType_i$ metric distinguishes between reconfigurable blocks and fixed blocks. $availability_i$ indicates the number of fixed blocks from one type on the system: there may be multiple copies of the same block embedded in one architecture. $reconfiguration_i$ gives the reconfiguration overhead in terms of reconfiguration time for programmable components.

$L = Core \cup Rec$, i.e., L contains both $Core$, the set of embedded customized blocks available on the target architecture and Rec, the set of reconfigurable modules. A schedule generates the set of resources $F = \{f_i \mid f_i \in L\}$ used from the library and a one-to-one matching $Binding : V \rightarrow F$. Also, a start clock cycle $start(op)$ for each operation $op \in V$ in the DFG is determined. From start times of each operation the finish time is found as $finish(op) = start(op) + delay(Binding(op))$.

A schedule is valid if:

- For each operation a valid start time and a binding are defined.
- Data dependencies imposed by the DFG are not violated: $\forall(op1, op2) \in E, start(op_2) \geq finish(op_1)$.
- Each resource can perform *at most* one operation at any given clock cycle: $\forall i, \forall j, \text{if} start(i) \leq start(j) < finish(i) then$ $Binding(i) \neq Binding(j)$.
- At any time step, number of active resources of type t, must be less than or equal to the number of available modules of type t.

4.2 Proposed Scheduling Algorithm

The general resource-constrained scheduled problem is known to be NP-Complete [Garey and Johnson, 1999]. In this section we present an algorithm for this problem which can optimally solve subproblems within the global problem. Our method provides an incremental scheduling scheme by handling one path within the given DFG at a time. The

global problem is first divided into sub-problems of scheduling individual paths. The subproblem is then solved with a bipartite graph matching approach. A similar method proposed in [Timmer and Jess, 1995] generates a complete bipartite matching for a prospective schedule and then imposes precedence constraints on the matching by removing edges. Our approach to the problem is different, in that the bipartite matching is integrated with the scheduling constraints in one formulation. Then each matching problem is solved optimally. In the following, we describe each task within our method individually.

Selection of Paths in the DFG: Paths are selected from the input DFG according to a criticality criterion. Paths are prioritized according to their expected lengths. After a selected path is scheduled, the schedules of the operations on it are fixed. The operations are removed from the DFG while annotating the DFG with the delay information of the scheduled parts. From the remaining graph paths, *partial* paths are selected and scheduled. The schedule of each (partial) path depends on the existing partial schedule. For the solution to the subtask of scheduling a path , we have adopted a geometric approach.

Scheduling Paths using Non-Crossing Bipartite Matching:

- Given a DFG and a target architecture, the first step is to set up a *resource assignment table*. The rows in this table correspond to clock cycles, and the columns correspond to hardware resources. The architecture specifications provide information on the types and numbers of fixed blocks. For each of those, a column is generated. The table can be extended with new columns as reconfigurable modules are added to the binding set. For an architecture model, where the set of available resources is known a priori a fixed size table is created. We will adopt the later assumption throughout the rest of the discussion. If an operation is scheduled at a certain cycle on a hardware resource, the corresponding entry in the table is marked as well as consecutive cycles within the same column for the duration of the operation's execution time.

- The problem instance for a path P is then converted to a bipartite graph representation $B(V^B, E^B)$, where
  $V^B = O \cup C,\ O = \{op \in P\}\ and\ C = \{cycle_i \mid 1 \leq i \leq max_{cycles}\}$
  $E = \{(u, v) \mid u \in O, v \in C\}$
  An example of a possible representation is depicted in Figure 6.15(a). The vertices on the left hand side of the bipartite graph correspond to operations in the path P. The nodes on the right hand side correspond to clock steps. An edge between an operation vertex and a

clock step vertex denotes the possibility of assigning the operation to that clock step. Every edge in B is assigned a weight w. The meaning of the edge weights and their usage in the algorithm will be explained later. First we explain how to generate the edges of B.

- For two consecutive nodes in P $i, j \in O$ let $c_i \in C$ be the earliest clock step node with an edge between i and c_i. Similarly let c_j be the earliest clock step node with an edge between j and c_j. Then,
  $c_j > c_i + min\{delay(f) \mid f \in F \ \ can\ implement\ i\} - 1$
  This condition prevents the representation from violating data dependencies.
- Another issue to consider is related to the availability of hardware resources. If the algorithm is applied to a model with a fixed amount of resources, then the following holds:
  between any operation $i \in O$ and clock step c there cannot be an edge, if
  at clock step c no resource of matching type for operation i is available. This information is obtained from the resource assignment table. This condition is not applied to the problem, if we assume that there is reconfigurable logic available to create extra resources if necessary.
- If path P is scheduled before path Q and there is an operation $i \in Q$ with a predecessor in P, then i must also obey the dependencies to this predecessor. The earliest clock cycle node adjacent to i must be later than the finish cycle of the predecessor.
- Similarly, let P contain an operation i and let a previously scheduled path Q contain a successor j of i. Then no edge between i and any clock step c can exist, if $c + max\{delay(i)\} > start(j)$.

- Each edge is assigned a weight w that represents conceptually the following:
  - From a single operation's perspective, the weight w of an edge $e = (i, c)$ represents the tendency of operation i to be scheduled at clock step c. Each operation would preferably be scheduled at the clock step onto which an edge from the operation is incident with highest weight.

- Comparing several operations with edges incident on the same clock step, the operation with highest edge weight would have highest priority.

We generate weights for the edges in the bipartite graph considering several factors. The most obvious one is the tendency of operations to be scheduled *early*. Since the primary goal is to obtain a schedule with low latency, the algorithm should not delay the schedule of operations. If the objective of our scheduling algorithm were only latency minimization, then the weights for edges could be generated using a monotonically decreasing function starting from the weight of the edge incident on the earliest cycle.

Having a target architecture providing alternative hardware resources (customized blocks and reconfigurable logic, different implementations of modules for different performance criteria), the meaning of edge weights is extended to incorporate the tendency of operations towards certain resources. When considering two different clock steps for the possible start time of an operation, the one with a free customized resource will be preferred over another at which no optimized resource is free. There still may be an edge between the operation and the latter cycle, assuming a reconfigurable module is available. Nevertheless, the edge weight of the first cycle is made larger in order to make the algorithm aware of the resource preferences. In this case the function computing the edge weight incident on a particular clock step will be a function of both the clock step itself and availability of certain type of resource at that step. If no customized block is free at a certain step, then a constant value could be subtracted from the index of the clock step. In this manner the customized features of the target architecture can be exploited. Similarly, depending on the particular optimization objective at the current level, different implementations of resources might be preferred. Optimization objectives such as low power, high speed performance, low reconfiguration overhead might lead to different choices on the resources from a library. Such considerations are incorporated into our algorithm through appropriately generated weight functions.

- The solution to the scheduling problem of a given path P is now finding a non-crossing matching on the constructed bipartite graph such that the sum of edge weights are maximized. Such a solution will provide a one to one matching between operation nodes and clock step nodes. As the operations in the current path are ordered according to their respective data dependencies in the DFG, the

selected edges in the bipartite matching should not cross. A cross would indicate that a node is assigned to an earlier clock step that its predecessor in the path, which violates the data dependency constraint. We find this matching by converting the problem into a geometric representation. For each possible matching indicated by an edge in the bipartite graph, a point in the $x-y$ plane is created. For $e = (i, c)$ a point $p = (x, y)$ is created, such that $x = index(i)$ and $y = c$. The operation nodes in the path are indexed starting from 1. An example bipartite graph and corresponding point set on the plane are depicted in Figure 6.15. The weight of each edge is transferred as a weight to the corresponding point in the plane. The problem of finding a maximum-weight non-crossing matching is now equivalent to finding a maximum-weight chain of length k, where k is equal to the number of operations on path P. An example of such a non-crossing matching and the corresponding k-chain are shown in Figure 6.16. Within this chain each point *dominates* the preceding point, hence dependencies are observed.

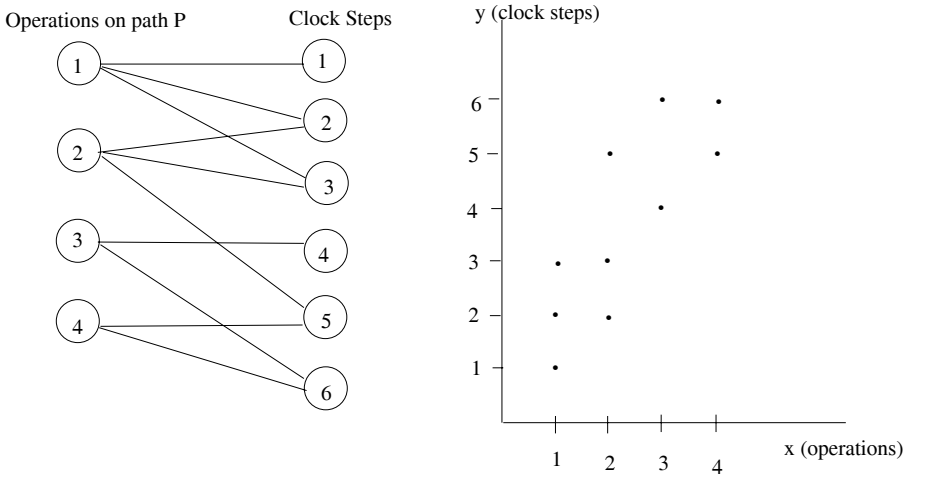
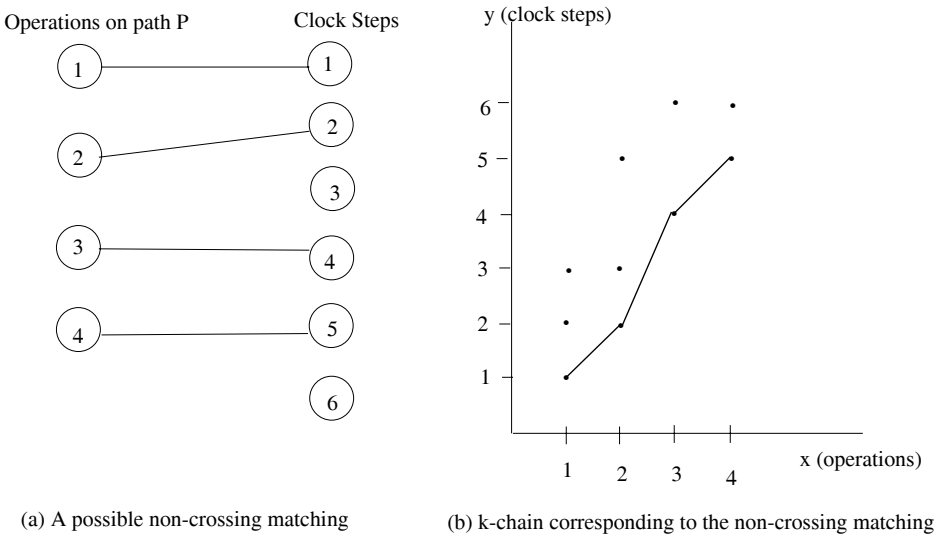


Figure 6.15. **Bipartite Graph Representation and Corresponding Point Set on the x-y Plane.**

Definition 1 *Point* $p = (p_x, p_y)$ *dominates point* $q = (q_x, q_y)$ *iff* $p_x > q_x$ *and* $p_y > q_y$.



(a) A possible non-crossing matching

(b) k-chain corresponding to the non-crossing matching

Figure 6.16. **Non-crossing bipartite matching example.**

Definition 2 *Each point p in the plane is associated with a level, such that none of the points in the same level dominate each other. Points in higher levels dominate points in lower levels.*

Definition 3 *The level associated with a point p in the plane is equal to the highest level among the points dominated by p incremented by one. The origin is assumed to have level* 0*, which is not used to place any points. The first actual level is level 1.*

Lemma 4.1 *There are exactly k levels in the resulting point set in our maximum-weight non-crossing matching problem. Two points have the same level iff they have the same x-coordinate.*

Proof: Since there are only k possible different x-coordinates for all points in our problem, there can only be k possible levels. Every x-coordinate corresponds to one of the k operations on path P. The points with same x-coordinate, but different y-coordinates have the same level, since none of them can dominate another. Let two points $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$ be at different x-coordinates, such that $x_1 < x_2$, i.e. on the path P, operation corresponding to p_1 precedes operation corresponding to p_2. In order for these points to have the same level y_2 must be less then

or equal to y_1, otherwise p_2 would have dominated p_1. There can not exist such a point p_2, because the first rule of edge generation prohibits assignment of an edge to p_2 anytime equal or earlier to the first cycle that p_1 was matched with an edge. That means any point at x-coordinate x_2 having smallest y-coordinate would dominate at least one point at x-coordinate x_1. □

According to Lemma 4.1, a level value to each point in the plane can easily be assigned, where the level of each point is equal to its x-coordinate. A polynomial time algorithm for finding the maximum-weight chain is proposed in [Atallah and Kosaraju, 1989]. In the case of unit weights, this algorithm returns the longest chain, which naturally corresponds to the maximum sum of weights. However, when arbitrary weights are assigned to the points in the plane, this algorithm yields the maximum weighted chain, but not necessarily of length k.

Another procedure proposed by [Raje and Sarrafzadeh, 1993] *only* creates a chain of length equal to the number of operations by using the level information. The *max_weighted_k_chain()* procedure is given in Figure 6.17.

```
max_weighted_k_chain()

1 initialize labels of all points to their weights
2 for level = 2 to k do
3     for each point in level i do
4         max_label:= maximum label found at previous level
                      among dominated points
5         label(point):= label(point) + max_label
6         assign pointer from point to the dominated point
          providing the max_label
7     end for
8 end for
```

Figure 6.17. **Pseudocode for max_weighted_k_chain() Procedure.**

```
geom_Scheduling(DFG, IP_Library,
reconfigurableResource, costFunction(Parameter_List))

1  Initialize the Resource Assignment Table
2  while there exist unscheduled operations
3        select the most critical (partial) path
4        Generate the bipartite graph B=(V, E)
5        Assign weight w = costFunction(Parameter _List)
         to each edge e
6        Apply  max_weighted_k_chain()
7        Update the  Resource Assignment Table
8  end while
```

Figure 6.18. **Pseudocode for Overall Scheduling Algorithm.**

Following the pointers created in the procedure, the actual chain can be constructed as follows. The max-weighted chain construction starts with the point p_k at level k with highest label. p_k is added to the max_chain as the p_kth element. The pointer created after the update of p_k's label links it to a point p_{k-1}. By tracing this pointer, p_{k-1} is located and added to the max_chain as the $(k-1)$th element. A similar step is applied to p_{k-1}, and so on, until the last pointer points to the first element of the max_chain at level $= 1$.

Theorem 6 *max_weighted_k_chain() produces an optimal k-chain.*

Proof: When computing the labels of the points at $level = 2$, each point is linked to a unique point with largest label at $level = 1$, such that the sum of the weights of the two points is maximum and the dominance relation holds between the points. At this point the maximum label among all points at level 2 would indicate the maximum possible sum of weights for a 2-point chain. When the algorithm proceeds to the next level, labels at the new level will be computed using the partial sums computed in the earlier levels. We

know that those partial sums were the maximum possible values while maintaining dominance condition in the chain. Since each point at the new level will pick the maximum partial sum carried from the immediate lower level, the new partial chain sums (labels) will remain maximal. By induction on the number of levels, at kth level, the point with maximum label indicates the maximum sum of weights of a k-chain. □

Combining all the steps explained above, the overall scheduling algorithm is summarized in Figure 6.18.

The core of our algorithm is the maximum-weight k-chain procedure. In this procedure, at each level, the number of points is bounded by $O(C_{max})$, where C_{max} is the maximum number of clock cycles included in the bipartite graph. The number of levels is equal to the number of operations on the given path P. If we denote the number of operations on P by p, then the total number of points on the plane (the number of edges in the bipartite graph) is bounded by $O(pC_{max})$. The complexity of *max_weighted_k_chain()* becomes $O(pC_{max})$. This procedure is repeated until all operations in the input DFG are scheduled. If at every step i a path containing p_i operations is scheduled, then the total time to schedule a DFG can be expressed as

$$\sum_{i=1}^{i=max_{step}} p_i \cdot C_{max},$$

which is equal to

$$C_{max} \cdot \sum_{i=1}^{i=max_{step}} p_i \text{ and } \sum_{i=1}^{i=max_{step}} p_i = N.$$

max_{step} denotes the maximum number of scheduling iterations, in each iteration one path being scheduled. Assuming each operation in the DFG can be performed within a constant number of clock cycles, the maximum number of clock cycles required to schedule a DFG is bounded by the number of operations with a constant coefficient l. $C_{max} = O(lN)$. Eliminating the constant l, the complexity of the algorithm becomes $O(N \cdot N) = O(N^2)$.

5. Conclusions

Reconfigurability, an essential feature in present and future hardware designs, provided a tradeoff between design flexibility and performance. This produces the need for new optimization techniques in the design of programmable systems. In this chapter, we have presented a complete

design flow from application source code to reconfigurable hardware. We have furthermore described a hierarchical approach to optimize this design by identifying specific problems at each stage. We presented a theoretical approach to the optimization of these problems, and also provided efficient algorithms to perform these optimizations.

Reconfigurability is a new and challenging model of programming and execution of hardware, and it requires new methods and models to fully exploit its power. We have presented methods which are theoretically useful in mapping applications onto reconfigurable devices. However, there are certainly many other potential techniques for optimization yet to be investigated which will provide further exploitation of reconfigurability.

References

Atallah, M. and Kosaraju, S. (1989). An efficient algorithm for maxdominance with applications. *Algorithmica*.

Briggs, P., Cooper, K., Harvey, T., and Simpson, L. (1998). Practical improvements to the construction and destruction of static single assignment form. *Software Practice and Experience*, 28:859–881.

Bringmann, O. and Rosenstiel, W. (1997). Cross-level hierarchical high-level synthesis. Presented at the Design Automation and Test in Europe.

Brown, S., Francis, R., Rose, J., and Vranesic, Z. (1992). *Field Programmable Gate Arrays*. Kluwer Academic Publishers.

Cadambi, S. and Goldstein, S. C. (1999). Cpr:a configuration profiling tool. Presented at the IEEE Symposium on FPGAs for Custom Computing Machines.

Cytron, R., Ferrante, J., Rosen, B. K., and M. N. Wegman, F. K. Z. (1991). Efficiently computing ϕ-nodes on-the-fly. *ACM Transactions on Programming Languages and Systems*.

Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadek, F. K. (1989). An efficient method of computing static single assignment. Presented at the ACM Symposium on Principles of Programming Languages.

Dougherty, W. and Thomas, D. (2000). Unifying behavioral synthesis and physical design. Presented at the Design Automation and Test in Europe.

Garey, M. and Johnson, D. (1999). *Computers and Intractability*. W.H. Freeman and Company.

Graham, S. L. and Wegman, M. (1976). A fast and usually linear algorithm for global flow analysis. *Journal of the ACM*, 23(1):172–202.

Hauck, S. (1998). The role of fpgas in programmable systems. *Proceedings of the IEEE*, 86:615–638.

Karn, J. B. and Ullman, J. D. (1991). Global data flow analysis and iterative algorithms. *IEEE Trans. on Computer-Aided Design*, 10(1):172–202.

Kennedy, K. (1981). *A Survey of Data Flow Analysis Techniques, Program Flow Analysis: Theory and Applications*. Prentice-Hall.

Keutzer, K., Malik, S., Newton, R., and J. Rabaey, A. S.-V. (2000). System level design: Orthogonalization of concerns and platform-based design. *IEEE Trans. on Computer-Aided Design*, 19(12).

Khouri, K., Lakshminarayana, G., and Jha, N. (1998). Impact: A high-level synthesis system for low power control-flow intensive circuits. Presented at the Design Automation and Test in Europe.

Macii, E., Pedram, M., and Somenzi, F. (1998). High-level power modeling, estimation, and optimization. *IEEE Trans. on Computer-Aided Design*, 17:1061–1079.

Muchnick, S. S. (1997). *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, San Francisco.

Park, S., Kim, K., Chang, H., Jeon, J., and Choi, K. (1999). Backward-annotation of post-layout delay information into high-level synthesis process for performance optimization. Presented at the International Conference on VLSI and CAD.

Raje, S. and Sarrafzadeh, M. (1993). Gem: A geometric algorithm for scheduling. Presented at the IEEE Symposium on Circuits and Systems.

Schaumont, P., Verbauwhede, I., Keutzer, K., and Sarrafzadeh, M. (2001). A quick safari through the reconfiguration jungle. Presented at the Design Automation Conference.

Timmer, A. and Jess, J. (1995). Exact scheduling strategies based on bipartite graph matching. Presented at the European Design and Test Conference.

Wong, J., Megerian, S., and Potkonjak, M. (2002). Forward-looking objective functions: Concepts and applications in high level synthesis. Presented at the Design Automation Conference.

Xu, M. and Kurdahi, F. (1997). Layout-driven rtl binding techniques for high-level synthesis using accurate estimators. *ACM Transactions on Design Automation of Electronic Systems*, 2:312–343.