CONSTRAINT PROGRAMMING TECHNIQUES FOR GENERATING EFFICIENT HARDWARE ARCHITECTURES FOR FIELD PROGRAMMABLE GATE ARRAYS

by

Atul Kumar Shah

A thesis submitted in partial fulfillment
of the requirements for the degree

of

MASTER OF SCIENCE

in

Computer Engineering

Approved:

Dr. Brandon Eames Major Professor	Dr. Arvind Dasu Committee Member
Dr. Edmund Spencer Committee Member	Dr. Byron R. Burnham Dean of Graduate Studies

UTAH STATE UNIVERSITY
Logan, Utah

2010

Abstract

Constraint Programming Techniques for Generating Efficient Hardware Architectures for Field Programmable Gate Arrays

by

Atul Kumar Shah, Master of Science

Utah State University, 2010

Major Professor: Dr. Brandon Eames
Department: Electrical and Computer Engineering

This thesis presents an approach for modeling and generating efficient hardware architectures using constraint programming techniques, targeting field programmable gate arrays (FPGAs). The focus of this thesis is the derivation of optimal or near-optimal schedules for streaming applications from data flow graphs (DFGs). The resulting schedules are then used to facilitate the architecture generation process. Most streaming applications, like digital singal processing (DSP) algorithms, are repetitive in nature: the same computation is performed on different data items. This repetitive nature of streaming applications can be used to expose additional parallelism available across different iterations, by creating multiple instances of the same computation. The replication of the single computation, when applied to high level synthesis (HLS), improves the performance of the design but requires additional area. The amount of additional area required for a replicated graph can be reduced through the use of pipelined functional units and the addition of some extra clock cycles beyond the critical path of the DFG. This thesis discusses the use of a constraint programming (CP)-based scheduler to generate optimal schedules based on designer-provided replication level and critical path relaxation. The scheduler is an integrated part of the design tool, called CHARGER, which analyzes the resulting schedules to allocate memory for storing intermediate data, creates the infrastructure necessary to efficiently execute the application, and finally generates a synthesizable Verilog/VHDL code for the controller. The performance of the architectures derived using the CP-based scheduler is compared with the architectures generated using a force directed scheduling (FDS)-based scheduler for algorithms selected from embedded/multimedia applications. The results show that our CP-based scheduler outperforms the FDS-based scheduler, both in terms of area and efficiency of the generated architectures. The results show average area saving of 39% and average performance improvement of 41%.

(123 pages)

To my beloved family and friends

Acknowledgments

This research project would have never been completed without my advisor and mentor, Dr. Brandon Eames. This project is the outcome of his constant motivation and belief in me. I would have never been able to complete this research without his guidance, patience, and expertise in the field of constraint programming and embedded systems. His courses provided the foundation for my research and helped me become a good researcher. I am really grateful to him and would like to thank him for all his guidance, support, and precious time that he has devoted on this project. I would like to thank my committee members, Dr. Arvind Dasu and Dr. Edmund Spencer, for extending their support.

I would like to acknowledge my friends here at Logan, especially Nagendra, Karthik, Amrita, Santhi, and Lalit, with whom I have spent the best times of my student life. I would also like to thank my friends at the ECE Department: Rohit Saraswat, Hari Krishna Samala, Shant Chandrakar, and Sravanthi Venigalla for helping me on various occasions. Especially I would like to thank Hari for sharing his expertise on FPGAs and helping me with the VHDL/Verilog Controller generator.

And last, but not least, I would like to thank my father who always told me never to give up, and my mother who is the most important person in my life and the greatest strength behind all my successes.

Atul Kumar Shah

Contents

List of Tables

Table Page

List of Figures

Figure Page

Chapter 1

Introduction

Generating efficient optimal hardware architectures for field programmable gate arrays (FPGAs) is a challenging problem because of the large number of architectural possibilities. The problem is even more challenging for embedded/multimedia applications as they demand high throughput with minimal resource consumption. A significant amount of research has been done in the area of generating hardware architectures for different types of applications using both heuristic and constraint programming-based approaches. The research is mainly based on heuristic approaches. We intend to focus on generating area-efficient hardware architectures for streaming applications using constraint programming techniques.

Many embedded/multimedia applications like real-time data processing, mobile computing, and radar processing, require high computational throughput. Such applications perform high degrees of computation and pre-defined architectures may not be able to effeciently exploit the inherent parallelism that these computations offer. FPGAs provide the ability to generate highly customized architectures that can efficiently execute these type of computations. FPGAs have specialized functionality cores such as general purpose processors, multipliers, single and dual port static random access memory (SRAM), and digital signal processing (DSP) slices in order to speedup computation. FPGAs can better exploit the available parallelism to implement the efficient hardware architectures for streaming applications due to the large number of available reconfigurable units.

Streaming applications can be considered as a sequence of operations performed iteratively on different data items. This iterative nature of streaming applications can be used to expose parallelism available across iterations, by creating multiple instances of the same computation. This is commonly referred to as loop unrolling or loop folding. Such tranformation, by revealing the parallelism between loops or iterations, is used to optimize either for latency or throughput of the application. This transformation can be realized automatically or it can be user-driven. When no dependency exists between loop iterations in a fixed computation that is required to be repeated many times, we can achieve higher throughput by replicating the computation with limited impact on the footprint. We can also achieve higher throughput by using pipelined functional units, coupled with a scheduling technique involving the addition of some extra clock cycles beyond the critical path of the single computation. The critical path of a computation is defined as the minimum number of clock cycles that is required by the computation to produce the results. In this thesis we specify the additional clock cycles or slack in terms of the critical path of the application. Additionally, our approach makes use of designer's knowledge about the application by allowing the designer to specify the level of replication and critical path relaxation.

Constraint programming (CP) techniques [1] have been used in the past to perform high-level synthesis (HLS). HLS can be broadly categorized into two tasks: (i) scheduling and (ii) hardware allocation. To achieve an optimal design, both steps should be performed simultaneously. However, due to the NP-complete nature of both steps, most tools perform them separately. Scheduling, according to Gajski [2], is the most important step in HLS. It determines the throughput area tradeoffs of the generated design. To achieve higher throughput a scheduling algorithm attempts to parallelize operations, subject to timing constraints. As mentioned earlier, techniques analogous to loop unrolling in software can be applied to increase exposure to the scheduler, whereby better parallelization decisions can be made. On the other hand, to reduce the area, the scheduler attempts to support functional unit reuse by serializing operations, using pipelined functional units, and relaxing timing constraints. We intend to examine applications, which can tolerate relaxation in the critical path in order to achieve higher computational throughput and reduction in footprint. Pipelined scheduling [3] is a common way of achieving higher computational throughput, by invoking multiple instances of the same sequence of operations. The streaming nature of many applications permits the exploitation of pipelined functional units on an FPGA to achieve higher throughput.

Constraint programming techniques have also been applied to optimal instruction scheduling [4,5]. Instruction scheduling is aimed at determining a minimum length schedule for a given data flow graph (DFG). CP-based schedulers used for instruction scheduling are designed to schedule operations for an already fabricated central processing unit (CPU). Moreover, these schedulers do not take loop unrolling and critical path relaxation into consideration for scheduling the operations. CP-based techniques discussed in this thesis leverage techniques originally proposed for instruction scheduling. We analyze the effects of these additional constraints, when applied to scheduling in data path synthesis. Modified versions of these constraints have been developed to take loop unrolling and critical path relaxation into consideration in order to apply them to simultaneous scheduling and resource allocation.

In this thesis we propose that CP-based optimal instruction scheduling techniques can be applied to facilitate the generation of optimal or near optimal architectures for streaming computations. Using the instruction scheduling techniques, we can implement a tool flow which can generate architectures with improved area and efficiency as compared to force directed scheduling (FDS)-based architectures. We present a modified CHARGER (**C**onstraint based **H**ardware **AR**chitecture **GE**ne**R**ator) tool, which improves the quality of the generated architectures by employing optimal scheduling techniques using constraint satisfaction. The contributions of this thesis include:

1. The integration of a CP-based scheduler to the CHARGER design tool;
2. The evaluation of performance and scalability of the CP-based scheduler against CHARGER's FDS-based scheduler;
3. The analysis of the impact of loop unrolling and critical path relaxation on the CP-based scheduler when provided with different input and output interfaces;

4. The extension of the CP-based scheduler with previously developed and published techniques targeting optimal instruction scheduling, e.g. the determination of regions, dominance constraints, predecessors, and successors constraints;

5. The inclusion and analysis of a heuristic integrated in CHARGER's FDS scheduler into the CP scheduler, and its comparison with CHARGER-generated schedules.

This thesis is organized in the following manner. Chapter 2 presents related work along with work done in the area of instruction scheduling. Chapter 3 discusses finite domain constraints and scheduling algorithms. Chapter 4 discusses the constraint model used in the proposed CP-based scheduler. Chapter 5 evaluates the performance of the CP-based scheduler against an FDS-based scheduler. Chapter 6 proposes a modified pre-synthesis estimation technique and HDL generation to support architecture generation. Finally, Chapter 7 concludes the thesis.

Chapter 2

Related Work

A significant amount of work has been done in the area of scheduling, component allocation, and high-level synthesis. Many tools, with different scheduling techniques, have been developed. Most tools use heuristic approaches for scheduling and allocation. There are tools based on other approaches like integer linear programming (ILP) [6] and constraint programming over finite domain variables [7] for scheduling DFGs. This section discusses some of the approaches that have been used in the past to solve the high-level synthesis problem. Since, the main focus of this thesis is to generate an optimal schedule for a given DFG, this section also discusses some of the techniques that are applied to instruction scheduling.

2.1 Scheduling Using Heuristic Approaches

A number of heuristic scheduling algorithms have been developed to schedule data flow graphs. These algorithms include, but are not limited to, list scheduling (LS) [2], FDS [8,9], genetic algorithms [10], and simulated annealing (SA) [11]. This section discusses some of these algorithms in detail along with the approaches taken for streaming computations.

List scheduling is a resource constrained scheduling algorithm which generates a schedule by prioritizing each node such that a given resource constraint is satisfied. FDS, developed by Paulin and Knight [9], is a time constrained scheduling algorithm developed for deriving hardware architectures from DFGs. FDS is different from other scheduling algorithms in that it simultaneously determines a resource allocation and schedule for a given DFG based on the critical path of the DFG. FDS relies on as soon as possible (ASAP) and as late as possible (ALAP) scheduling algorithms to set lower and upper bound on the start times of each node in the DFG. A node is said to be scheduled when upper bound and lower bound on start time are found to be equal. The algorithm proceeds by iteratively reducing these bounds until all nodes are scheduled. When the scheduling begins, nodes along the critical path will all have scheduled start times, unless slack has been added to the critical path constraint of the DFG. Other nodes, which do not belong to the critical path, have a range of start times as calculated using ASAP and ALAP. The goal of FDS is to produce a schedule that satisfies a given time constraint and at the same time minimizes resource consumption. FDS calculates forces to determine the effect of scheduling each of these nodes at all possible schedule steps. The forces calculated are propotional to the change in number of resources required at all different start times. Since FDS attempts to minimize the number of resources required to schedule the DFG, a node with minimal force is selected and assigned the corresponding schedule step. The process is repeated for all the nodes present in the graph, until all the nodes have been allocated a particular schedule step. Once all the nodes present in the DFG have been scheduled, the number of resources needed are determined by finding the maximum number of operations of each type scheduled at a particular clock cycle. CHARGER uses a modified version of the FDS algorithm to find a schedule for a data flow graph and then generates the hardware architecture based on the schedule.

Sun et al. [3] developed a synthesis methodology which combines module selection and resource sharing for efficient pipeline synthesis. Module selection refers to selection among a variety of circuit implementations for each operation and resource sharing refers to time multiplexing each module. These techniques are applied together to minimize the area of the implementation while maintaining the user-defined throughput constraint. The throughput is defined as, $T_0 = \frac{f}{\delta}$, where f refers to the system clock frequency and δ refers to the initiation interval. The initiation interval specifies the number of clock cycles between the initiation of the sequential iteration of the pipeline. Each module in the circuit is also pipelined and is characterized with two parameters, the first parameter is the module latency, λ_m, which represents the number of clock cycles to complete a single computation and the second parameter is the module initiation interval, δ_m, which represents the number of clock cycles separating the next initiation of the module. They proposed two approaches for combining scheduling, module selection and resource sharing. The first approach, called ASAP Exploration, performs module selection concurrently with resource sharing. This approach uses modulo scheduling with ASAP priority and no backtracking combined with a branch and bound technique for module selection and resource sharing. The second approach, called iterative modulo exploration, performs module selection before resource sharing. This approach uses iterative modulo scheduling [12] to perform pipeline scheduling and resource sharing combined with a greedy module selection. In both the approaches, amount of parallelism exposed to the scheduler is determined automatically through modulo scheduling.

Bellas et al. [13] proposed a template-based automated method to perform architecture exploration for streaming applications. The proposed architectural template has two parts: streaming data path and the stream unit. The stream unit provides an interface to fetch/deliver data from/to system memory or peripherals through a system bus. The data path template is an interconnect of reconfigurable functional units that produce and consume streaming data, and communicate via reconfigurable links. The main components of their proposed tool include a common template-based architecture for streaming applications, an iteration engine to instantiate system parameters that meet system and user defined constraints, a scheduler which performs scheduling and hardware allocation, and an RTL constructor engine which produces synthesizable Verilog code. The scheduler takes the streaming application, represented as DFG, and user specified unroll factor as inputs. It uses modulo scheduling to overlap multiple iterations in each cycle and exploits maximum available parallelism under the resource constraint and data dependencies. The scheduler calculates the minimum initiation interval (MII) based on the number of available resources and data dependencies between iterations. The scheduler also binds operational nodes to functional units and generates delay registers to store intermediate data. RTL constructor is then used to generate a Verilog code for data path and stream unit.

2.2 Scheduling Using Constraint Programming Techniques

The main focus of this thesis is the use of constraint programming techniques for scheduling DFGs and derivation of custom architectures based on the generated schedules. Constraint programming techniques have been used in the past in many high-level synthesis tools to find optimal resource allocations and schedules. In this section, we evaluate some of these techniques used in high-level synthesis and instruction scheduling for single- and multi-issue processors.

In the constraint programming approach, the formulation of the problem into a constraint model plays a significant role in determining the performance and scalability of the scheduler. Most CP-based schedulers suffer from long execution times and are applicable only to small DFGs. Some ILP approaches have also been studied in the context of high-level synthesis by many researchers. Hwang et al. [6] used ILP to formulate time constraint, resource constraint and feasible scheduling problem. A set of constraints were formulated using ILP and cost functions were used to minimize functional units and scheduling steps. Let C_t denote the cost of a functional unit of type t (F_t), and there are m functional units. M_t is an integer variable denoting the number of functional units of type t and s denotes the total number of control steps. x_{ij} is an 0-1 integer variable associated with an operation O_i where $x_{ij} = 1$ if O_i is scheduled at control step j; otherwise, $x_{ij} = 0$. The earliest possible start time (ASAP) and latest possible start time (ALAP) of O_i are S_i and L_i, respectively. Assuming one cycle propagation delay and non-pipelined functional units scheduling problem was formulated as follows:

$$\sum_{o_i \in F_t} x_{ij} \leq M_t, \quad 1 \leq j \leq s, 1 \leq t \leq m, \tag{2.1}$$

$$\sum_{j=S_i}^{L_i} x_{ij} = 1, \quad 1 \leq i \leq n, \tag{2.2}$$

$$\sum_{j=S_i}^{L_i} (j * x_{ij}) - \sum_{j=S_k}^{L_k} (j * x_{kj}) \leq -1. \tag{2.3}$$

The cost function is a combination of time-constraint objective function, which minimizes number of functional units, and resource-constraint objective function, which minimizes scheduled step. The cost function does not find an optimum solution but determines whether a feasible solution exists. Using formulations described in (2.1), (2.2), (2.3) and the provided cost function, a feasible solution to the scheduling problem is determined.

Ahmed and Abdel-Malek [14] proposed a hybrid approach using ILP and constraint logic programming (CLP) for scheduling DFGs in high-level synthesis. The hybrid model is based on the fact that some of the constraints are better treated as ILP constraints, and others are better treated as CLP constraints. Their proposed model is very similar to one proposed by Hwang et al. [6], but uses a decomposition algorithm, which first solves ILP model to get a partial solution, and then constructs a feasibility model, which is then solved using CLP, to generate the final solution. In the both the papers, discussed above, results were shown only for a fifth-order wave filter, so it is not clear how these formulations would behave for large DFGs with loop folding and pipelined functional units.

Kuchcinski [7] makes use of constraint programming over finite domain variables for scheduling and resource allocation in high-level synthesis. His model supports both resource and time constraint scheduling with different optimization criteria. To find an optimal solution to the scheduling and resource allocation problem, a JaCoP (**Ja**va **Co**nstraint **P**rogramming) constraint solver is implemented which uses constraint consistency and entailment methods to keep the constraint store consistent while searching for a solution.

Both functional and non-functional requirements of the system were modeled using a set of constraints. Several basic constraints like task precedence, resource sharing along with some combinatorial constraints like element, cumulative, and diff2 were selected and implemented. They extended the solver to include special constraints like pipelined functional units, chaining and replication through functional pipelining. The scheduler performance, for HLS, was evaluated using finite impulse response (FIR), fifth-order elliptic wave filter (EWF), AR lattice filter (AR), and discrete cosine transform (DCT). Each example was scheduled using multicycle and pipelined components, chaining and functional pipelining (two or three stages). The proposed approach does not support architecture generation as all the results shown were targeted for a pre-defined architectural design. Also, the data flow graphs that were considered in their experiments were small.

ILP and constraint programming techniques have also been applied to instruction scheduling for generating optimal schedules for DFGs modeling instruction sequences which target single- and multi-issue microprocessors. The main objective of instruction scheduling is to a find a minimum length schedule for a basic block subject to precedence, latency, and resource constraints. A basic block is a sequence of instructions with a single entry point and single exit point. Wilken et al. [15] developed a robust scheduler, using ILP, that could solve large basic blocks. They showed the results for basic blocks using basic ILP formulations and a set of DFG transformations. They used DFG partitioning and introduced regions to develop a robust scheduler which could solve large basic blocks. Their approach, however, only targets single-issue microprocessors. Van Beek and Wilken [4] improved on those results, using constraint programming techniques.

Malik et al. [5] have extended this approach to multi-issue microprocessors, implementing the algorithm using constraint programming techniques. They developed a optimal scheduler which is robust and can schedule large basic blocks (consisting of $\approx$ 2600 instructions) using a set of graph transformations. According to Malik et al. [5], the application of distance constraint in regions, predecessor, and successor constraints, and dominance constraint were found to be essential in improving the efficiency of the search for a schedule. The proposed optimal scheduler tries to find a schedule for a given set of resources within the critical path latency of the DFG. If no solution is found, then scheduler increases the latency by one clock cycle and again tries to find a schedule for the given DFG within the new latency constraint. The process is repeated until either a schedule is found or time limit is reached. They experimentally evaluated the optimal scheduler on SPEC2000 integer and floating point benchmarks.

Since, instruction scheduling is aimed at finding a minimal length schedule, instruction schedulers do not take loop folding and critical path relaxation into consideration when generating an optimal schedule for the basic blocks. Moreover, the scheduler generates the minimal length schedule for pre-defined CPUs. The architecture designs that instruction scheduling targets have fixed set of resources and a pre-defined instruction issue width. We intend to use some of the optimal instruction scheduling techniques, specified in constraint model proposed by Malik et al. [5] and analyze the effect of these additional constraints, when applied to scheduling, in data path synthesis.

2.3 High-Level Languages to HDLs

The proposed constraint-based scheduler is an integral part of a tool flow. In this section, we briefly discuss some of the existing hardware design tools and the approach taken to carry out the architecture generation. There has been intense research done to automate the architecture generation process for FPGAs directly from high-level languages like C, Matlab, Simulink, etc. Some of the tools include, but are not limited to, Handel-C [16], Mitrion-C [17], Impulse-C [18], and PACT HDL [19].

PACT (Power Aware Architecture and Compilation Techniques) HDL [19], as the name suggests, was developed to create power aware architectures. It is a C to HDL compiler which merges architecture generation of high-level algorithms with power and performance optimizations. It performs the architecture generation process in three stages. In the first stage, C code is converted to high-level C type abstract syntax tree (AST). In the second stage, C AST is converted to a finite state machine (FSM) type HDL AST based on the target architecture specifications. Finally, it generates a synthesizable VHDL and verilog code for the architecture. While their approach was sufficient and effective, it does not make use of any heuristic scheduling algorithms, apart from priority based ASAP and ALAP scheduling, to optimize hardware architectures.

Handel-C [16] is a complete emulation of C language except for some complex elements like structure, pointer, etc. It synthesizes user-provided C code to FPGAs. Handle-C extends C with some addition constructs to support timing and parallelism. The compiler translates input Handel-C code to an abstract syntax tree, which is then used to generate a high-level netlist that contains coarse function blocks. Handel-C then optimizes the high-level netlist, which it then compiles to the FPGA bitstream. The major drawback of Handle-C is that the user is required to specify the parallel portions of the code. Also, the tool does not support loop unrolling which is commonly used for exposing parallelism.

ImpulseC [18] is a C-based development system for coarse-grained programmable hardware and targets FPGA platforms. It can process blocks of C code and convert them into VHDL or Verilog hardware descriptions. It performs scheduling with support for optimizations like pipelining and loop unrolling. The tool includes C-compatible functions that let the designer specify parallelism using streaming, shared-memory, and multi-process programming models. It also includes some platform specific packages to simplify the C-to-HDL process.

Mitrion-C [17] represents a new approach for software programmability for FPGAs. It uses the Mitrion virtual processor which is a massively parallel high-performance processor for FPGAs that executes software written in the Mitrion-C programming language. Mitrion-C complements the processor's fine-grained parallelism by providing a fully parallel programming language. Mitrion-C's processing model is based on data dependecies rather than on a sequential execution model. This provides the ability to extract maximum parallelism available in the applications to be mapped on FPGAs.

There are tools which use Matlab [20] and Simulink [21] representation to generate HDL. These tools are discussed in detail by Areno [22].

Most approaches, discussed in this chapter, that are used in HLS do not support generation of custom hardware architectures (except Bellas et al. [13]). These approaches target fixed architecture designs for scheduling and hardware allocation, where number and type of resources are known beforehand. These approaches focus on the derivation of a schedule which satisfies the given architecture specifications rather than on the derivation of a custom architecture which can effeciently execute the given application. Thus, it is very difficult to extend these approaches to target a slightly different architecture. This is because these approaches are based on heuristic methods which are usually designed for a particular problem. It is very difficult to extend them to a slightly modified problem.

Constraint programming approaches are better in handling different constraints, but current tools either do not facilitate custom architecture generation or generate template-based architectures. FPGAs provide the facility to derive custom architectures which can best execute the application.

Chapter 3

Background

In this chapter, we introduce the concept of finite domain constraints, Oz programming language [23], and Mozart programming system [24]. We also discuss briefly the formulation of scheduling and resource allocation using finite domain constraints. In constraint programming, a system is modeled using a set of constraints over the finite domain variables, where a constraint is simply a relationship among several variables, each taking a value in its given domain. The two basic phases of constraint programming are (i) propagation, and (ii) distribution. The propagation phase gathers information from all the constraints and accumulates that information in a data structure called the constraint store. In the distribution phase the problem is split into complementary cases, each of which is independently solved. By iterating between propagation and distribution, each finite domain variable will be eventually bound to a singleton value, which determines the solution to the problem. Alternatively, the solver will prove that no satisfactory binding of values to variable exists, proving the constraints are too tight for a solution to exist.

3.1 Finite Domain Constraints

Constraint programming is a methodology to solve combinatorial problems. In our approach, we use finite domain constraints to model the scheduling problem. We first introduce a constraint satisfaction problem (CSP) using finite domain constraints and then discuss the formulation of the scheduling problem in terms of these constraints.

According to Kuchcinski [7], CSP is a 3-tuple, $S = (V,\ D,\ C)$, where
$V = \{x_1, x_2, ..., x_n\}$ is a finite set of variables also known as finite domain variables,
$D = \{D_1, D_2, ..., D_n\}$ is a finite set of domains, and
$C = c_1 \wedge c_2 \wedge c_3, ..., \wedge c_n, \quad n \geq 0.$

Constraints $c_i, i \leq n$ are called primitive constraints. C is said to be satisfied only when each primitive constraint in C is satisfied. A constraint satisfaction problem seeks to determine whether C is satisfiable. A constraint is a formula of the predicate logic. Constraints are defined by equations, inequalities, and global constraints. Finite domain constraints are constraints in which the value set defined by the domain is a finite set of non-negative integers. We use the notation $T = \{n\#m\}$ to define finite domain $n, ..., m$ for finite domain variable T. Here are some typical examples of constraints occurring with finite domain problems.

$$X = 67,\ X \in 0\#9,\ X = Y,\ \ X^2 + Y^2 = Z^2,\ X + Y + Z < U,\ X + Y \neq 5 \cdot Z \tag{3.1}$$

The Mozart Programming System and constraint solver facilitates the development of various applications in the Oz language. Oz is a multi-paradigm language designed for concurrent, networked, and reactive applications. It has been extended to support constraint programming with finite domain constraints. The Mozart Programming System provides a development environment for Oz and offers compiler/linker, debugger, and runtime support for Oz-based applications.

The Mozart Programming System solver facilitates the determination of the finite domain constraint problem through propagation and distribution. Once all the constraints are defined, the solver attempts to narrow the domains of all finite domain variables in the problem. All the computations occur in a space, which consist of a number of propagators connected to a constraint store, as shown in fig. 3.1. The constraint solver, through propagation and distribution, tries to find a solution to the constraint problem by binding values to the variables present in the constraint store. The constraint store stores the set of all the basic constraints with their current domains. An example of a constraint store is shown in (3.2).

$$X \in 0\#5 \wedge Y = 8 \wedge Z \in 13\#23 \tag{3.2}$$

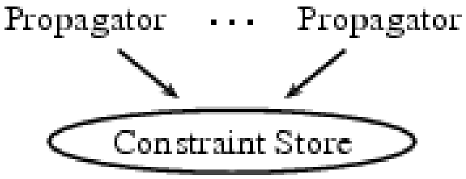


Fig. 3.1: Constraint store.

A finite domain constraint is a relation between finite domain variables and is composed of a set of basic constraints. Each finite domain variable is associated with a domain which is a subset of the constraint domain. Consider a finite domain constraint $x < y$, where the variables are defined such that $x \in \{23\#100\}$ and $y \in \{1\#33\}$. The solver narrows the domain of x and y to $x \in \{23\#32\}$ and $y \in \{24\#33\}$. Whenever the solver determines a change in the variable's domain, the constraint store is updated with the new domain. This process is called constraint propagation.

In Mozart, this process is accomplished through a propagator. A propagator for a constraint C is a concurrent computation agent that tries to narrow the domains of the variables occurring in the constraint C. The propagator updates all the changes of a finite domain variable in the constraint store. The constraint store then propagates the changes to all other finite domain variables involved in the constraint. If two or more propagators share a common finite domain variable, they communicate through the constraint store. As soon as a space is created, propagators tell the basic constraints to the constraint store. Constraint propagation alone is not sufficient for a constraint solver to find complete solution. Propagation continues until the space becomes stable, i.e., now new information can be obtained to further reduce the domains of the variables.

If the space is stable, but not solved, we choose a constraint C and distribute the space with respect to constraint C. This distribution yields two complementary spaces, one with the propagator for C and one with the propagator for $\neg$ C. The distribution phase solves the finite domain problem using backtracking. Typically C is selected by selecting an unbound variable and binding that variable to a value from its domain. This constraint is inserted to the space while $\neg$ C is inserted into a copy of the space. The two spaces are then solved independently, under the assumption that only a single solution exists, and only one of these spaces will yield a solution. The other will result in a contradiction. The distribution can be recursively applied to find the best solution. The solution to a space is an assignment of values to all the variables in the constraint problem such that all the constraints are satisfied. The distribution phase terminates either when a solution to the space is found or a contradiction is encountered indicating that no solution exist.

There could be multiple solutions that satisfy the defined constraints. The optimal solution is the one which satisfies the cost function. For the scheduling and resource allocation problems, an optimal solution is the one which either maximizes or minimizes the given cost function. In our approach, we try to find a schedule which has minimum WSDP.

3.2 Formulation and Scheduling

3.2.1 Data Flow Graph Representation

To use the constraint programming approach for generating an optimal schedule and hardware architecture, the system must be formulated using a data flow graph representation, which is a directed graph G=(N,E), where G is the data flow graph, N represents a set of all nodes present in the DFG, and $E \subseteq N \times N$ represents a set of directed edges connecting two nodes. Each node in the DFG represents an operation and is associated with a non-negative integer value which represents the latency of the node. Edges represent the precedence relationships that exist between node i and node j. The precedence relationship ensures that node j is not issued before node i has executed for l clock cycles. The critical path distance from node i to node j in a DFG is the maximum sum of all the latencies along all the paths from i to j. A node i is a predecessor of node j, if there exists a direct path from i to j; vice versa node j is successor of node i if there exists a direct path from i to j. The nodes in the DFG without any predecessors are called Root Nodes. The nodes in the DFG without any successors are called Leaf Nodes. A simple data flow graph is shown in fig. 3.2.

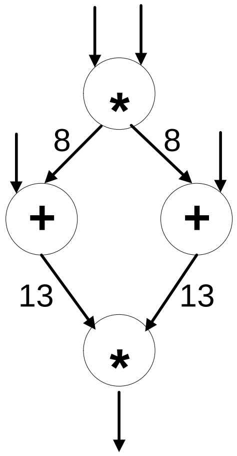


Fig. 3.2: A simple data flow graph.

3.2.2 Oz-Based Scheduler

Scheduling, in the context of this thesis, is the process of simultaneously determining issue time for each node in a provided data flow graph, as well as the number of functional units required to execute that graph. We assume that the graph models a streaming application, requiring the repeated application of the graph across a large set of input data. Consequently, we apply loop unrolling and critical path relaxation to aid in the architecture generation process. We extend the formulations proposed by Malik et al. [5] for basic block instruction scheduling in our Oz-based scheduler to simultaneously determine optimal resource allocation and scheduling for generating hardware architectures for streaming applications. In particular, we analyze the effect of regions, dominance constraints, predecessor and successor constraints on scheduler performance, subject to different input and output interfaces, in generating an optimal schedule based on user specified replication and critical path relaxation.

3.3 CHARGER Tool Flow

CHARGER offers an integrated tool for modeling, scheduling, resource and memory allocation, binding and HDL generation. Figure 3.3 illustrates the CHARGER toolflow. CHARGER takes as input a DFG representation of the application, architecture description, and amount of replication and critical path relaxation. The architecture description charaterizes the available functional units, provides total available area and platform specific information. This information is then passed to a modified FDS scheduler where the DFG is replicated to the specified degree. The FDS scheduler determines the number of functional units as well as a schedule for the operations targeting these functional units. The schedule indicates the start times for each operational node present in the replicated DFG. The product of the FDS scheduler does not represent a complete hardware architecture design. CHARGER facilitates the creation of a hardware architecture from a DFG based on the result produced by FDS scheduler. Once scheduling and resource allocation is done, CHARGER deals with memory allocation and binding. Memory allocation examines the FDS generated schedule and addresses inter-unit data communication. Binding determines which instance of a particular functional unit will execute which operation in the schedule. When CHARGER is finished with memory allocation and binding phase, an infrastructure is created that is required to generate the necessary VHDL code. This infrastructure generates all the files necessary to develop an instance of the architecture in Xilinx ISE.

This thesis extends the existing CHARGER tool by including constraint-based scheduler and Verilog generator. The constraint-based scheduler uses an Oz-based constraint solver to schedule a given DFG based on the level of replication and critical path relaxation provided by user. The resulting schedule and resource allocation is then used by CHARGER to generate the verilog files that can be used to instantiate the architecture. CHARGER tool flow is shown in fig. 3.3.

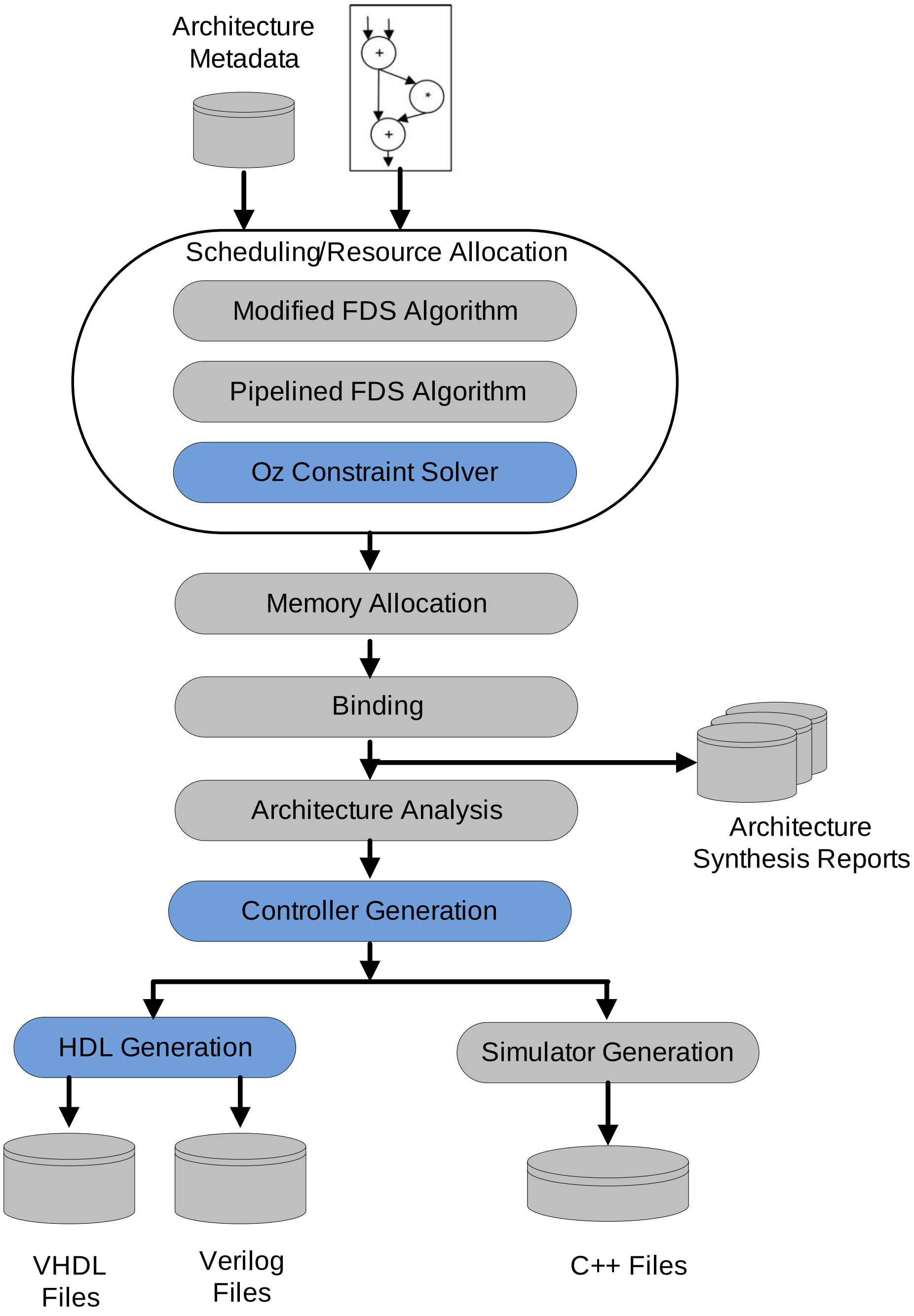


Fig. 3.3: Proposed modifications to CHARGER.

Chapter 4

Constraint Model for Scheduling and Resource Allocation

Scheduling is considered as the most important step in a data path synthesis. This chapter discusses the constraint programming approach used to model the scheduling and resource allocation problem. In constraint programming, we model a given data flow graph using a set of constraints. We start by listing all the constraints used to model a given application represented in the form of data flow graph (DFG). The constraints used to model a DFG includes the latency constraint, area constraint, precedence constraints, distance constraints, predecessor and successor constraints, dominance constraints, and priority constraints. In order to compare our CP-based approach with FDS, various versions of the CP-based scheduler were developed using different combinations of these constraints, adding a heuristic to the scheduler and by providing different input/output interfaces. We added thesed different constraints to improve the performance and scalability of the Oz-based scheduler. In the following section we describe, in detail, all the constraints that were used in the constraint model.

As mentioned earlier, CHARGER takes architecture metadata as one its inputs. This metadata takes the form of an architecture description file, defining a family of resources and their types. Each resource is characterized by latency (in terms of clock cycles) and area (in terms of device primitives). We assume that resources are fully pipelined and each node in the DFG is associated with a particular resource type. The latency of the resource type associated with a node defines the latency of that node. Some of the notations that are used to specify constraints are summarized in table 4.1. The notations are taken from Malik et al. [5].

Table 4.1: Notations used for specifying constraints.

lat(i)	latency of node i
st(i)	start time of node i
type(i)	type of operation associated with node i
pred(i)	set of all immediate predecessors of node i
succ(i)	set of all immediate successors of node i
cp(i, j)	critical-path distance between node i and j
k_t	number of functional units of type t
onpath(i, j, t)	set of all nodes that are on some path from node i to node j. node $i \in$ *onpath(i, j, t)* if *type(i)* $= t$, and node $j \in$ *onpath(i, j, t)* if *type(j)* $= t$.

4.1 Latency Constraint

The latency constraint represents the time step by which all the leaf nodes must produce an output. Given a labeled DFG G=(N,E), where N represents a set of all nodes presents in the DFG and $E \subseteq N \times N$ represents a set of all edges connecting two nodes, a latency constraint as shown in algorithm 4.1 is applied to all the leaf nodes present in the DFG. The latency constraint reduces the possible start times of nodes by establishing upper bounds on the start time variables.

Algorithm 4.1 Latency Constraint

```
let $leaf \subseteq N$ be the set of all leaf nodes
let $latency \in \mathbb{Z}^+$ be the critical path of the system

$\forall n \in leaf:$
        $st(n) \leq latency - lat(n)$
```

4.2 Resource Constraint

To generate an optimal schedule for a given DFG, the number of resources of each type of operation should be known either beforehand or must be determined during scheduling. Most approaches schedule the DFG based on a user provided set of resources. In our approach we try to simultaneously determine the start time for each node in a provided DFG, and the number of functional units of each type required to execute the DFG. To determine the number of resources of each type simultaneously with scheduling, the constraint solver determines an lower and upper bound on each type of resource based on the replication index, critical path relaxation, and the area constraints imposed by the designer. To apply a resource constraint, a finite domain variable for each type of resource is created by the Oz scheduler which indicates the number of resources for that type to be instantiated to schedule the DFG. Constraints must be applied in order to ensure that number of resources required to support parallel operation issue is consistent with the maximum issue concurrency for each operation type in the schedule. Such constraints are a special case of a well-studied constraint called the global cardinality constraint [25]. A global cardinality constraint (GCC) over a set of variables states that the number of variables instantiating to a value must be between a given upper and lower bound. This constraint has been used in instruction scheduling, to ensure that the number of instructions of each type on a given clock cycle should not exceed the available functional units of that type [5]. When GCC is applied to the instruction scheduling problem, all the lower bounds are set equal to zero and upper bounds are set equal to number of functional units of each type.

Mozart does not provide a built-in GCC. A custom constraint is implemented which is used to apply the GCC for each type of functional unit. In our case the number of resources of each type is a finite domain variable determined by the Oz scheduler. We have modified the GCC to permit the available resources of each type to be defined as a finite domain variable, instead of a pre-defined grounded integer. This modified version of the GCC has been implemented by Christopher Crapo at Utah State University. The GCC is applied during both propagation and distribution.

4.3 Precedence Constraint

The precedence constraint, in addition to the latency constraint, imposes a lower bound on the start time of each node present in the DFG. Given a DFG $G = (N, E)$, for a pair of nodes i, j such that $(i, j) \in E$, a precedence constraint of the form, as shown in algorithm 4.2, is added to the constraint model. This constraint assumes that once the functional unit completes an operation, the result is available for consumption by the subsequent functional unit. The communication infrastructure between the subsequent functional units is assumed to not add any delay after the result is produced. The hardware implementation of this communication network will be discussed later in detail.

Algorithm 4.2 Precedence Constraint

```
$\forall j \in N :$
    $\forall i \in pred(j) :$
        $st(j) \geq st(i) + lat(i)$
```

4.4 Priority Constraint

Streaming applications require a fixed computation sequence to be performed iteratively on different data items. The proposed approach allows the user to specify the level of loop unrolling or replication to consider. The constraint solver replicates the DFG, based on the user provided replication index, prior to scheduling. All the nodes present in the original loop body are replicated and each replicated node is associated with a iteration number, which denotes the priority of that node. The priority constraint, as shown in algorithm 4.3, creates a partial order on nodes present in different iterations. The priority constraint ensures that nodes in the later iterations can not execute prior to the nodes in earlier iterations. This eliminates potential symmetries in the constraint problem.

Let R be the amount of loop unrolls provided by the designer. The constraint solver then replicates the DFG and a creates new DFG, $G_R = (N_R, E_R)$, consisting of R instances of the original DFG G.

4.5 Distance Constraint

The distance constraint is applied between the source node and the sink node of subgraphs called Regions [15]. A pair of nodes i, j in a DFG define a region if there exists

Algorithm 4.3 Priority Constraint

```
let $Pri : N_R \rightarrow \{1, 2, 3, ..., R\}$ returns the priority of a node
$\forall n \in N$: let $rep_n \subset N_R$ be the set of nodes replicated from $N$ during unrolling
$\forall i, j \in rep_n$ :
            if $Pri(i) < Pri(j)$
                $st(i) \leq st(j)$
```

more than one path between node i and node j and there does not exist a node k such that every path between i and j goes through k.

Given a DFG $G = (N, E)$, for each pair of nodes $(i, j) \in N$, which forms a region, the distance constraint as shown in (4.1) is added to the constraint model.

$$st(j) \geq st(i) + d(i,j) \tag{4.1}$$

Regions were first introduced by Wilken et al. [15] for instruction scheduling on a single-issue processor using ILP. They applied a graph transformation technique called region linearization which orders the set of nodes contained within a region. Region linearization is used to derive a sequential schedule for the nodes contained within a region.

Van Beek and Wilken [4] proposed a constraint programming approach to instruction scheduling for a single-issue processor. In their proposed constraint programming model, regions were identified and a distance constraint was applied between the source node i and sink node j. Distance constraints restricts the lower bound on the number of clock cycles that must elapse between when node i is scheduled and when node j is scheduled. Distance constraints can be an improvement over latency constraints because latency constraints consider only a single path when calculating distance between source node and sink node. Whereas, distance constraints look at all the possible paths from the source node to the sink node and can determine additional information about the number of nodes that need to be scheduled before the sink node, thus can improve the distance between source node and sink node.

Malik et al. [5] extended the constraint model prosposed by Van Beek to a multi-issue processor. Consider the notations described in table 4.2. The distance is estimated between source and sink node using (4.2).

$$d(i,\ j) = MAX\{r_1(i,j,t) + r_2(i,j,t) + r_3(i,j,t) - 1\} \tag{4.2}$$

We implemented the distance constraint in our constraint solver. The constraint takes a slightly different form in our implementation as it is applied to an as-yet undefined width processor. The difference in our implementation is when the distance constraint is applied, the number of functional units of each type is a finite domain variable. The distance constraint proved to be very useful for instruction scheduling on a multi-issue processor because the processor, although able to issue multiple instructions, had only one functional unit for each type of instruction. When there is only one functional unit present for each type of instruction and more than one operations of a particular type needs to be performed within a region, it improves the distance between the source node and the sink node.

In our approach, the distance constraint is applied as a lower bound on the number of clock cycles that must elapse between the issue of the source and the sink node. We have experimentally found that the application of distance constraint, in our case, never obtains an improvement over latency constraint. When we estimate optimal distance for a

Table 4.2: Additional notations used for specifying the distance constraints.

$r_1(i,\ j,\ t)$	The minimum number of cycles that must elapse before the first operation in $onpath(i,j,t)$ can be issued; i.e., $\min\{cp(i,k) \mid k \in onpath(i,j,t)\}$, the minimum critical-path distance from node i to any node in *onpath(i, j, t)*.
$r_2(i,\ j,\ t)$	The minimum number of cycles to issue all of the operations in $onpath(i,j,t)$; i.e., $\lceil \lvert onpath(i,j,t) \rvert / k_t \rceil$, the size of the set of operations divided by the number of functional units that can execute instructions of type t, rounded up to the next highest integer value.
$r_3(i,\ j,\ t)$	The minimum number of cycles that must elapse before the last operation in $onpath(i,j,t)$ is issued and node j can be issued; i.e., $\min\{cp(k,j) \mid k \in onpath(i,j,t)\}$ the minimum critical-path distance from any node in $onpath(i,j,t)$ to node j.

region using (4.2), it provides the same lower bound on the distance as imposed by latency constraint.

The determination and analysis of regions still proves to be useful in our approach in reducing the size of search space. CHARGER discovers the regions present in the provided DFG. The constraint solver then replicates the graph including regions, based on the user-provided replication index. The priority of each region is determined by the priority of the source node. A schedule for the nodes within each region is determined in isolation, in order to find the optimal distance between source node and sink node. In the distribution phase, regions are sorted based on priority and the highest priority region is selected. Once a region is selected, the constraint solver creates a subspace to distribute on the selected region. In the subspace, constraint solver re-creates the full DFG and applies all precedence constraints and resource constraints. The solver then determines a schedule for the nodes present in the region. Since precedence and resource constraints are taken into consideration when distributing on the subspace, the resulting schedule adheres to the correctness criteria for the global schedule. Once the distance between source and sink node is determined in the subspace, a distance constraint is posted in the parent space, allowing propagation to proceed. This process is repeated for all the regions present in the replicated DFG.

Once a schedule is determined for a region, each node in the region is bound to a particular start time and it is then merged with the main space to determine the schedule for the entire DFG. This merging leads to a new ordering and time bounds on the rest of the unscheduled nodes in the DFG, i.e., strengthened earliest start times and latest start times for the nodes and thus reducing the search space and total distribution steps. The only drawback of distributing on regions is for DFGs with small replication index and critical path relaxation, it takes slightly more time to determine an optimal schedule. This is due to the fact that for each region distribution, the entire DFG is recreated to find a schedule which adds overhead. But for DFGs with large replication index and critical path relaxation, region distribution can help in reducing the search time and size of the search space.

4.6 Predecessor and Successor Constraint

Given a DFG $G = (N, E), \forall i \in N$ which has more than one immediate predecessor and successor, a single predecessor and successor constraint of the form described in algorithm 4.4 is added to the constraint model, where $\lceil x \rceil$ returns the smallest integer value not less than x and $P(N)$ represents the power set of N. Predecessor and successor constraints can be considered an adaptation of edge finding rules [26]. The edge-finding technique consists of deducing a new ordering relations in the graph and new time bounds. It helps in improving the lower bound of the domain of the node's start time. The predecessor and successor constraints are adapted from Malik et al. [5] and modified slightly to fit in our approach. The difference is in our approach, number of functional units of each type is an unbound finite domain variable when predecessor and successor constraints are applied.

Similar to distance constraints, predecessor and successor constraints can derive a stronger relationships during distribution. This is because once the constraint solver is finished distributing on the resources, the finite domain variable associated with each type of functional unit is bound to a particular value. Using these values in algorithm 4.4, reduced start time domains of the nodes can be obtained. The solver then starts distributing on the start times of the nodes to determine a valid schedule for the DFG that satisfies the given latency constraint. Predecessor and successor constraints are applied before the

Algorithm 4.4 Predecessor and Successor Constraint

```
let $pred(i,t) : N \rightarrow P(N)$ be the set of all immediate predecessor nodes to node $i$ of type $t$
let $succ(i,t) : N \rightarrow P(N)$ be the set of all immediate successor nodes to node $i$ of type $t$
let $st_{lower}(i) : N \rightarrow \mathbb{Z}^+$ be the lower bound of the domain of the start time of node $i$
let $st_{upper}(i) : N \rightarrow \mathbb{Z}^+$ be the upper bound of the domain of the start time of node $i$


$\forall i \in N$:
    $\forall k \in pred(i,t)$
            $st(i) \geq st_{lower}(k) + \lceil |pred(i,t)| / k_t \rceil + lat(k) - 1$

    $\forall l \in succ(i,t)$
            $st(i) \leq st_{upper}(l) - \lceil |succ(i,t)| / k_t \rceil - lat(l) + 1$
```

constraint solver start distributing on start times.

Example: Consider the partial DFG shown in fig. 4.1, where the domains of the start time variables are as shown. The domains of the start time variables obtained by the application of a latency constraint are shown next to the associated node; assuming that constraint solver has selected a single instance of each type of resource.

Applying predecessor and successor constraints improves the bounds of node A, I, and J. The earliest that one of the predecessors of node I can be scheduled is at cycle 27, therefore using algorithm 4.4, the earliest that node I can be scheduled is 36. In the similar manner, the lower bound of node J can be raised. As well, using a successor constraint upper bound of node A can be reduced to 3. Predecessor and successor constraints strengthen the domains of the start time variables and thus reduce the distribution steps taken by the constraint solver to find an optimal schedule.

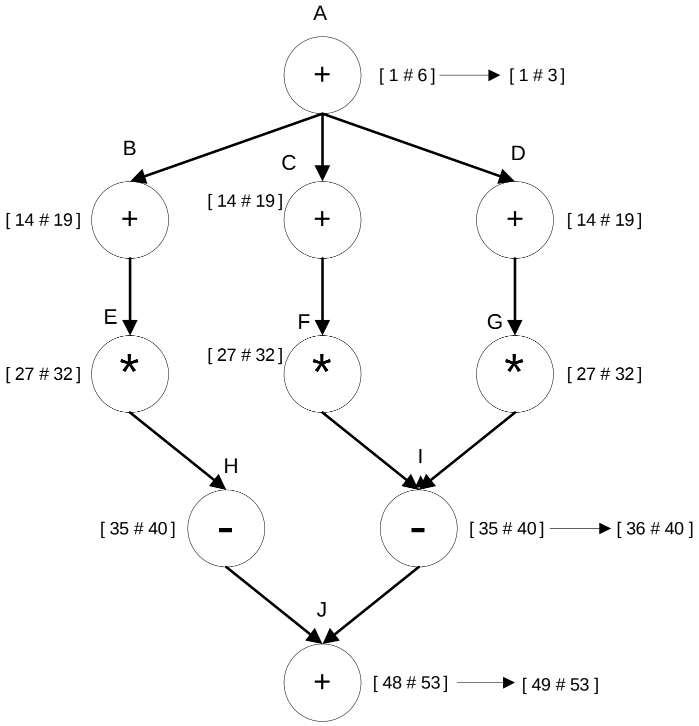


Fig. 4.1: Example of improving lower and upper bound of start time variable using predecessor and successor constraints.

4.7 Dominance Constraint

A set of graph transformations were proposed by Heffernan and Wilken [27] for DFGs and proved that these transformations pruned redundant, and inferior schedules, making the scheduler faster and robust while preserving the optimality of the generated schedules. Since their approach targeted a single-issue processor, these transformations simply add a latency zero edge between a pair of nodes which are independent. A pair of nodes i and j are independent if node i is neither a predecessor nor a successor of node j. These graph transformations can prove to be useful in constraint programming. In the context of constraint programming, under certain conditions detailed below, these transformations add a simple constraint of the form shown in (4.3), known as a dominance constraint [5].

$$st(i) \leq st(j) \tag{4.3}$$

We are interested in finding a pair of independent, isomorphic nodes i and j so that a dominance constraint can be applied. Nodes i and j are isomorphic, if they are identical, i.e., both the nodes are associated with same resource type. In our approach, we focus on regions to find isomorphic nodes. Under the following conditions, adding a constraint described in (4.3) does not eliminate the optimal solution.

1. node i and node j are independent
2. $type(i) = type(j)$
3. $\forall k \in pred(i),\ lat(k) \leq\ cp(k, j)$
4. $\forall l \in succ(j),\ lat(j) \leq cp(i, l)$

In our approach, we determine a pair of independent and isomorphic nodes inside a region and apply dominance constraints. Given a region defined by nodes i and j, a depth-first search is performed to determine the separate components of the regions. The main idea behind applying dominance constraints is to remove the similarities present the in a DFG.

Dominance constraints are stronger than precedence constraints because precedence constraints determine a partial order on the nodes that are on the same critical path while dominance constraints determine a partial order on the nodes that belong to different critical paths. Dominance constraints are similar to priority constraints discussed earlier in sec. 4.4. The difference is that dominance constraints create a partial order on the nodes within a single iteration while priority constraints determine a partial order on the nodes occuring in different iterations.

Example: Consider a subgraph within a DFG as shown in fig. 4.2. We can apply the dominance constraints between independent nodes as follows. From fig. 4.2, it can be seen that nodes B and C are isomorphic and satisfy all the conditions described above. Hence, the constraint $st(B) \leq st(C)$ can be added to the constraint model. Similarly, nodes D, E, and F, E are isomorphic and satisfy the conditions and therefore, constraints $st(D) \leq st(E)$ and $st(F) \leq st(E)$ can be added to the constraint model.

Although these constraints were originally developed for processors with predefined issue width, they are still applicable in our approach where the issue width of processor is not yet determined. Consider again the example shown in fig. 4.2. Applying GCC will constrain the number of functional units required to execute the DFG. For the DFG,

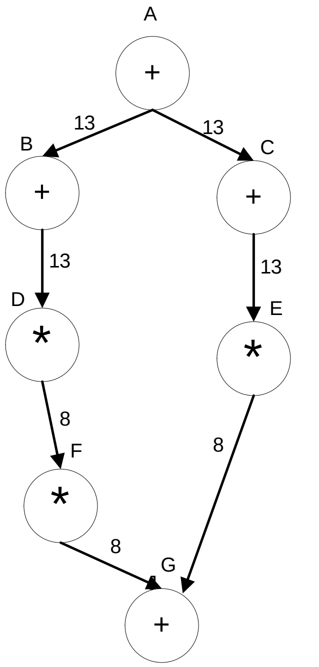


Fig. 4.2: Example of adding dominance constraint.

given below, the finite domain variables used for adders and multipliers will have domain [1#2]. Depending on the latency constraint imposed by the designer, if the constraint solver determines that the optimal schedule requires one instance each of an adder and a multiplier, then $st(B) < st(C)$. Similarly, $st(D) < st(E)$ and $st(F) < st(E)$. If the solver determines that the schedule requires two adders and two multipliers, then $st(B) = st(C)$, $st(D) \leq st(E)$ and $st(F) \leq st(E)$. Thus, adding dominance constraints to the constraint model do not eliminate the optimal solution.

4.8 Area Constraint

The area constraint represents the maximum available area of the target FGPA. As we focus on architecture generation for streaming applications with user-specified replication level and critical path relaxation, some of the designs might not be feasible on the target FPGA. FPGAs consist of a number of device primitives (LUTs, FFs, DSP48s, and BRAMs). To ensure that resulting hardware architecture satifies area requirements, an area constraint of the form shown in algorithm 4.5 is added to the constraint model.

For a given latency constraint, there could be multiple designs that satisfy an area constraint. Therefore to compare two designs and select the best among them, the weighted sum of device primitives (WSDP), a metric to determine the relative cost of a resource, is used in our approach. This metric was proposed by Phillips [28] and refined by Samala [29].

Algorithm 4.5 Area Constraint

```
let $N_r$ be the number of instances of resource type $r$, where $r \in S$
let $k_i$ be the number of device primitives required to implement the resource type $r_i$
for primitive type i
let $A_i$ be the available number of device primitives of type $i$

$\forall i \in \{LUTs, FFs, DSP48s, BRAMs\}$:
    $\forall r \in R$:
        $\sum(k_i N_r) \leq A_i$
```

WSDP, for a resource $r \in R$, is calculated using (4.4),

$$W_r = \sum_i \left(\frac{k_i}{A_i}\right) \ \forall i \in \{LUTs, FFs, DSP48s, BRAMs\}, \tag{4.4}$$

where k_i and A_i are defined in algorithm 4.5. The total area cost of a generated architectures can be calculated using (4.5),

$$W_{total} = \sum_r (N_r W_r) + W_{muxes} + W_{shift\_reg}, \tag{4.5}$$

where W_{muxes} and W_{shift_reg} represents the WSDP of multiplexers and shift registers, respectively.

4.9 Distribution and Search

To determine the optimal schedule for a given data flow graph, we construct the finite domain model for the scheduling problem and employ all the constraints described above to facilitate propagation. As mentioned earlier, although effective, propagation alone is not sufficient to arrive at a solution. To arrive at a solution, all the constraints must be satisfied and therefore effective distribution strategy also plays a significant role in determining the solution to a constraint satisfaction problem.

In the distribution phase, a dynamic variable ordering is used which select an unbound variable with the smallest domain size and tries to find a solution using backtracking search. The constraint solver first propagates all the finite domain constraints described in sec. 4.1 through sec. 4.8 and forms an initial constraint store. Once all the constraints have been propagated and a reduced constraint store has been obtained, the constraint solver starts the distribution phase with the intial constraint store. The constraint solver has three classes of varibles to distribute: resource variables corresponding to each resource type in the resource set, region variables corresponding to each region, and start times variables modeling the scheduling cycle for nodes present in the DFG.

The constraint solver starts the distribution phase by distributing on the resource variables. At this point, distribution algorithm sorts all unbound resource variables by their domain size. The sorting algorithm also removes from consideration all variables which are already bound to fixed values, since distribution on those variables is irrelevant. Once the distributor has filtered all the resource variables, it selects a variable with the smallest domain size. It then applies a mid-point distribution algorithm by constraining the variable to be equal to the value closest to the mid-point value of its current domain. The distributor applies the same algorithm to all the remaining resource variables.

Once the distributor finishes with resource variables, it then proceeds with the distribution on region variables. At this point, the distributor selects a region with the highest priority. To solve a region, distributor forks a subspace for the current region. Within each space, parent space's intial constraint store is replicated and all the constraints are applied, as mentioned in sec. 4.5. Once all the constraints have been re-applied in the subspace, the distributor invokes best-case search using source-to-sink distance as the comparison metric. The sub-space distributor has a different distribution algorithm than parent space's distributor. The sub-space distribution algorithm has two classes of variables: distance variables, modeling the distance between the source and sink node of the current region; and start time variables, modeling the schedule time for each node in the region. In the case of distribution on both distance variables and start time variables, sub-space distributor applies the first-fail heuristic, by selecting the distance variable or start time variable with smallest domain size. It then applies a min-point distribution algorithm by constraining the value of the selected variable to be equal to the minimum value of its current domain. The sub-space distributor toggles between distance and start time variables until a solution is found. Once a solution is found, a comparison function calculates the distance between the source node and sink node of the current region, and asserts that the next solution should be "better" than the current solution, with "better" being defined as smaller distance.

Once the distributor finishes with the distribution on region variables, the sub-space, created in the region distribution mode, is merged with the parent space by copying the values of the start time variables determined in the sub-space to the corrosponding variables in the parent space. Thus, when the distributor is finished with the region distribution, all the nodes which form a region will be bound to fixed values. The distribution algorithm then filters start time variables which are already bound to fixed values. For the distribution on the remaining start time variables, the distributor employs the same distribution algorithm as the region distribution. The parent space distributor toggles between resource variables, region and start time variables and performs a best-case search to find solution with WSDP as the metric of comparison. Once a solution to the space is found, comparison function calculates the WSDP of the current solution and asserts that WSDP of the next solution should be less than WSDP of the current solution. The constraint solver has been configured to terminate the search when the best solution is found or time-out is reached. The time-out for the search is set to 10 minutes.

4.10 Summary

In this chapter, we discussed all the constraints that are used to formulate a scheduling problem. We implemented all these constraints in Mozart Programming System using Oz and a basic constraint solver is developed. We experimentally evaluated the performance of the constraint solver on various multimedia kernels. To improve the performance and the scalibility of the solver, we implemented different versions of the constraint solver. A detailed analysis of the performance and scalability of different versions of the constraint solver is described in the next chapter.

Chapter 5

Experimental Evaluation and Results

This chapter discusses in detail the experimental evaluation of our Oz-based constraint model. We implemented the constraint model in Oz and examined its performance on four DFGs to derive schedules and determine number of functional units. The DFGs used to evaluate the performance of the constraint solver model streaming applications like fast fourier transform (FFT), discrete cosine transform (DCT), inverse discrete cosine transform (IDCT), and euclidean distance calculation (DC). The structures of these kernels are shown in table 5.1. The constraint solver, which is an integral part of CHARGER, does not facilitate the complete architecture generation. The constraint solver only determines the number of functional units required to execute the DFG as well as schedule for the operational nodes targeting these functional units. We evaluated the CP-based scheduler for various levels of replication index and critical path relaxation. As mentioned in sec. 3.3, CHARGER implements a tool flow for automating the generation of hardware architectures based on the results produced by FDS and the Oz-based scheduler.

CHARGER also generates a number of statistical reports for each value of user specified replication index and critical path relaxation, which characterizes the generated architectures. These reports include schedule of the replicated DFG; memory allocation and binding results and a summary report describing the total FPGA resource usage in terms of device primitives (#LUTs, #FFs, #BRAMs, and #DSP48s); an estimate of throughput, WSDP and efficiency which is the ratio of throughput and WSDP, of the architectures generated using both FDS and CP-based scheduler. We compare the efficiency of the architectures generated based on the schedules produced by the CP-based scheduler against the architectures generated based on the schedules produced by FDS scheduler.

As mentioned earlier, scheduling and resource allocation are NP complete problems.

Table 5.1: Basic structures of various data flow graphs.

DFG	#Nodes	Resource Types	#Edges	CP Length	#Nodes on CP	#Regions
DCT	71	FP_MULT FP_ADD FP_SUB	92	91	10	18
IDCT	71	FP_MULT FP_ADD FP_SUB	92	91	10	18
FFT	64	FP_MULT FP_ADD FP_SUB	80	65	8	nil
DC	18	FP_MULT FP_ADD FP_SUB FP_COMPARE	17	52	7	nil

Thus, the constraint solver should be scalable and robust. We implemented multiple versions of the constraint solver to improve the performance and robustness. All versions have a basic constraint model in common, which implements the constraints discussed in the previous chapter. A detailed discussion about these versions is done in the following sections. We compared the constraint solver experimentally with non-pipelined version of FDS algorithm and a pipelined version of FDS described by Paulin and Knight [8]. The pipelined version of FDS assumes that data arrives at a fixed interval, called the Initiation Interval(II), while the non-pipelined version of FDS treats the replicated data flow graph as a single, large DFG and tries to spread the slack evenly among all the iterations.

5.1 Oz Scheduler (Version 1)

This version of the constraint solver implements the basic constraint model using all the constraints discussed in Chapter 4. We evaluated the performance of the constraint solver to determine the schedules for various levels of user-specified replication index and slack. The DFG used to evaluate the performance of the Oz-based scheduler are the DCT and FFT kernels. The main objective is to determine the impact on scalability and robustness of the constraint solver as the size of search space increases. For each value of replication index and critical path relaxation, the constraint solver determines a schedule which has a minimum WSDP. The resulting schedule is then used by CHARGER to generate a hardware architecture, after which CHARGER determines the overall WSDP of the generated architecture.

We compared WSDPs of the architectures generated based on schedules produced using the modified FDS algorithm against the architectures generated based on schedules produced by the Oz-based scheduler. Table 5.2 and table 5.3 show the specifications of the architectures generated based on schedules produced by the Oz-based scheduler and FDS scheduler, respectively.

It can be seen from table 5.2 and table 5.3 that the architectures which are generated

Table 5.2: Specifications of the architectures generated based on Oz Scheduler (version 1) for DCT data flow graph.

LU	CPR	OpLUTs	TotLUTs	TotFFs	DSP48s	WSDP	$\frac{TPUT}{WSDP}$
5	0	18664	23336	19978	80	1.08	0.122
5	25	2484	4596	2308	4	0.0804	1.09
5	50	2100	3764	2052	4	0.0743	1.02
5	75	2100	3764	2052	4	0.0743	1.02
5	100	2100	3796	2052	4	0.0743	1.02
10	5	9304	16120	9443	32	0.477	0.437
10	25	4232	8104	4040	8	0.151	1.16
10	50	4068	7876	4204	8	0.151	1.03
10	75	8112	13488	4713	4	0.144	0.869
10	100	4656	8336	3528	4	0.108	1
15	5	15876	27556	16266	52	0.788	0.397
15	30	8412	15484	6667	12	0.249	1
15	50	7332	15044	5741	8	0.2	1.09
15	75	6788	13380	5676	8	0.19	0.998
15	100	6596	13060	5580	8	0.188	1.01
20	5	18608	31536	19078	64	0.951	0.438
20	25	8784	16560	7759	16	0.303	1.16
20	50	11292	20540	8044	12	0.285	1.02
20	75	11680	21664	7504	8	0.247	1.01
20	100	7844	14788	5870	8	0.199	1.09

using the schedules obtained from the constraint solver, have significantly lower WSDP and higher efficiency as compared to the architectures generated using the schedules obtained from modified FDS algorithm. Since, this version of the Oz-based scheduler models only the basic constraints needed to find a schedule for a DFG, optimal values of WSDPs are obtained. However, scalability is a major issue with this version of the constraint solver. For a given DFG, as the size of the search space increases with increase in replication index and CPR, the constraint solver either fails to find a solution within the specified time limit or it blows up in memory when finding a solution.

As mentioned in sec. 4.9, to find a schedule with minimum WSDP, the distributor invokes best-case search using mid-point distribution with WSDP as the metric of comparison. The comparison function asserts that future solutions must be "better" than the current

Table 5.3: Specifications of the architectures generated based on the modified FDS algorithm for DCT data flow graph.

LU	CPR	OpLUTs	TotLUTs	TotFFs	DSP48s	WSDP	$\frac{TPUT}{WSDP}$
5	0	25312	30688	26112	80	1.15	0.0943
5	25	6260	9684	6025	16	0.255	0.357
5	50	7056	10544	6764	16	0.264	0.296
5	75	8320	11712	8233	24	0.362	0.189
5	100	7088	10576	6762	16	0.264	0.231
10	5	18000	25392	18016	48	0.744	0.28
10	25	8908	14316	7889	16	0.291	0.613
10	50	8048	13584	7181	16	0.283	0.535
10	75	9624	16568	9449	28	0.438	0.297
10	100	7824	12784	7115	16	0.278	0.413
15	5	24096	35488	24136	64	1	0.312
15	30	10112	18400	9194	24	0.405	0.628
15	50	11352	19160	10352	24	0.416	0.539
15	75	9408	17152	8779	24	0.396	0.486
15	100	9320	16232	8498	16	0.305	0.558
20	5	30896	45904	30576	80	1.26	0.33
20	25	11420	21180	9998	24	0.425	0.84
20	50	10560	21344	9358	24	0.422	0.707
20	75	12016	23440	11311	28	0.487	0.527
20	100	9920	19328	9164	24	0.41	0.548

solution, with “better” being defined as lower WSDP. WSDP is chosen over throughput (TPUT) as the metric of comparison because TPUT is calculated as a ratio of replication index and relaxed critical path. So, all the solutions generated by the solver for a particular replication index and CPR will have same TPUT value but will vary in terms of WSDPs. The constraint solver determines a number of solutions before finding the optimal solution. As the size of the search space increases, the number of these intermediate solutions also increases, which results in a large amount of memory consumption and longer search time. To reduce the search time and memory consumption, we have modified the comparison function to assert that WSDP of the next solution should be at least 10% less than the current solution. Thus, constraint solver skips some of the intermediate solutions before finding the final solution. In this case, the final solution obtained may not be optimal because of the fact that there could be a number of solutions having WSDP values less than the current solution, but not 10% better.

However, the modified compare solution significantly reduces the time by the constraint solver. Table 5.4 shows the time taken by the solver using the original and the modified compare functions. To illustrate the impact of the modified compare function on search time, we plot the difference in the search time taken by both the functions for various replication index. This plot is shown in fig. 5.1.

Table 5.5 and table 5.6 show the difference in WSDP and efficiency of the architectures obtained using the original and modified comparison functions for DCT and FFT DFGs. A detailed comparison of the difference between WSDP and efficiency for DCT is shown in fig. 5.2 and fig. 5.3. Similarly, the difference between WSDP and efficiency for FFT is shown in fig. 5.4 and fig. 5.5. From the plots, it can be seen that for higher values of relplication index and CPR, the estimated efficiency and WSDP values of the resulting architectures obtained using the modified compare function by the constraint solver are “better” than the values obtained using the original compare function. Positive values on these plots indicate that WSDP of architectures resulting from the original compare function is better than the modified compare function,while for negative values reverse is true.

Table 5.4: Search times of the solver using the original and the modified compare functions for DCT data flow graph.

LU	CPR	Search Times Using Original Compare Function (sec)	Search Times Using Modified Compare Function (sec)
5	5	90.902	32.617
5	25	33.170	18.854
5	50	56.188	22.449
5	100	50.671	25.350
10	5	601.265	33.882
10	25	141.115	61.411
10	50	600.000	600.004
10	100	240.196	153.705
15	5	600.836	84.139
15	25	632.925	312.559
15	50	600.017	600.020
15	100	600.007	600.006

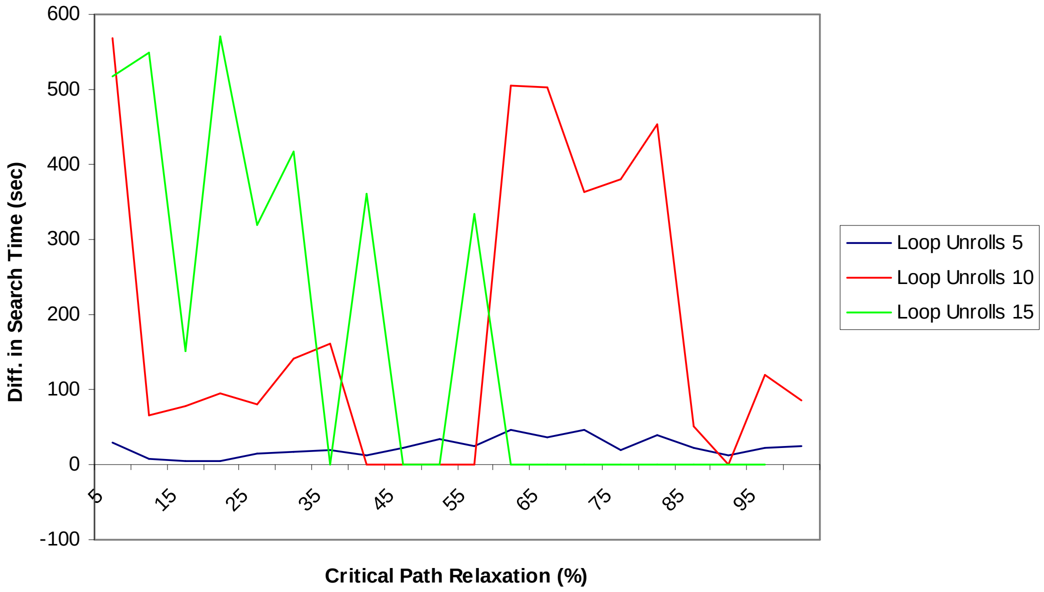


Fig. 5.1: Difference in search times of the solver using the original and the modified compare functions for DCT data flow graph.

Table 5.5: Difference in WSDP and efficiency of the architectures generated based on the constraint solver (version 1) using original and modified compare functions for DCT data flow graph.

LU	CPR	WSDP	$\frac{TPUT}{WSDP}$
5	0	0	0
5	50	0.0068	0.086
5	100	0.0066	0.086
10	5	0.014	0.013
10	50	0.017	0.102
10	100	0.002	0.016
15	5	-0.057	-0.031
15	50	0.017	0.09
15	100	0.023	0.108
20	5	0	0
20	50	0.006	0.02
20	95	-0.031	-0.16

Table 5.6: Difference in WSDP and efficiency of the architectures generated based on the constraint solver (version 1) using original and the modified compare functions for FFT data flow graph.

LU	CPR	WSDP	$\frac{TPUT}{WSDP}$
5	5	0.006	-0.065
5	25	-0.004	-0.026
5	50	0	0
5	75	0.0018	0.03
5	100	0.0052	0.11
10	5	0.069	0.046
10	25	-0.025	-0.18
10	50	0.002	-0.15
10	75	-0.001	-0.03
10	100	-0.004	-0.35
15	5	0.063	0.03
15	25	0	0
15	50	0	0
15	70	0.007	-0.1
15	100	-0.02	-0.15

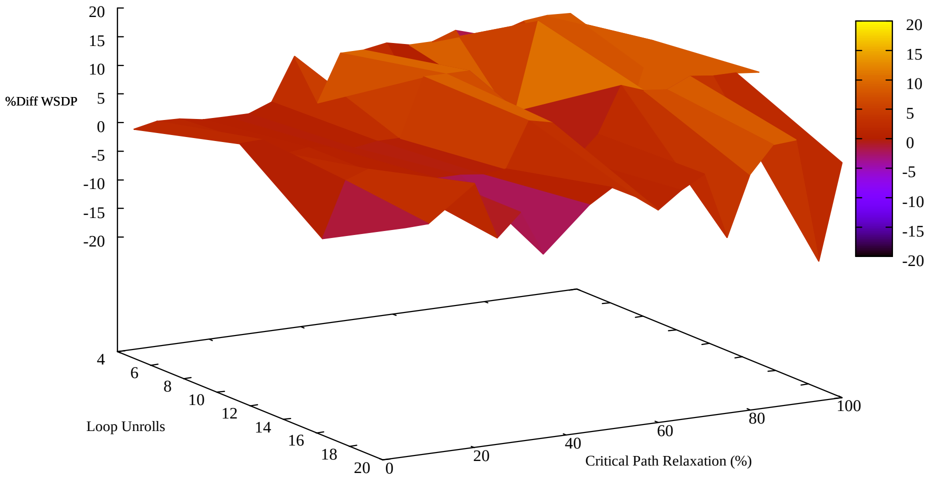


Fig. 5.2: Change in WSDP between the solver (version 1) and the modified FDS algorithm for DCT data flow graph.

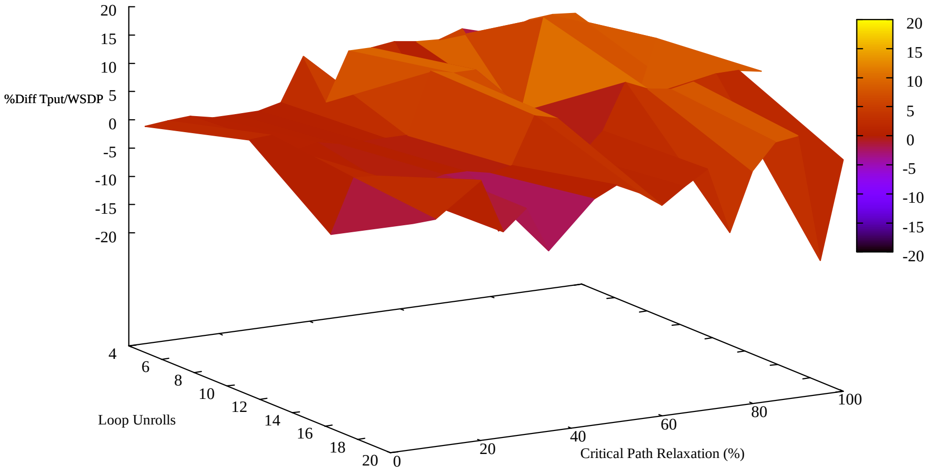


Fig. 5.3: Change in efficiency between the solver (version 1) and the modified FDS algorithm for DCT data flow graph.

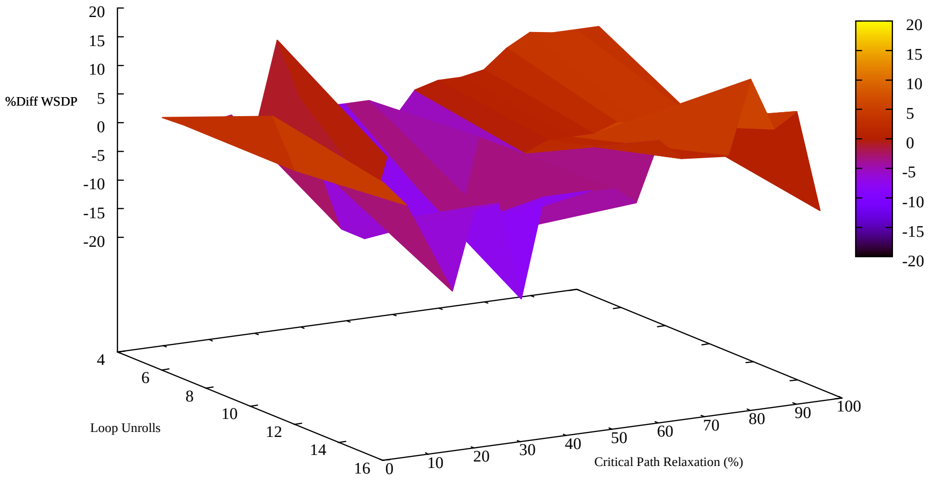


Fig. 5.4: Change in WSDP between the solver (version 1) and the modified FDS algorithm for FFT data flow graph.

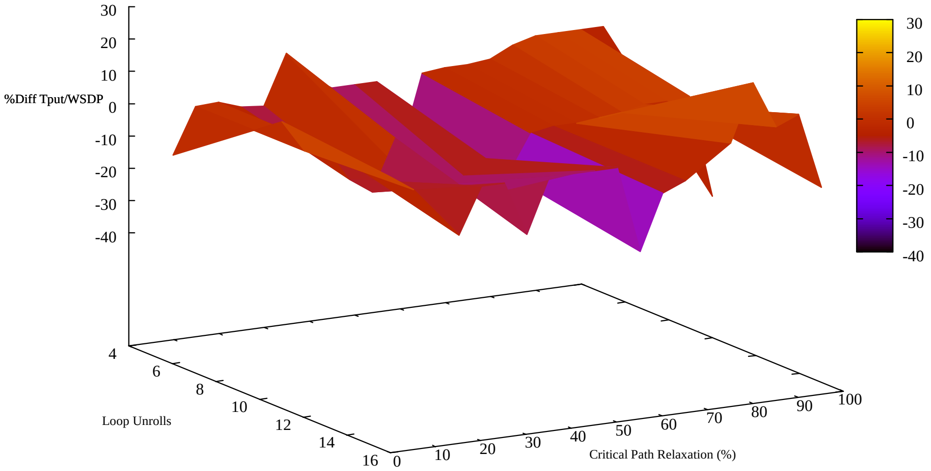


Fig. 5.5: Change in efficiency between the solver (version 1) and the modified FDS algorithm for FFT data flow graph.

For the DCT graph (fig. 5.2 and fig. 5.3), when the original compare function is used, the constraint solver determines better solutions on a regular basis. For some higher values of replication index and CPR, the modified compare function determines better solutions. This is because when the original compare function is used by the constraint solver, the number of intermediate solutions increases with increase in replication index and CPR, and therefore, the solver is not able to arrive at the optimal solution within the specified time limit. But, for the similar situations when the modified compare function is used, the constraint solver arrives at a solution much faster because it skips some of the intermediate solutions. However, under these circumstances, schedule generated by the solver may not be optimal.

For the FFT graph (fig. 5.4 and fig. 5.5), the trend is not consistent; at times the modified compare function out performs the original, while at other times the reverse is true. For the regions where the original compare function out performs, the difference in the WSDP and the efficiency values is small.

We can conclude from the results that although modified compare function does not out perform original compare function in all the situations, it significantly reduces the search time, and therefore can be used in the future versions of the constraint solver.

5.2 Oz Scheduler (Version 2)

The performance of previous solver version, which models only basic constraints, does not scale up for large DFGs. The key to scaling up to large DFGs is a well-defined constraint model and the reduction of the search space in the propagation phase. To improve the scalibility of the constraint solver, in addition to the basic constraints, we included additional constraints which model a heuristic develeoped by Areno [22], called the slack spreading heuristic. Since, CHARGER allows the designer to relax the critical path of the system, slack spreading seeks to divide the relaxed critical path evenly among the replications. To evenly distribute the additional slack, the start times of root nodes and leaf nodes are adjusted as a function of total slack available and node priority. The available slack is calculated using (5.1). Slack spreading heuristic is described in algorithm 5.1. Let $G_R = (N_R, E_R)$ be a data flow graph consisting of R replications of G.

$$slack = \left\lceil cp \times \frac{CPR}{100} \right\rceil \tag{5.1}$$

The heuristic, described in algorithm 5.1, is adapted from Areno et al. [30]. The heuristic is added to the constraint model and the performance of the constraint solver is evaluated for various DFGs. We have used DCT, FFT, and IDCT data flow graphs to illustrate the effectiveness of the Oz-based scheduler in determining the schedules and then generating the hardware architectures based on these schedules. With the addition of slack spreading heuristic, the constraint solver arrives at a solution much faster as compared to the previous solver version. However, the resulting schedule is not an optimal solution.

We compare the search time of the solver in determining schedules for the DCT data flow graph against the search time of the previous solver version. Table 5.7 shows the search times of both versions of the solver. The difference in the search times of the previous version of the constraint solver (version 1) and this version is plotted in fig. 5.6.

From fig. 5.6, it can be seen that addition of the heuristic signficantly reduces the search time of the constraint solver except for one case where version 1 of the constraint solver has lower search time. However, the architecture generated using this solver version is different than version 1 generated architectures. The previous solver version (version 1) generates

Algorithm 5.1 Slack Spreading Heuristic

```
let $Pri : N_R \rightarrow \{0, 2, 3, ..., R-1\}$ returns the priority of a node

$\forall n \in N_R$ :
        **if** $pred(n)$ == NULL
                $st(n) \ \geq \ \left(\frac{Pri(n)}{R}\right) \times slack$

        **else if** $succ(n)$ == NULL
                $st(n) \ \leq \ cp - lat(n) + \left(\frac{Pri(n)}{R}\right) \times slack$
```

optimal schedules and the resulting architectures have lower WSDP as compared to the current version (version 2) based architectures. A detailed comparison of the architectures generated using all the versions of the constraint solver is described in sec. 5.3.3.

In the plots below, we compare the estimated efficiency of the Oz-generated architectures, as compared to those based on the modified FDS algorithm. We have executed each algorithm a number of times, varying the graph replication index and CPR. To compare the performance of the Oz-based constraint solver against the modified FDS algorithm, we plot the difference in the efficiency of the architectures as a percentage. We also plot the difference in the WSDP of the architectures as a percentage. The plots are shown in fig. 5.7 through fig. 5.12. Table 5.8 and table 5.9 show the difference in the WSDP and efficiency of the architectures based on Oz scheduler and FDS algorithm for the various data flow graphs.

For the DCT data flow graph, the results are plotted in fig. 5.7 and fig. 5.8. The plots show the difference in efficiency and WSDP of the Oz-generated architectures and the modified FDS-generated architectures as percentages. From the plots, it can be seen that based on the efficiency and WSDP, the constraint solver consistantly outperforms the modified FDS algorithm. Although the difference in the efficiency of the architectures reduces as we increase replication index and CPR, the constraint solver-generated architectures are still

Table 5.7: Search time taken by version 1 and version 2 of the constraint solver for the DCT data flow graph.

LU	CPR	Search Times of version 1 (sec)	Search Times of version 2(sec)
5	5	32.617	16.314
5	50	22.449	4.860
5	100	25.350	5.283
10	5	33.882	31.488
10	50	600.004	12.947
10	100	153.705	18.392
15	5	84.139	71.150
15	50	600.020	28.826
15	100	600.006	66.053

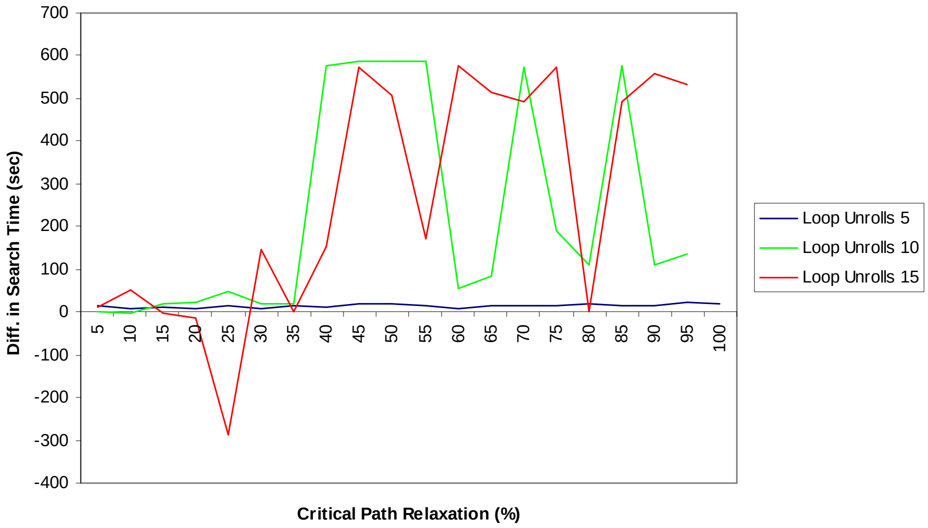


Fig. 5.6: Difference in the search time of version 1 and version 2 of the constraint solver for the DCT data flow graph.

Table 5.8: Difference in WSDP and efficiency of the architectures generated based on the constraint solver (version 2) and the modified FDS algorithm for DCT data flow graph.

LU	CPR	WSDP	$\frac{TPUT}{WSDP}$
5	5	0.252	0.228
5	50	0.036	0.046
5	100	0.032	0.028
10	5	0.255	0.146
10	50	0.034	0.065
10	100	0.025	0.042
15	5	0.268	0.115
15	50	0.038	0.053
15	100	0.051	0.105
20	5	0.306	0.107
20	50	0.124	0.281
20	100	0.139	0.273
30	5	0.58	0.126
30	50	0.061	0.137
30	100	0.112	0.252
40	10	0.18	0.075
40	50	0.044	0.106
40	100	0.159	0.29

more efficient and have lower WSDP as compared to the modified FDS-based architectures.

Table 5.9 shows the summary of the results obtained for IDCT data flow graph. A comparison of the architectures generated for IDCT data flow graphs, using schedules produced by modified FDS algorithm and Oz-based scheduler, is shown in fig. 5.9 and fig. 5.10. Similarly, for FFT data flow graph, the results are plotted in fig. 5.11 and fig. 5.12.

In fig. 5.7 - fig. 5.12, it can be seen that the Oz-based constraint solver out performs the modified FDS algorithm in all the circumstances. The performance of the constraint solver varies for different data flow graphs. For the IDCT data flow graph, it can be seen that efficient architectures are generated using schedules produced by the constraint solver. For smaller values of replication index and CPR, the difference in the efficiency is not large. However, for large levels of graph replication, the constraint solver-generated architectures are on an average more efficient, as compared to the modified FDS-generated architctures.

Table 5.9: Difference in WSDP and efficiency of architectures based on the constraint solver (version 2) and the modified FDS algorithm for IDCT data flow graph.

LU	CPR	WSDP	$\frac{TPUT}{WSDP}$
5	5	0.233	0.218
5	45	0.015	0.092
5	100	0.009	0.033
10	5	0.286	0.103
10	50	0.012	0.032
10	100	0.023	0.057
15	10	0.205	0.172
15	50	0.04	0.12
15	100	0.029	0.075
20	5	0.41	0.085
20	50	0.135	0.341
20	100	0.028	0.074
30	10	0.18	0.081
30	50	0.033	0.04
30	100	0.099	0.246
40	10	0.14	0.13
40	25	0.257	0.24
40	50	0.158	0.246

For the FFT data flow graph, the constraint solver is not able to find significantly improved architectures, as compared to those generated by the modified FDS algorithm.

The scalibility of the constraint solver has improved with the addition of algorithm 5.1 to the constraint model. But, there is still scope for improvement in the scalibility of the constraint solver. We have experimentally found that most of the search time is consumed in the determination of I/O resources required to schedule a given data flow graph. This issue is discussed, in detail, in the following version of the constraint solver.

5.3 Oz Scheduler (Version 3)

This version of the scheduler includes all the constraints mentioned in the basic constraint model which is described in sec. 5.1. It also includes the heuristic described in algorithm 5.1. This version is an extension of the constraint solver discussed in the previous section. The constraint solver described in sec. 5.2 is robust and scales up to large DFGs. But we experimentally evaluated that most of the search time is consumed in the determination of the number of I/O resources required to schedule a given DFG.

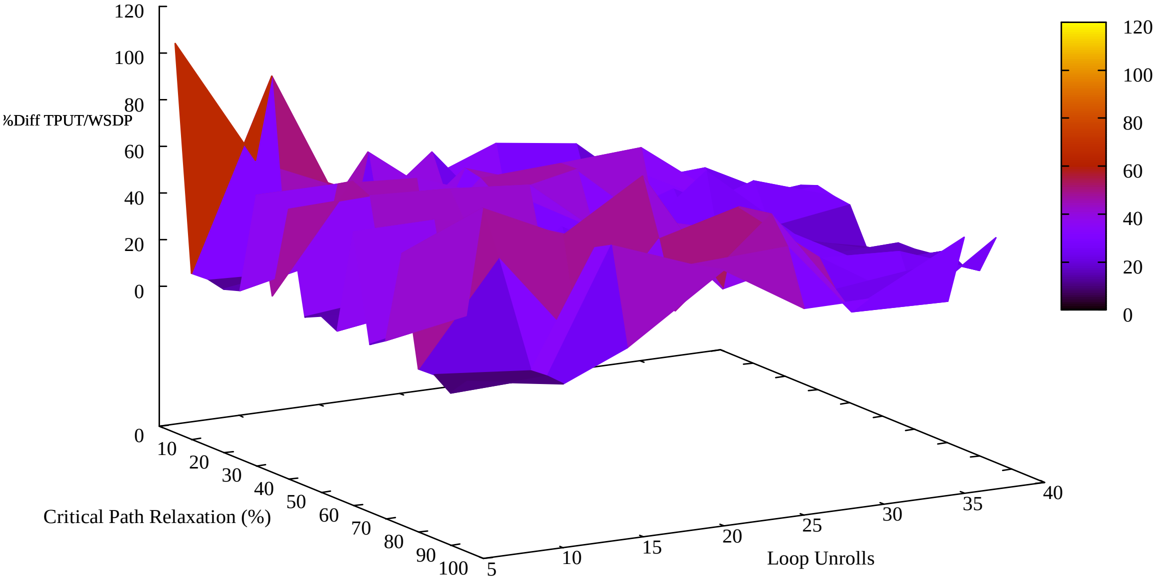


Fig. 5.7: Change in efficiency between the solver (version 2) and the modified FDS algorithm for DCT data flow graph.

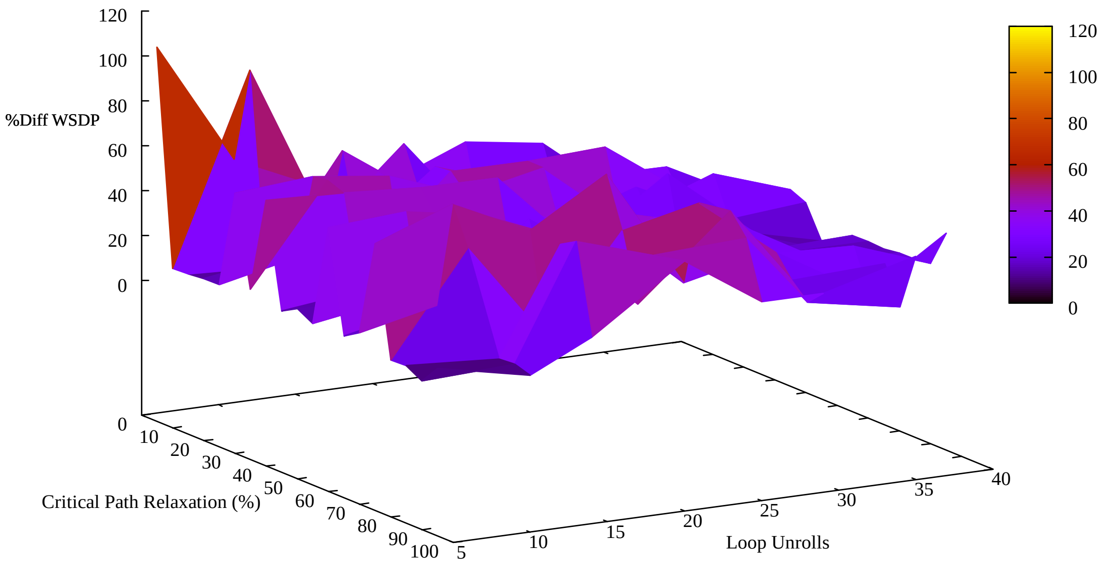


Fig. 5.8: Change in WSDP between the solver (version 2) and the modified FDS algorithm for DCT data flow graph.

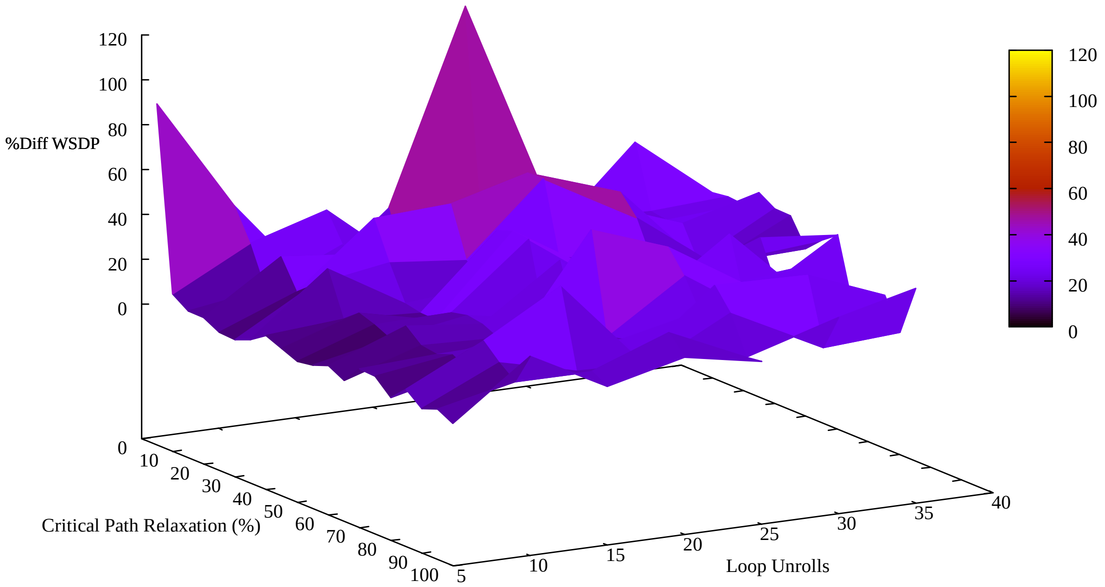


Fig. 5.9: Change in WSDP between the solver (version 2) and the modified FDS algorithm for IDCT data flow graph.

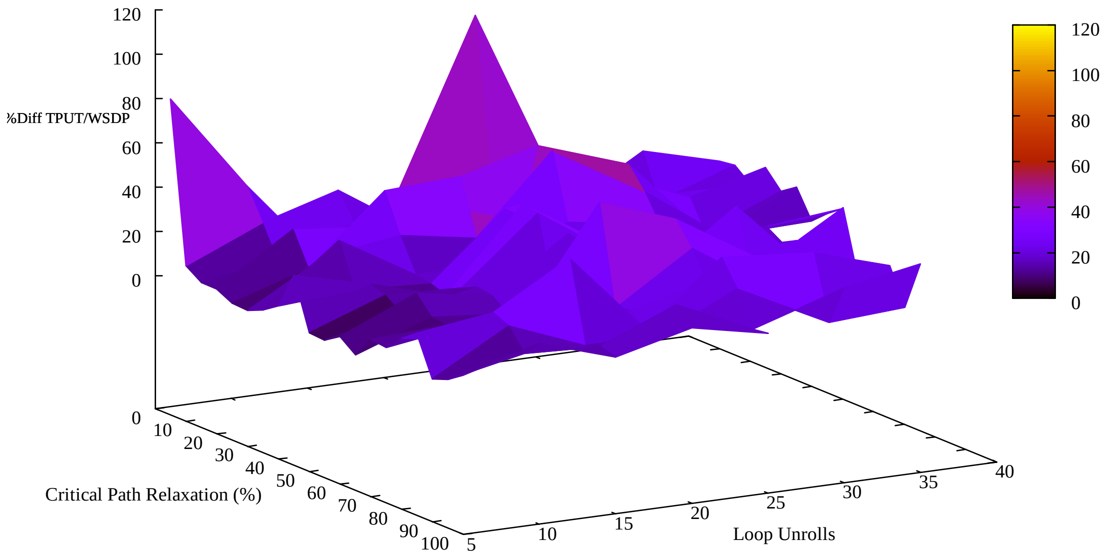


Fig. 5.10: Change in efficiency between the solver (version 2) and the modified FDS algorithm for IDCT data flow graph.

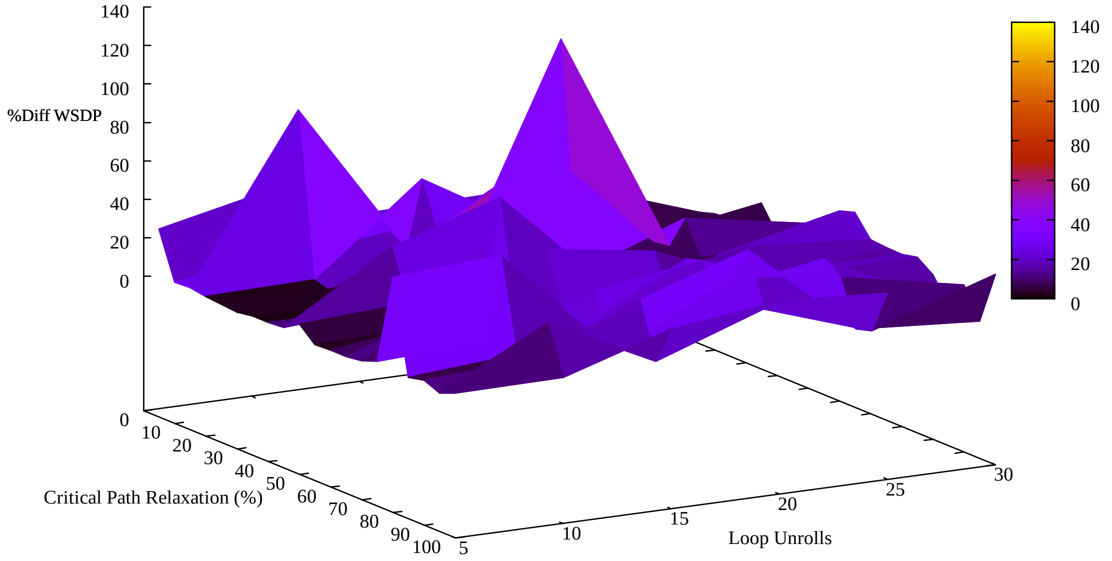


Fig. 5.11: Change in WSDP between the solver (version 2) and the modified FDS algorithm for FFT data flow graph.

5.3.1 Impact of I/O Resources on Search Time: Version 3(a)

As mentioned earlier, determination of I/O resources has a significant impact on the search time. Thus, to illustrate the impact of I/O resources on search time, we bind I/O resources during the propagation phase and thus, eliminate their consideration during search. We fix the number of I/O resources to be equal to the number of I/O nodes present in the replicated graph. This additional constraint significantly reduces the search space, as the finite domain variables associated with I/O resources are already assigned fixed values and the distribution on these variables is not required. A comparison of the search time to determine a final solution with a fixed I/O resource set, as compared to the search time without a fixed I/O resource set is shown in fig. 5.13. We plotted the results for DCT data flow graph with loop unrolls (LU) 5 and 10, varying critical path relaxation. From the plot, it can be seen that with fixed I/O resources the search time is very less as compared to the search time taken without fixed I/O resources.

Fixing the number of I/O resources to be equal to the number of I/O nodes also has a significant impact on the resulting architectures. Table 5.10 shows the difference in the specifications of architectures generated by the constraint solver with a fixed I/O resources

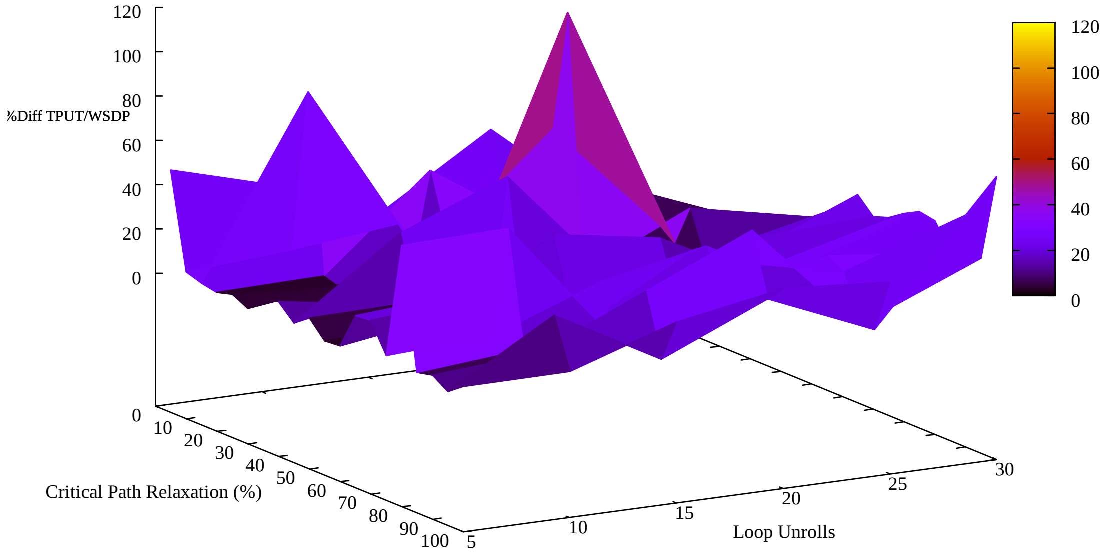


Fig. 5.12: Change in efficiency between the solver (version 2) and the modified FDS algorithm for FFT data flow graph.

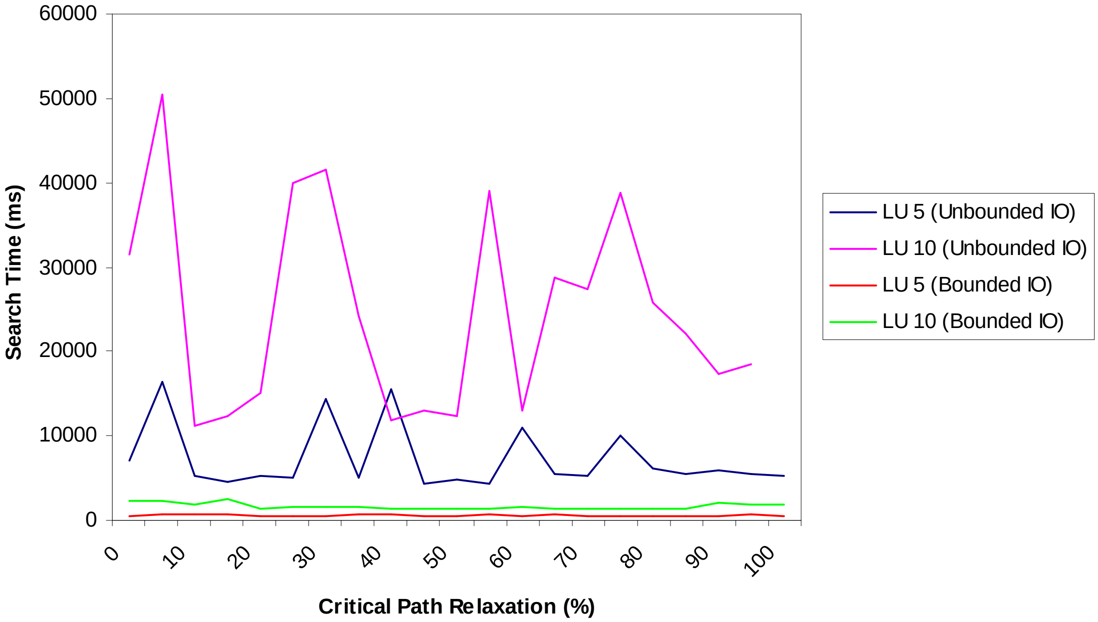


Fig. 5.13: Impact of fixed I/O resources on search time for DCT data flow graph.

and those generated by the solver without a fixed I/O resources (version 1). Positive values in the table indicate that, based on the efficiency and WSDP, architectures generated by the solver without a fixed I/O resources are better as compared to fixed I/O resources constraint solver-based architectures. The architectures generated based on fixed I/O resources are less efficient and have slightly higher WSDP.

5.3.2 Two Phase Search: Version 3(b)

As illustrated in previous section, a fixed I/O resource set helps in reducing the search space by eliminating I/O resources from the search space. But the problem with a fixed I/O resource set is that the number of I/O resources required in the schedule increases propotionaly with an increase in the graph replication level. To overcome this problem, we perform the search in two phases to determine a schedule for a given data flow graph, as described below.

1. In the first phase, we add the constraint which constraints the instances of I/O resources equal to number of I/O nodes which are present the replicated DFG along

Table 5.10: Difference in the specifications of architectures generated by the constraint solver using bounded I/O and unbounded I/O for DCT data flow graph.

LU	CPR	#TotLUTs	WSDP	$\frac{TPUT}{WSDP}$
5	5	4002	0.205	0.205
5	25	230	0.002	0.004
5	50	-154	0	-0.001
5	100	204	0.047	0.043
10	5	2668	0.192	0.12
10	25	952	0.005	0.016
10	50	-712	-0.006	0.024
10	100	108	0.046	0.071
15	5	2034	0.189	0.088
15	50	1776	0.011	0.017
15	100	-828	0.042	0.093
20	5	4720	0.206	0.079
20	50	570	0.086	0.223
20	100	2442	0.057	0.136
25	5	4854	0.21	-0.064
25	50	-1026	-0.044	0.096
25	100	456	0.047	-0.105
30	5	10180	0.43	0.1
30	100	7206	0.141	0.297
35	10	1324	0.059	0.039
35	100	2004	0.068	0.201

with the rest of the constraints described in sec. 5.2, and and use the constraint solver to determine the number of functional units required to schedule the graph.

2. In the second phase, using the set of functional units obtained in the first phase, we repeat the constraint satisfaction problem, fixing the resource counts but allowing the I/O resources to be unbound.

We experimentally evaluated the performance of the constraint solver, using the constraint model described above, against the modified FDS scheduler. We have used the FFT, DCT, and IDCT data flow graphs to compare the architectures generated based on schedules produced by the constraint solver, as compared to those based on modified FDS scheduler. To compare the performance of the Oz-based constraint solver, we use efficiency and WSDP of the architectures as the metric of comparison.

The two phase distribution also affects the number of I/O resources required in the resulting schedule. Table 5.11 shows the summary of the number of I/O resources required for various levels of replication index and critical path. A comparison of the number of input resources required in the schedule, generated by this version and version 2 of the constraint solver for DCT data flow graph is shown in fig. 5.14. Similar comparison for the number of output port required is shown in fig. 5.15.

We plot the difference in the efficiency and WSDP as a percentage of the architectures generated using the constraint solver and those generated using the FDS scheduler. These plots can be seen in fig. 5.16 - fig. 5.21. A summary of the differences in WSDP and efficiency between constraint solver-based architectures and the modified FDS-based architectures is shown in table 5.12 - table 5.14.

From the plots, it can be seen that the constraint solver consistently finds more efficient architectures, as compared to the modified FDS scheduler. The degree of efficiency of the constraint solver varies with different data flow graph, which mainly depends on the structure of the data flow graph. Figure 5.16 and fig. 5.17 show increase in the efficiency and difference in WSDP of the constraint solver-based architectures for DCT data flow graph. From these plots, it is be seen that this version of the solver generated architectures which have better efficiency as compared to the modified FDS algorithm. With the increase in replication index and critical path relaxation, the improvement in the efficiency is not significantly high. But, still the efficiency of the constraint solver-based architecture is better than those based on modified FDS scheduler.

The results for IDCT data flow graph are shown in fig. 5.18 and fig. 5.19. The plots clearly show that the constraint solver out performs the modified FDS scheduler. For certaint levels of replication index and CPR, the difference in the efficiency is very high while for others it is low; therefore no particular trend is seen in the plots beyond the conclusion that the constraint solver generates superior architectures for IDCT DFG .

Table 5.11: Comparison of #Input and #Output required in the schedules generated by vesion 2 and version 3(b) for DCT data flow graph.

LU	CPR	Bounded IO (Version 3(b))		UnBounded IO (Version 2)	
		#Input	#Output	#Input	#Output
5	5	8	2	8	1
5	25	8	1	8	2
5	50	8	1	8	3
5	75	8	1	8	1
5	100	8	1	8	3
10	5	16	2	20	5
10	25	8	2	8	4
10	50	8	1	8	3
10	75	8	1	8	4
10	100	8	1	8	2
15	5	24	4	28	6
15	25	8	2	8	4
15	50	8	2	8	3
15	75	8	2	8	2
15	100	8	1	8	1
20	5	40	5	36	6
20	25	8	4	8	7
20	50	8	3	8	6
20	75	8	2	8	4
20	100	8	3	8	4
25	5	46	6	42	12
25	25	10	4	12	7
25	50	8	3	8	5
25	75	8	3	8	6
25	100	8	2	8	5
30	5	48	6	56	15
30	25	16	5	13	8
30	50	8	4	8	5
30	75	8	3	8	7
30	100	8	3	8	7

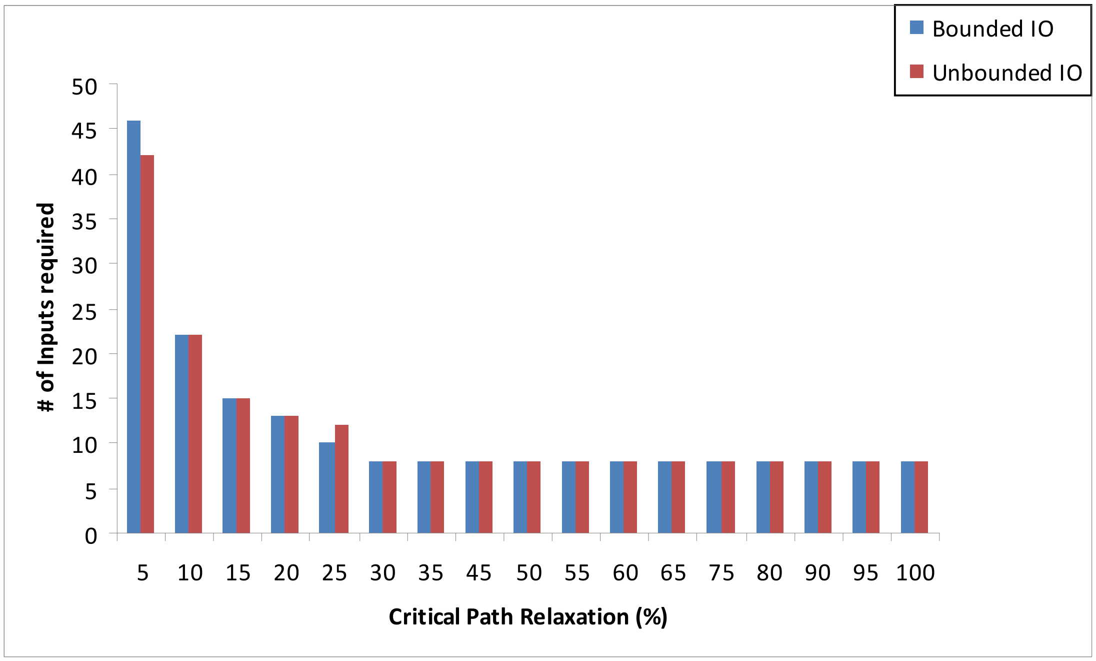


Fig. 5.14: #Input required for 25 replications of the DCT data flow graph with varying CPR.

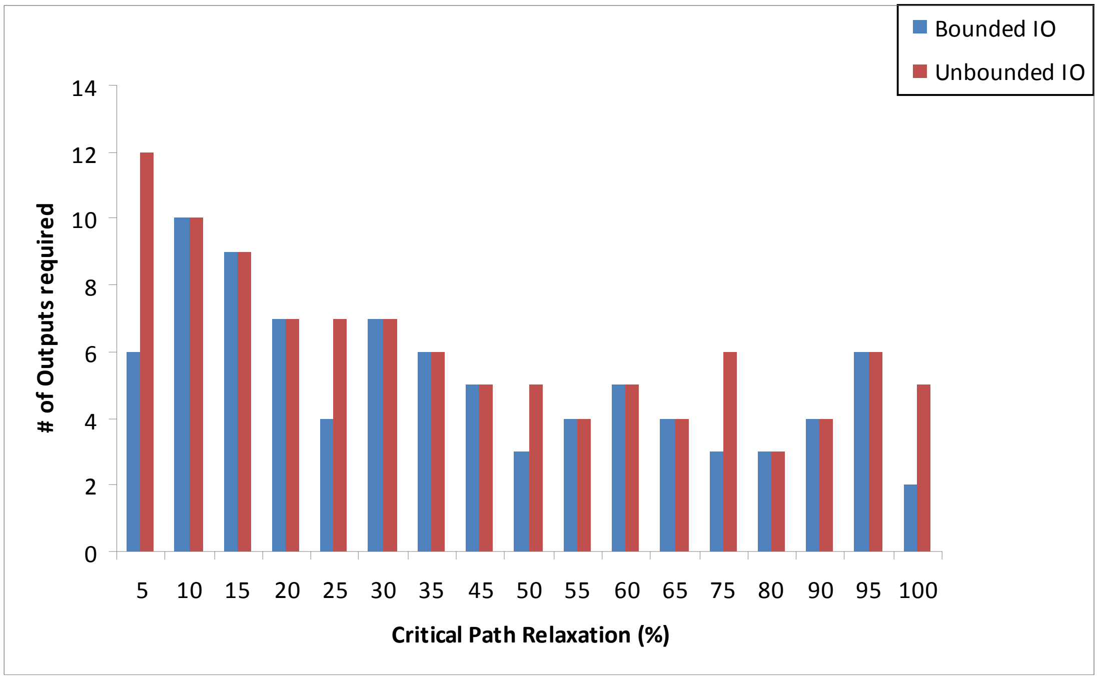


Fig. 5.15: #Output required for 25 replications of the DCT data flow graph with varying CPR.

Table 5.12: Difference in specifications of architectures based on the constraint solver (version 3(b)) and the modified FDS algorithm for DCT data flow graph.

LU	CPR	TotLUTs	WSDP	$\frac{TPUT}{WSDP}$
5	5	7556	105.042	105.164
10	10	8648	99.213	95.190
15	75	8436	58.400	58.230
20	75	9308	72.695	72.486
25	25	7128	51.762	51.980
30	55	3248	50.407	47.668
35	100	6860	58.678	57.037
40	5	41986	99.213	87.273
45	30	8720	52.738	49.792
50	55	8636	45.283	45.608
55	25	7156	41.604	41.190
55	100	7636	16.759	16.474

Table 5.13: Difference in specifications of architectures based on the constraint solver (version 3(b)) and the modified FDS algorithm for IDCT data flow graph.

LU	CPR	TotLUTs	WSDP	$\frac{TPUT}{WSDP}$
5	5	1832	0.109	0.067
10	5	3052	0.255	0.086
15	5	4420	0.261	0.19
15	85	6156	0.133	0.205
20	70	9096	0.197	0.376
25	30	10148	0.201	0.454
30	30	9884	0.243	0.432
35	45	10936	0.254	0.451
40	25	11552	0.395	0.45
40	70	13580	0.299	0.437
45	30	13112	0.367	0.507
50	40	15268	0.313	0.414
50	100	7068	0.043	0.09

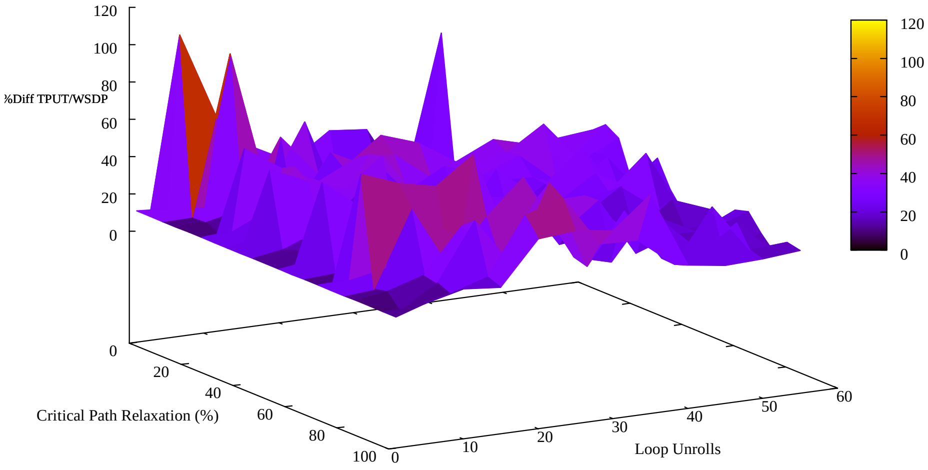


Fig. 5.16: Change in efficiency between the solver (version 3(b)) and the modified FDS algorithm for DCT data flow graph.

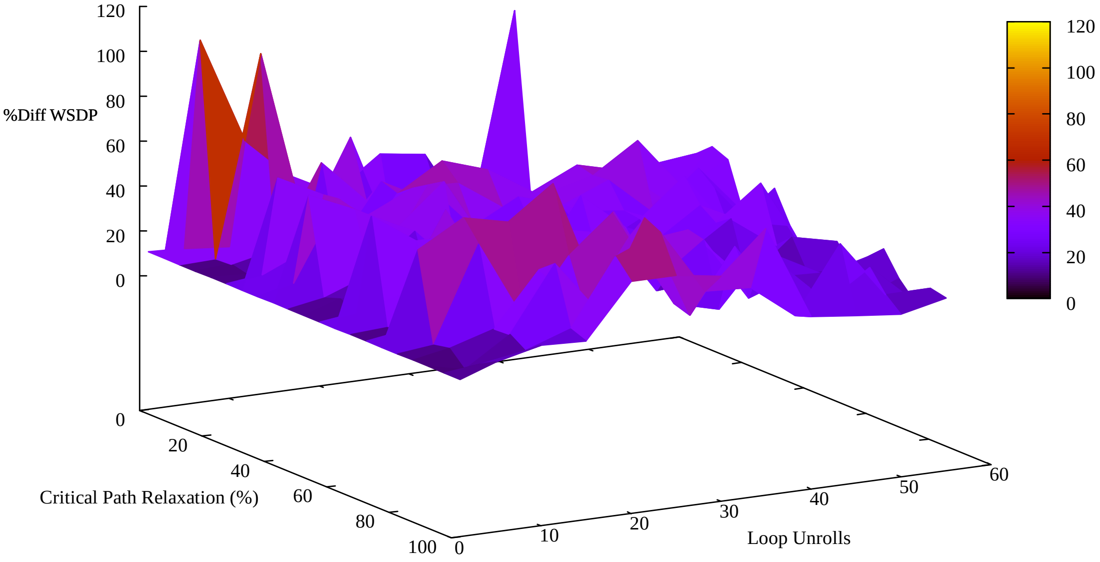


Fig. 5.17: Change in WSDP between the solver (version 3(b)) and the modified FDS algorithm for DCT data flow graph.

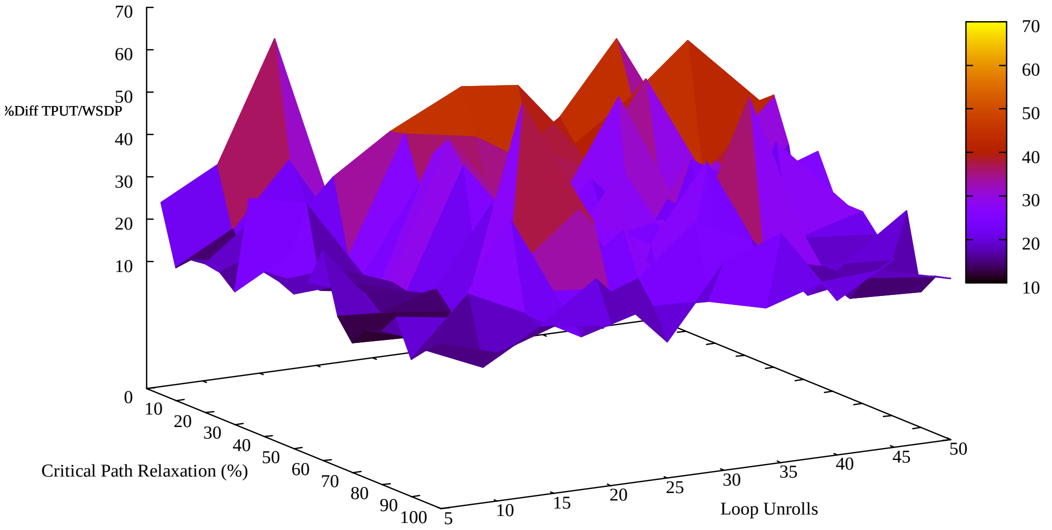


Fig. 5.18: Change in efficiency between the solver (version 3(b)) and the modified FDS algorithm for IDCT data flow graph.

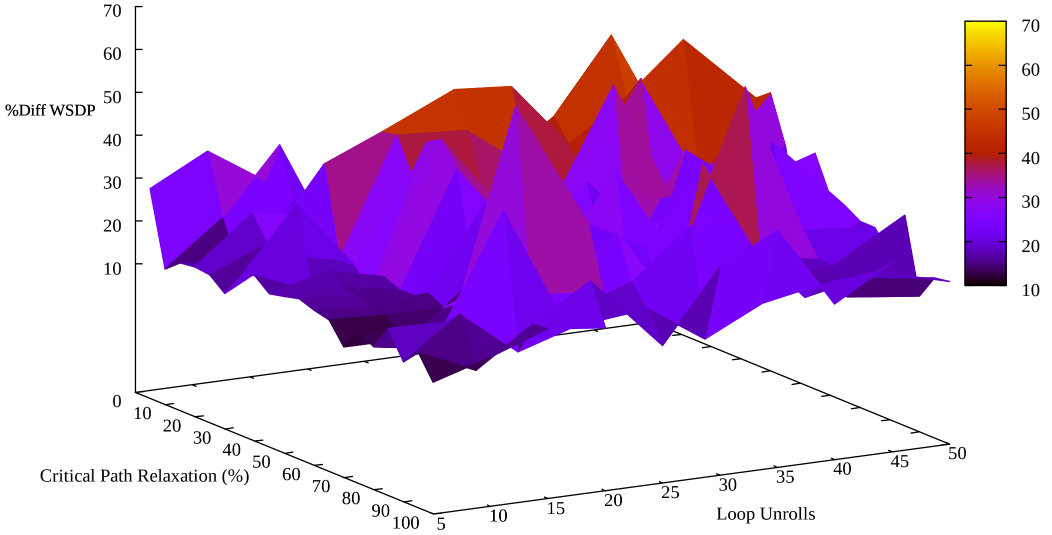


Fig. 5.19: Change in WSDP between the solver (version 3(b)) and the modified FDS algorithm for IDCT data flow graph.

Table 5.14 shows the difference in the #LUTs, WSDP, and efficiency of the architectures generated based on the constraint solver and those based on the modified FDS algorithm, for different levels of graph replication and CPR for the FFT kernel. These results are plotted in fig. 5.20 and fig. 5.21. From the plots, it is clear that the efficiency of the architectures based on the constraint solver is better than the efficiency of the architectures based on the modified FDS algorithm. It can also be seen that, with increase in the graph replication index and CPR, the benefits of the constraint solver becomes less pronounced. These regions correspond to a plateau where, after a certain level of replication index and CPR, no significant improvement in the efficiency and WSDP could be achieved.

We conclude from all these results that the constraint solver-based architectures are better, when compared to the modified FDS-based architectures. It can also be seen that, for all levels of graph replication index and critical path relaxation, the constraint solver outperforms the modified FDS scheduler.

Table 5.14: Difference in specifications of architectures based on the constraint solver (version 3(b)) and the modified FDS algorithm for FFT data flow graph.

LU	CPR	TotLUTs	WSDP	$\frac{TPUT}{WSDP}$
1	5	4112	0.1705	0.615
1	25	4112	0.171	0.623
1	100	4112	0.171	0.623
5	5	1472	0.105	0.528
5	35	4944	0.1758	1.091
5	80	8304	0.294	0.888
10	5	4512	0.216	0.538
10	30	6760	0.149	0.836
10	65	10320	0.307	0.962
15	60	9168	0.298	0.75
15	100	8720	0.2	0.586
20	5	4096	0.2	0.498
20	100	4688	0.171	0.527
30	35	9824	0.248	0.6
40	5	832	0.18	0.436
40	45	11168	0.258	0.44
50	5	608	0.18	0.438
50	35	14144	0.368	0.56
60	10	9472	0.04	0.205
60	95	6800	0.085	0.16
65	35	16184	0.244	0.35
65	40	10584	0.209	0.27

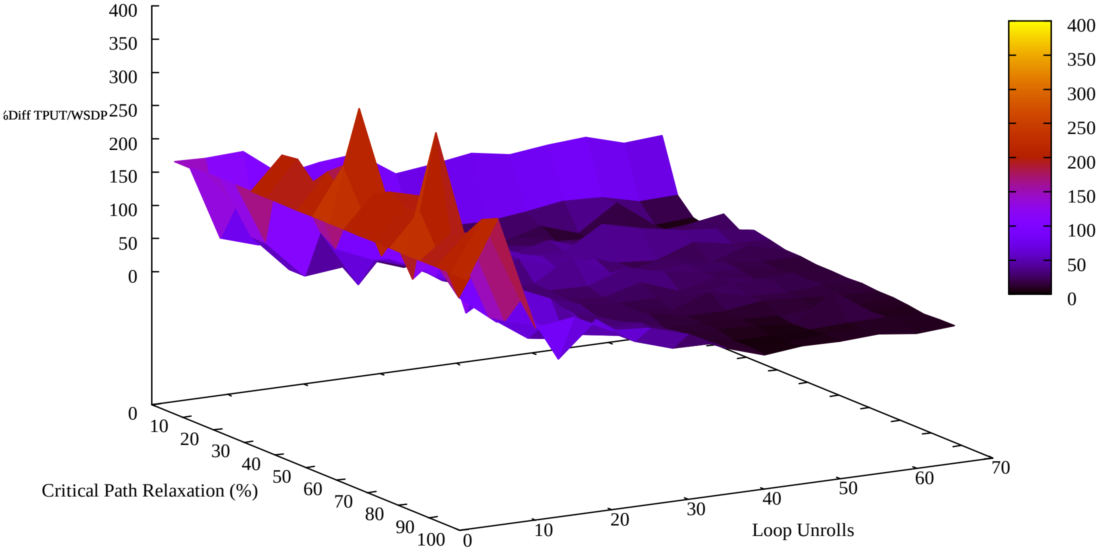


Fig. 5.20: Change in efficiency between the solver (version 3(b)) and the modified FDS algorithm for FFT data flow graph.

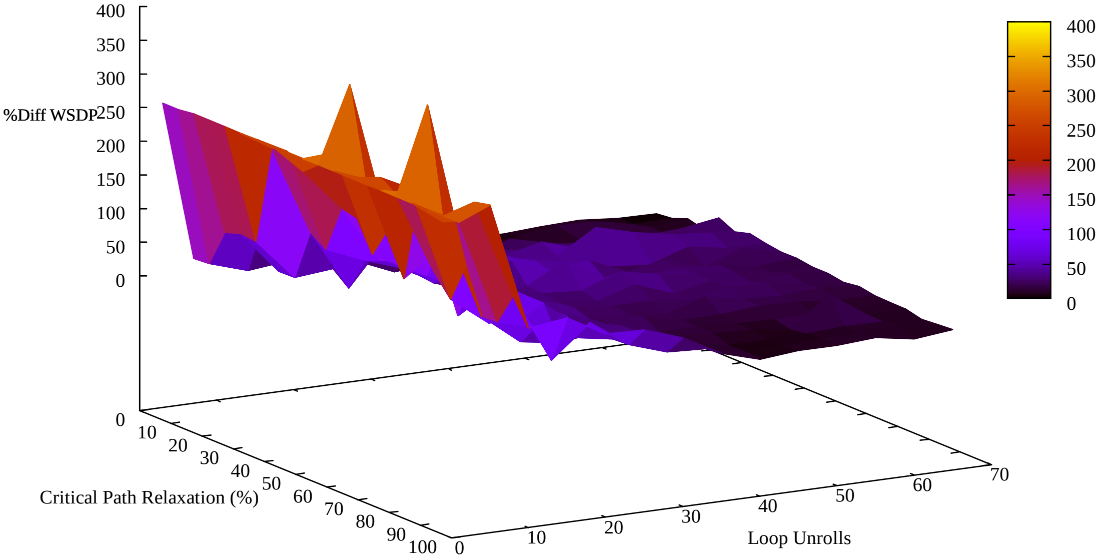


Fig. 5.21: Change in WSDP between the solver (version 3(b)) and the modified FDS algorithm for FFT data flow graph.

5.3.3 Summary

Before we proceed to next section where we compare the performance of the constraint solver against the pipelined version of FDS, we briefly summarize the various versions discussed so far in this chapter. In the previous sections, we compared each version of the constraint solver individually against the modified FDS algorithm. This section compares the performance between all versions of the constraint solver and with the modified FDS algorithm. We compare the performance of various versions of the constraint solver based on efficiency and WSDP of the resulting architectures. Table 5.15 shows the specifications of architectures generated based on the different versions of the constraint solver and specifications of architectures generated based on the modified FDS algorithm.

From table 5.15, it can be seen that version 1, which includes only basic constraints, generates the most efficient architectures having significantly low WSDP. However for large levels of replication index and critical path relaxation, version 1 of the constraint solver fails to find an optimal schedule within the specified time limit. Under these circumstances, version 2 and version 3(b) generate better architectures within the specified time limit. But it does not mean that version 2 and version 3(b) of the solver are better, as compared to version 1 of the solver. If there is no upper limit on the search time, version 1 of the solver will always find better solution for a given data flow graph.

5.4 Oz Scheduler (Version 4)

In this version of the Oz-based constraint solver, we compare its performance against the pipelined variant of the FDS scheduling algorithm proposed by Paulin and Knight [8]. The pipelined version of the FDS algorithm is limited in its consideration of initiation interval. The algorithm only considers integer initiation intervals, with the smallest initiation interval (II) that can be considered is $II = 1$. To generate efficient architectures, CHARGER executes in one of two modes, depending on the amount of CPR specified by user. If the provided slack satisfies the condition shown in (5.2), then the pipelined variant of the FDS algorithm, as shown in algorithm 5.2, is used. If not, then non-pipelined variant of the FDS algorithm is used, which treats the DFG, replicated R times, as one single large DFG and distributes the available slack across all the iterations. At this point in this chapter, all the previous versions of the constraint solver has been compared against the non-pipelined variant of the FDS algorithm.

Table 5.15: Specifications of architectures generated based on all the versions of the solver and the modified FDS algorithm for DCT data flow graph.

LU	CPR	WSDP				$\frac{TPUT}{WSDP}$			
		Ver 1	Ver 2	Ver 3(b)	Mod FDS	Ver 1	Ver 2	Ver 3	Mod FDS
5	5	0.236	0.238	0.239	0.488	0.441	0.435	0.437	0.213
5	10	0.134	0.235	0.236	0.261	0.733	0.426	0.424	0.383
5	20	0.129	0.237	0.237	0.259	0.733	0.391	0.39	0.364
5	30	0.0783	0.231	0.228	0.36	1.06	0.379	0.385	0.243
5	40	0.0784	0.232	0.235	0.366	1.01	0.353	0.349	0.228
5	50	0.0743	0.228	0.229	0.264	1.02	0.342	0.341	0.296
5	60	0.0745	0.241	0.229	0.361	1.02	0.305	0.321	0.204
5	70	0.0743	0.232	0.234	0.264	1.02	0.303	0.297	0.268
5	80	0.0743	0.241	0.237	0.265	1.02	0.277	0.281	0.252
5	100	0.0745	0.238	0.228	0.264	1.02	0.254	0.267	0.231
10	5	0.472	0.477	0.489	0.744	0.442	0.426	0.437	0.28
10	10	0.257	0.26	0.254	0.506	0.771	0.755	0.764	0.395
10	20	0.199	0.245	0.241	0.287	0.933	0.743	0.754	0.646
10	40	0.152	0.244	0.261	0.383	1.03	0.662	0.662	0.422
10	60	0.2	0.237	0.263	0.379	0.686	0.594	0.535	0.437
10	80	0.14	0.246	0.248	0.382	0.873	0.514	0.511	0.336
10	100	0.108	0.26	0.231	0.278	1	0.443	0.497	0.413
15	5	0.718	0.732	0.744	0.788	0.435	0.427	0.442	0.28
15	10	0.394	0.444	0.453	0.506	0.746	0.676	0.662	0.395
15	20	0.266	0.263	0.276	0.287	1.02	1.04	0.987	0.646
15	40	0.2	0.274	0.259	0.382	1.17	0.898	0.948	0.436
15	60	0.211	0.275	0.268	0.283	0.972	0.78	0.799	0.512
15	80	0.19	0.255	0.25	0.438	0.998	0.753	0.769	0.297
15	100	0.188	0.254	0.255	0.278	1.01	0.663	0.661	0.413
20	5	0.951	0.954	0.978	1.26	0.438	0.437	0.426	0.33
20	10	0.517	0.517	0.504	0.788	0.759	0.758	0.779	0.508
20	20	0.369	0.364	0.378	0.538	0.985	0.999	0.963	0.676
20	45	0.291	0.284	0.278	0.325	1.04	1.1	1.11	0.948
20	60	0.287	0.284	0.31	0.418	0.955	1.01	0.897	0.664
20	80	0.267	0.277	0.288	0.413	0.914	0.926	0.858	0.605
20	100	0.197	0.271	0.286	0.41	1.1	0.821	0.777	0.548

$$slack \geq II \times R \tag{5.2}$$

Let CPR represents the percentage of the critical path, added to the end-to-end latency of the system to provide scheduling slack. In the pipelined variant of the FDS algorithm, we calculate CPR in terms of critical path (*cp*), initiation interval and the number of graph replications, as shown in (5.3). Given the critical path of the system and the value of CPR calculated from (5.3), we calculate the available slack in the critical path using (5.1). These equations have been adapted from Areno et al. [30].

$$CPR = \left\lceil 100 \times \frac{II \bullet R}{cp} \right\rceil \tag{5.3}$$

As mentioned earlier, pipelining is an efficient way to allow reuse of the functional units and increasing the efficiency of the generated architctures. But this efficiency comes at the expense of some extra clock cycles. In the pipelined variant of the FDS algorithm, data is assumed to arrive at a fixed interval corrosponding to the initiation interval of the pipeplined data path. In other words, if a given DFG is replicated R times, the data inputs for each replication r_k, $k \in \{1, 2, ..., R\}$ is assumed to arrive at cycle $(k-1) * II$.

To compare the performance of the constraint solver, we added a constraint, described

Algorithm 5.2 Adjusting the Start Times of Root Nodes and Leaf Nodes

```
let $root : N \rightarrow P(N_R)$ be the set of all root nodes
let $leaf : N \rightarrow P(N_R)$ be the set of all leaf nodes
let $Iter : N \rightarrow \mathbb{Z}^+$ be the iteration value of a node
let $let\ cp \in \mathbb{Z}^+$ be the critical path length of the system
let $let\ R \in \mathbb{Z}^+$ be the amount of graph replication

$\forall r \in N_R$ :
        $st(r) = II \times Iter \qquad where,\ r \in root(n)$

$\forall l \in N_R$ :
        $st(l) = cp + (\frac{Iter}{R} \times II) - 1 \qquad where,\ l \in leaf(n)$
```

in algorithm 5.2, to the constraint model. This additional constraint is applied on all the root nodes and leaf nodes in the new replicated graph $G_R = (N_R, E_R)$, whose excution windows are adjusted to produce the allocation and schedule. The algorithm described in algorithm 5.2 is different from slack spreading heuristics in that it is calculates the available slack based on the initiation interval rather than CPR. Also the algorithm described in algorithm 5.2 provides an interface for the input and output nodes.

We examined DCT, FFT, and IDCT data flow graphs and plotted the % difference in the WSDP and efficiency of the architectures, varying levels of graph replication and initiation intervals. These plots are shown in fig. 5.22 - fig. 5.27. Table 5.16 - table 5.18 show the difference in #LUTs, WSDP, and the efficiency of the architectures generated based on the constraint solver and those based on pipeline FDS algorithm for different data flow graphs.

For DCT data flow graph (fig. 5.22 and fig. 5.23), it can be seen that the constraint solver is able to determine architectures with high throughput and low WSDP, therefore these architectures are more efficient as compared to the architectures generated based on pipelined FDS algorithm. It can also be noted that with increase in graph replication level and initiation interval, the difference in the efficiency and WSDP increases significantly.

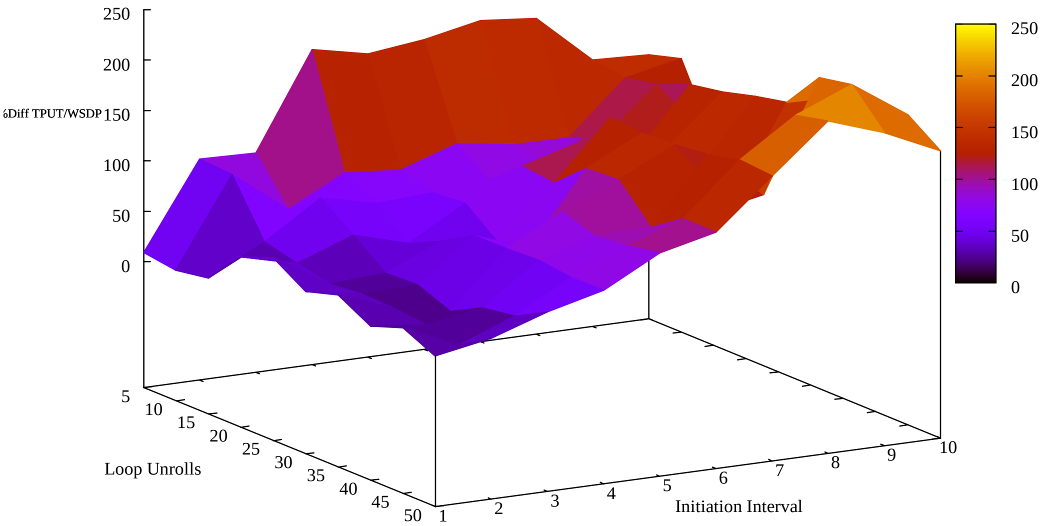


Fig. 5.22: Change in efficiency between the solver (version 4) and the pipelined FDS algorithm for DCT data flow graph.

Table 5.16: Difference in specifications of architectures based on the constraint solver (version 4) and the pipelined FDS algorithm for DCT data flow graph.

LU	II	TotLUTs	WSDP	$\frac{TPUT}{WSDP}$
5	1	2142	0.022	0.034
5	7	6458	0.1886	0.529
10	2	7392	0.143	0.551
10	9	4530	0.165	0.512
15	3	7050	0.143	0.481
20	1	6678	0.134	0.362
20	9	16362	0.207	0.668
30	6	13628	0.224	0.644
30	9	19252	0.227	0.779
40	5	17782	0.205	0.545
40	9	21872	0.284	0.906
50	1	14704	0.131	0.26
50	8	20984	0.315	0.887
50	10	20148	0.231	0.72

For the IDCT data flow graph (fig. 5.24 and fig. 5.25) and FFT data flow graph (fig. 5.26 and fig. 5.27), it can be seen from the plots that the constraint solver-based architectures compare well against the architectures based on the pipelined variant of the FDS algorithm. The architectures, generated based on the schedules produce by the constraint solver, are not only more efficient but also require less area on an FPGA. The constraint solver out performs the pipelined FDS algorithm for each level of replication index and critical path relaxation. In case of IDCT, the difference in the efficiency of the architectures is very high in some regions and less for others. In case of FFT, we observed a gradual increase in the difference in the efficieny with an increase in graph replication index and initiation interval.

We conclude from these results that the constraint solver outperforms the pipelined FDS algorithm, for all levels of loop unrolls and critical path relaxation which indicates the robustness and scalability of the constraint solver. For all the data flow graphs that we examined, the constraint solver generates architectures with high throughput and minimum area consumption.

Table 5.17: Difference in specifications of architectures based on the constraint solver (version 4) and the pipelined FDS algorithm for IDCT data flow graph.

LU	II	TotLUTs	WSDP	$\frac{TPUT}{WSDP}$
1	1	2152	0.025	0.218
1	10	1200	0.0549	0.259
5	2	4104	0.037	0.177
5	9	5168	0.1776	0.575
10	1	4448	0.027	0.086
10	9	7716	0.101	0.346
15	5	5178	0.084	0.369
15	10	11424	0.168	0.528
20	2	6150	0.081	0.258
20	10	8180	0.151	0.376
25	1	9098	0.058	0.21
25	10	8266	0.149	0.379
30	1	11714	0.161	0.47
30	9	14624	0.187	0.676
35	1	12038	0.118	0.32
35	10	9844	0.163	0.405

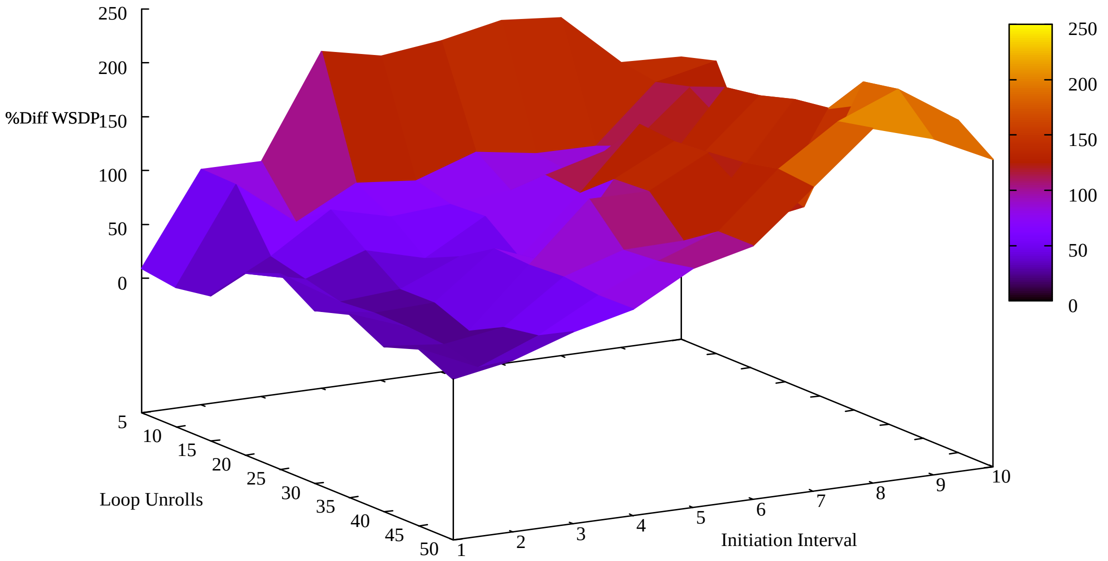


Fig. 5.23: Change in WSDP between the solver (version 4) and the pipelined FDS algorithm for DCT data flow graph.

Table 5.18: Difference in specifications of architectures based on the constraint solver (version 4) and the pipelined FDS algorithm for FFT data flow graph.

LU	II	TotLUTs	WSDP	$\frac{TPUT}{WSDP}$
1	1	2304	0.111	0.764
1	10	4240	0.1719	0.737
5	1	3072	0.022	0.789
5	10	6416	0.1863	0.668
10	1	5696	0.039	0.409
10	10	9856	0.158	0.45
20	1	11184	0.119	0.67
20	10	12028	0.214	0.637
30	1	8368	0.097	0.3
30	10	13308	0.222	0.709
40	1	16764	0.146	0.38
40	10	14204	0.227	0.744
50	1	7996	0.085	0.17
50	10	14652	0.229	0.774
60	1	10048	0.057	0.12
60	10	15356	0.233	0.79

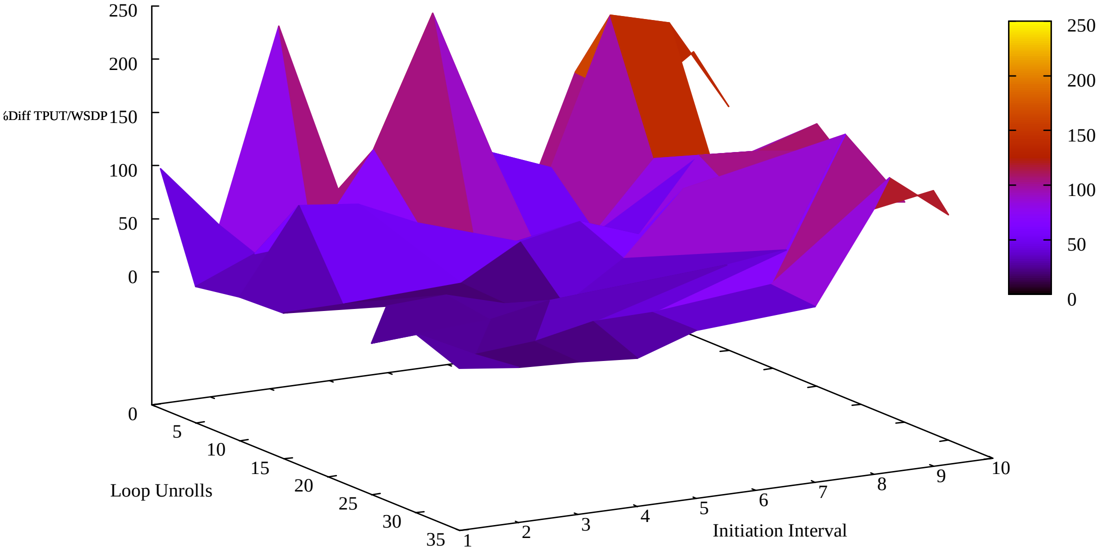


Fig. 5.24: Change in efficiency between the solver (version 4) and the pipelined FDS algorithm for IDCT data flow graph.

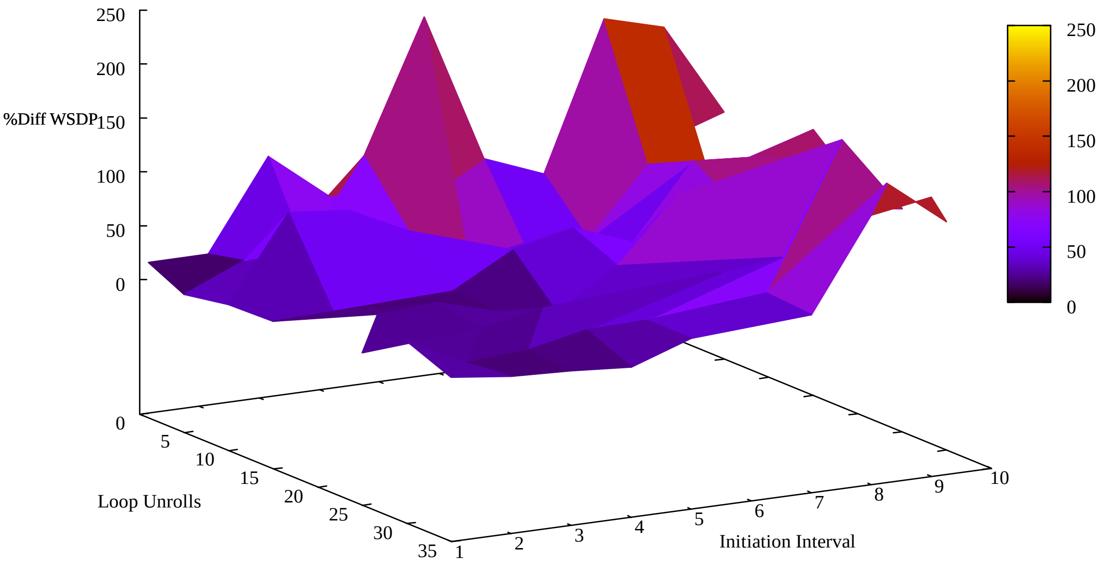


Fig. 5.25: Change in WSDP between the solver (version 4) and the pipelined FDS algorithm for IDCT data flow graph.

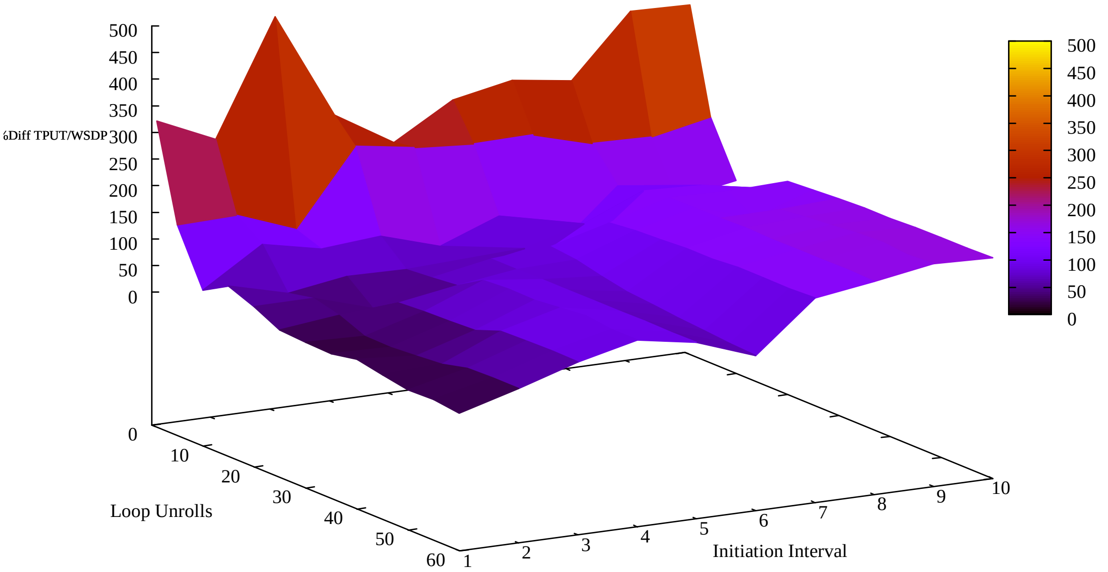


Fig. 5.26: Change in efficiency between the solver (version 4) and the pipelined FDS algorithm FFT data flow graph.

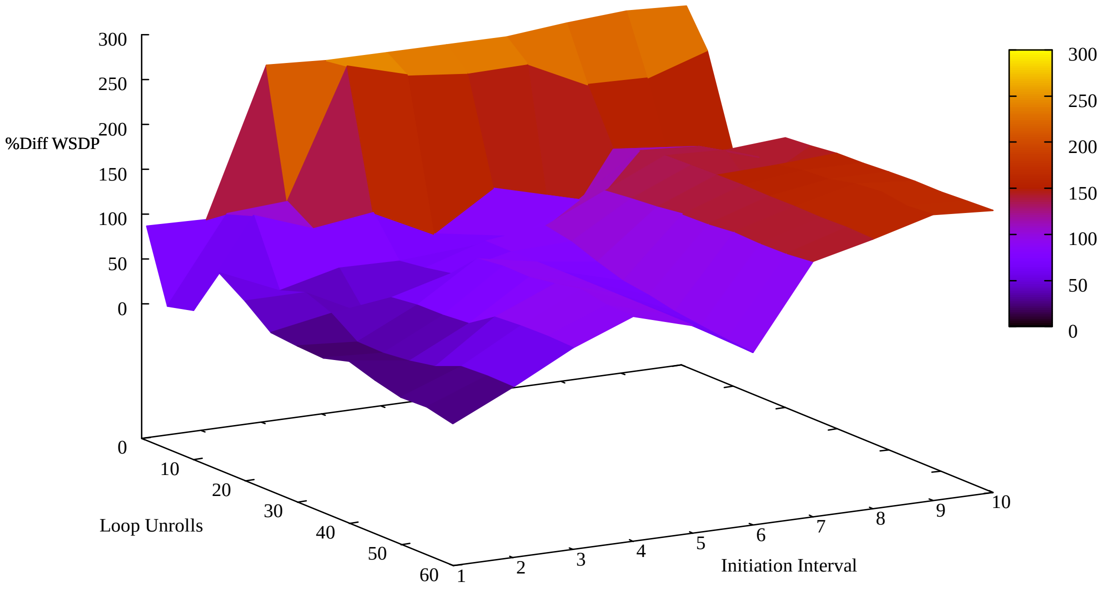


Fig. 5.27: Change in WSDP between the solver (version 4) and the pipelined FDS algorithm for FFT data flow graph.

Chapter 6

Area Estimation and Verilog Generator

Once all the nodes present in the data flow graph are scheduled, CHARGER begins the architecture generation process. This process is carried out in various phases, which includes, memory allocation, binding, architecture analysis and controller generation. Memory allocation examines the schedule of the operations and determines when data elements are need to be stored for more than one clock cycle between their production and consumption. CHARGER employs pipelined shift registers to store these intermediate data. The binding phase creates the functional unit shift register instances and determines which operation should execute on what instance. The use of pipelined functional units and shift registers facilitates efficient resource reuse. The architecture analysis phase performs the area estimation of multiplexers and shift registers (jointly referred as "routing resources"). Controller generation generates a synthesizable VHDL code for the controller. These phases are discussed in detailed by Areno [22].

This chapter discusses in detail the modifications done to the architecture analysis and controller generation approach taken in original CHARGER tool. CHARGER uses Xilinx CORE generators [31] to generate functional units and shift registers. By using Xilinx IP coregen components, various customizations like latency, number of pipelined stages, and use of DSP48 units can be made which directly affect the area of the generated architectures and provides flexibility to the designs. This facility can help the designer to make proper design choices. Currently, CHARGER targets the Xilinx Virtex4 family of FPGAs. CHARGER area estimates do not attempt to estimate the area of the controller and routing overhead.

6.1 Area Estimation

CHARGER performs presynthesis estimation of the throughput and resource consumption of the architectures generated based on the schedule generated by both the CP based scheduler and FDS scheduler. CHARGER provides an estimate of the cost of implementing the resulting architectures in terms of look-up-tables (LUTs), flip flops (FFs), DSP48 units, and block RAMs (BRAMs). It also provides a throughput estimate of the generated architectures. CHARGER attempts to estimate the total area consumption of the design by adding post place and route area estimates of the functional units with the area estimates of routing resources required in the generated architecture. This architecture analysis is carried out by CHARGER after the memory allocation and binding phase. Area estimation of routing resources consists of the multiplexer area estimation and shift register area estimation. The proposed area estimation approach does not attempt to estimate the area of the controller.

The multiplexer area estimation, in the original CHARGER tool, was calculated using (6.1) as proposed by Phillips [28].

$$buswidth \times \lceil log_2(w) \rceil \,, \tag{6.1}$$

where *buswidth* represents the width of each input line and w represents the width of the multiplexer. The multiplexer area, as calculated from (6.1), gives a very rough estimation. In this thesis, we have used the multiplexer area estimation algorithm as proposed by Samala and Dasu [29] and is shown in algorithm 6.1.

As mentioned earlier, CHARGER employs pipelined shift registers to store intermediate values. Pipelined shift registers facilitates resource reuse and are very helpful in reducing the width of multiplexers. The Xilinx CORE generator makes use of native SRL16 shift registers to generate the shift registers of various lengths. Area estimation of the shift registers is calculated using (6.2) and (6.3), as proposed by Samala and Dasu [29].

Algorithm 6.1 Multiplexer Area Estimation

```
$let\ n : N \rightarrow \mathbb{R}^+$ be the number of inputs to the multiplexer
$let\ datawidth : N \rightarrow \mathbb{R}^+$ be the width of each input

if($n \leq 1$)
    return 0
Else
{
  $p = n + \lceil log_2(n) \rceil$
  $L = 0$
  $while(\frac{p}{4} \neq 0)$
  {
    $L = L + \frac{p}{4}$
    $p = \frac{p}{4} + p\%4$
  }
  if($p > 1$)
      $L = L + 1$
}
$return(L \times datawidth)$
```

$$LUT_{reg} = \left\lceil \frac{number\ of\ delay\ cycles}{16} \right\rceil \times buswidth \tag{6.2}$$

$$FF_{reg} = \left\lceil \frac{number\ of\ delay\ cycles}{64} \right\rceil \times buswidth \tag{6.3}$$

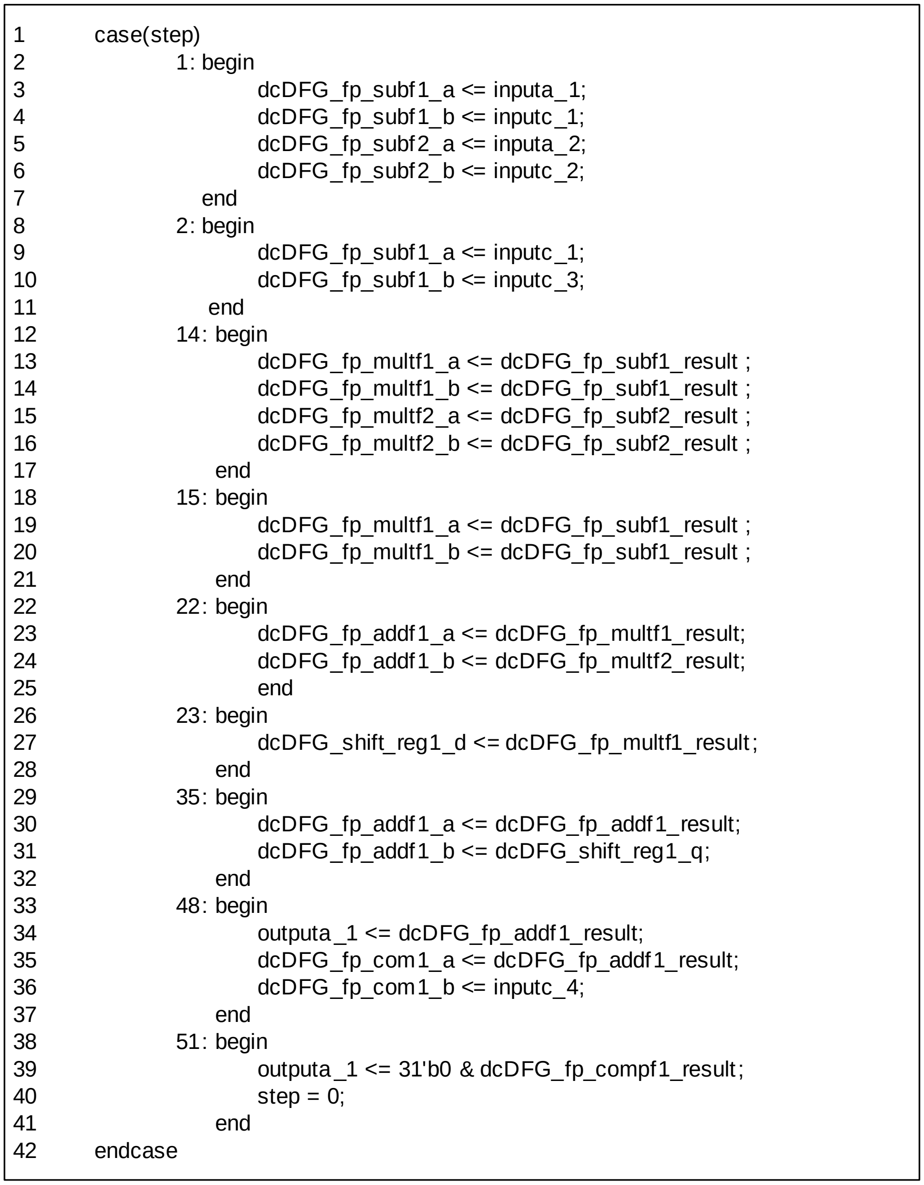
6.2 Controller Generation

The final step in the architecture generation process is the controller generation. The original CHARGER generates a synthesizable VHDL code for the controller. The controller defines a finite state machine which is responsible for properly executing the scheduled data flow graph on an FPGA. The controller also initializes and instantiates functional units and connects signals to functional units required to properly execute the scheduled graph. We modified the approach taken in CHARGER to generate controller and added a Verilog based code generator for the main controller.

The main controller controls all the input and output connections of the functional units and decides which node(s) should begin the execution in the scheduled DFG and also implements the multiplexers needed for the architecture. The approach taken in the original CHARGER tool to generate the code for the controller requires that all the connections to a functional unit should be channeled through the controller. This approach may require additional multiplexers or additional inputs to the multiplexers. CHARGER area estimates for the design do not include area estimate of the controller. Due to the additional resources included in the design by the controller, the difference between actual and estimated total area of the design was quite high. As shown by Areno [22], for some cases this difference was as high as 45%.

In the proposed estimation, we to reduce the difference between actual and estimated area consumption by eliminating the signals which do not need to be channeled through the controller. The finite state machine, implemented by the controller, includes only those signals which are connected to a functional unit through a multiplexer. The remaining signals, which are not connected through a multiplexer, are connected directly to the input of the functional unit. A code snippet of the controller generated by original CHARGER tool is shown in fig. 6.1.

The controller, shown in fig. 6.1, shows the implementation of the finite state machine as case statements. Each case statement represents the schedule step at which one or more nodes should start execution and produce data that needs to be channeled through the controller to its corresponding consumption location. Line (3) of the controller shows that at schedule step 1, *dcDFG_fp_subf1_a* takes input from *inputa_1* port. Line (9) of the controller shows that at schedule step 2, *dcDFG_fp_subf1_a* takes input from the same input port. So, there is no need for a multiplexer interface to the input line *a* of the functional unit *dcDFG_fp_sub1.* When the controller is synthesized using Xilinx ISE, it time multiplexes *inputa_1* and adds a multiplexer to one input line of the functional unit *dcDFG_fp_subf1.* Similarly, for the functional unit *dcDFG_fp_mulf1*, controller adds a multiplexer interface to both the input lines. These additional multiplexers increase the overall area consumption of the resulting architecture, and therefore the percentage of error between estimated and



```
case(step)
        1: begin
                dcDFG_fp_subf1_a <= inputa_1;
                dcDFG_fp_subf1_b <= inputc_1;
                dcDFG_fp_subf2_a <= inputa_2;
                dcDFG_fp_subf2_b <= inputc_2;
            end
        2: begin
                dcDFG_fp_subf1_a <= inputc_1;
                dcDFG_fp_subf1_b <= inputc_3;
            end
        14: begin
                dcDFG_fp_multf1_a <= dcDFG_fp_subf1_result ;
                dcDFG_fp_multf1_b <= dcDFG_fp_subf1_result ;
                dcDFG_fp_multf2_a <= dcDFG_fp_subf2_result ;
                dcDFG_fp_multf2_b <= dcDFG_fp_subf2_result ;
            end
        15: begin
                dcDFG_fp_multf1_a <= dcDFG_fp_subf1_result ;
                dcDFG_fp_multf1_b <= dcDFG_fp_subf1_result ;
            end
        22: begin
                dcDFG_fp_addf1_a <= dcDFG_fp_multf1_result;
                dcDFG_fp_addf1_b <= dcDFG_fp_multf2_result;
                end
        23: begin
                dcDFG_shift_reg1_d <= dcDFG_fp_multf1_result;
            end
        35: begin
                dcDFG_fp_addf1_a <= dcDFG_fp_addf1_result;
                dcDFG_fp_addf1_b <= dcDFG_shift_reg1_q;
            end
        48: begin
                outputa_1 <= dcDFG_fp_addf1_result;
                dcDFG_fp_com1_a <= dcDFG_fp_addf1_result;
                dcDFG_fp_com1_b <= inputc_4;
            end
        51: begin
                outputa_1 <= 31'b0 & dcDFG_fp_compf1_result;
                step = 0;
            end
endcase
```

Fig. 6.1: A Verilog code snippet of controller generated by original CHARGER tool.

actual area consumption is very high in the original CHARGER.

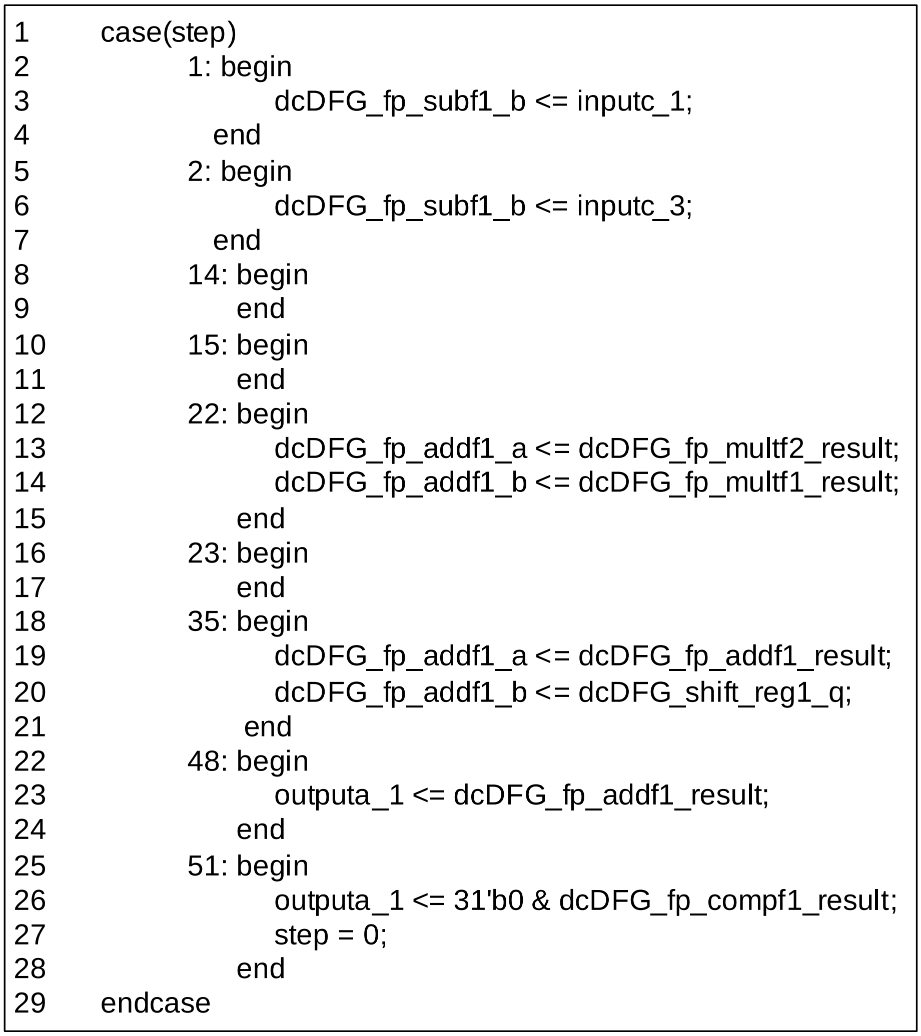
A code snippet of the controller generated using the proposed approach is shown in fig. 6.2. In the proposed approach, the functional units which do not need a multiplexer interface at the input are eliminated from the controller. The input of these functional units can be directly connected to the output of one of the other functional units. Line (3) of the controller, shown in fig. 6.2, shows that at input line *b* of the functional unit *dcDFG_fp_subf1* takes input from *inputc_1* at schedule step 1 and from *inputc_3* at schedule step 2. A multiplexer interface is needed at the functional unit *dcDFG_fp_subf1*, as it takes input from two different input ports. In this case, controller needs to control the input signals of the functional unit *dcDFG_fp_subf1*. Therefore, this data needs to be channeled through the controller as it implements both control logic and multiplexer. Also it can be seen from the code snippet, shown in fig. 6.2, that the controller eliminated *dcDFG_fp_sub1_a* signal because no multiplexer interface is needed for that signal and it can be directly connected to *inputa_1*. This approach eliminates the additional multiplexers that were created by the controller in the original CHARGER and thus reduces the error between actual and estimated area consumption.

6.3 Comparison of the Estimation Approach

This section discusses the effectiveness of the new approach in estimating the total area consumption of the derived architecture. As mentioned earlier, the total area consumption of the design is composed of post place and route area estimates of each functional unit coupled with the area estimates of multipliers and shift registers, as calculated from algorithm (6.1), (6.2), and (6.3), respectively. The hardware architectures derived from CHARGER are synthesized and post place&route results were obtained, using Xilinx ISE 10.1 targeting Xilinx Virtex4 XC4VLX200 device.

The estimated values were compared against the post place&route values for the generated architectures. Figure 6.3 and fig. 6.4 shows the percentage difference between actual and estimated total area in terms of device primitives (#LUTs and #FFs). Positive values in the graph indicates that the estimated values are more as compared to post place&route values produced by Xilinx ISE.

From the plots, it can be seen that the modified controller generation technique, coupled with the area estimation approach, is effective in estimating the area of the generated architectures. The maximum percentage error in LUT estimation was nearly 10% and the maximum percentage error in FF estimation was nearly 6%.



```
case(step)
        1: begin
                dcDFG_fp_subf1_b <= inputc_1;
          end
        2: begin
                dcDFG_fp_subf1_b <= inputc_3;
          end
        14: begin
            end
        15: begin
            end
        22: begin
                dcDFG_fp_addf1_a <= dcDFG_fp_multf2_result;
                dcDFG_fp_addf1_b <= dcDFG_fp_multf1_result;
            end
        23: begin
            end
        35: begin
                dcDFG_fp_addf1_a <= dcDFG_fp_addf1_result;
                dcDFG_fp_addf1_b <= dcDFG_shift_reg1_q;
             end
        48: begin
                outputa_1 <= dcDFG_fp_addf1_result;
            end
        51: begin
                outputa_1 <= 31'b0 & dcDFG_fp_compf1_result;
                step = 0;
            end
endcase
```

Fig. 6.2: A Verilog code snippet of controller generated by current CHARGER tool.

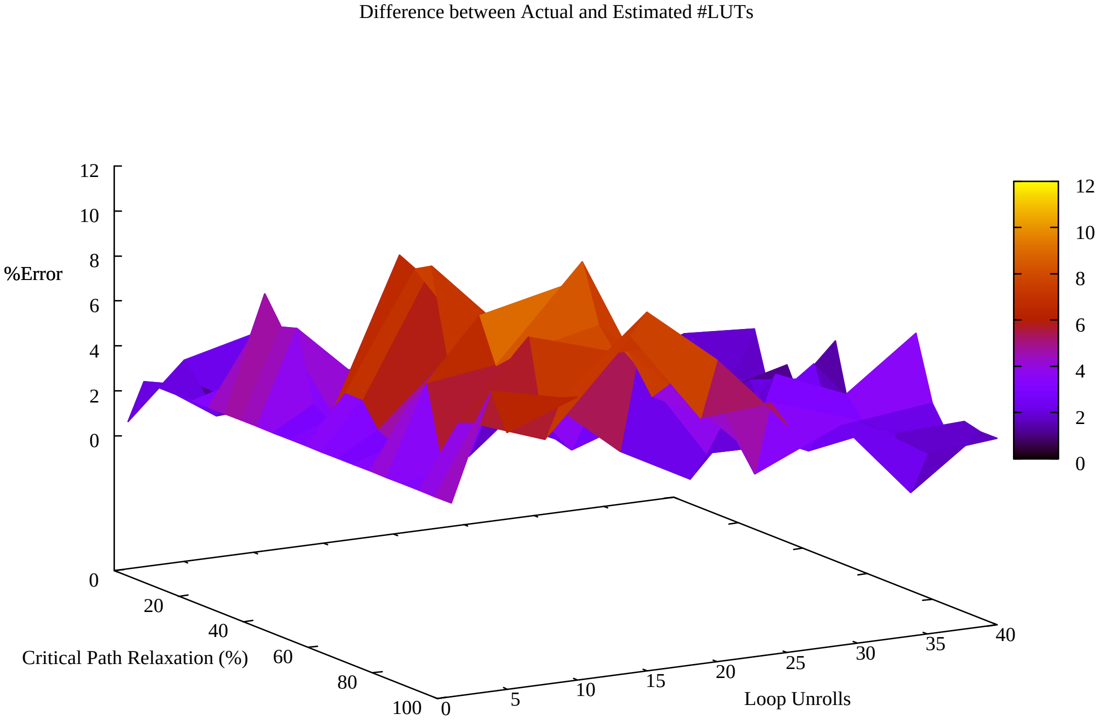


Fig. 6.3: % Difference between actual and estimated area of LUTs.

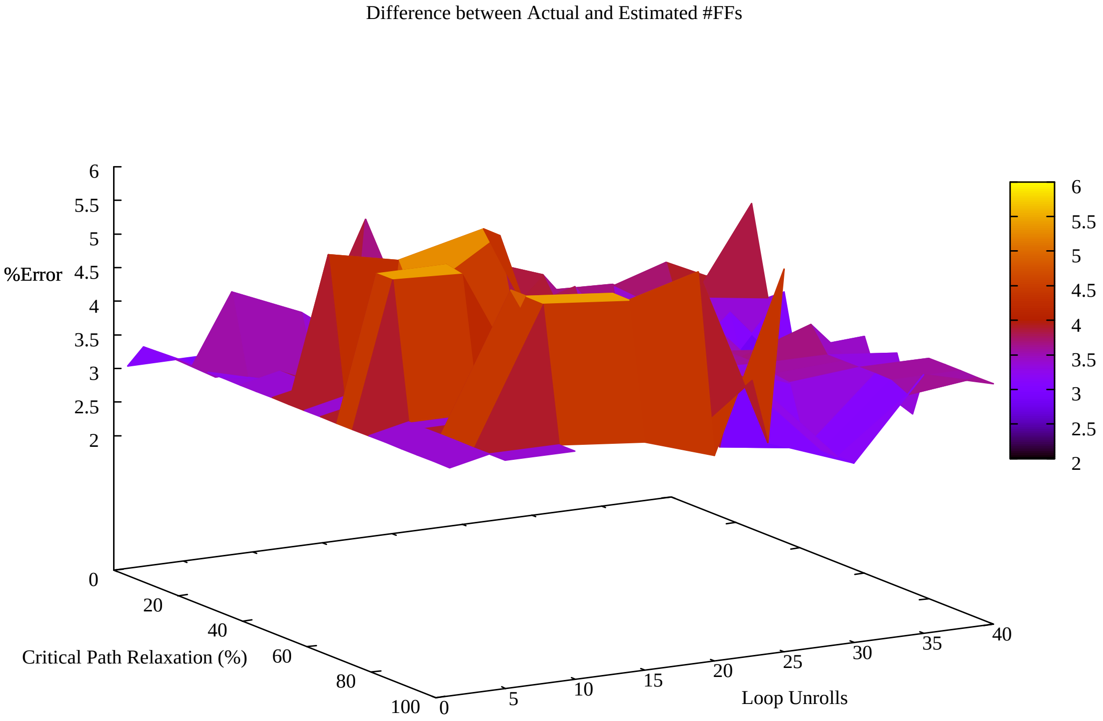


Fig. 6.4: % Difference between actual and estimated area of FFs.

Chapter 7

Conclusion and Future Work

In this thesis, we have presented an extension to CHARGER, a tool-flow which supports the automatic generation of hardware architectures with support for graph replication and critical path relaxation. The original CHARGER makes use of a novel adaptation of the FDS algorithm which supports generation of pipelined architectures when initiation interval is a fractional value. For integer-based initiation intervals, CHARGER uses the pipelined variant of the FDS algorithm to schedule data flow graph and generate hardware architectures based on the produced schedule. We have extended the original CHARGER tool flow to include a constraint programming-based scheduler to generate better architectures, as compared to those based on the variants of FDS algorithm.

In this thesis, we have presented the results that evaluate the performance of the constraint programming-based scheduler, when compared to the both non-pipelined and pipelined variants of the FDS scheduler based tool flow. The results indicate that in terms of throughput per unit area and WSDP, the constraint programming-based scheduler outperforms both pipelined and non-pipelined versions of the FDS scheduler.

The main purpose of including a constraint-based scheduler to the tool flow was to generate optimal schedules for streaming applications and determine their impact on the efficiency and area of the developed architectures. We presented our constraint programming-based scheduler which models the scheduling problem using a set of constraints and determines an allocation and schedule for a given DFG, using a backtracking search with a series of propagation and distribution steps. We extended our CP-based scheduler by adding some constraints which were originally developed for instruction scheduling. We modified these constraints to fit our approach and evaluated their impact on the scheduler performance and scalability. The addition of CP techniques helped in improving the performance of the constraint solver. We compared the performance of our CP-based scheduler against both the modified FDS and pipelined FDS schedulers, and from the results, we observed that architectures generated using the CP-based scheduler are more efficient and consume less area as compared to FDS-generated architectures.

Another contribution of this thesis was a modified Verilog-based controller generator to reduce the percentage error in actual and estimated area consumption. The modified controller helped in reducing the difference between estimated and actual area consumption by removing the connections that do not require a multiplexer interface to the functional units, and thus eliminates the need of additional multiplexers required to execute the derived schedule. The percentage of error between estimation and actual area consumption was reduced from 45% to 10% on tested kernels.

Although the current constraint programming-based tool flow is robust and scalable, there are some areas which need additional work so that the performance of the constraint solver can be improved. The current tool flow does not support multiple latency/area pairs for a single resource. Including these features in the CP-based scheduler will take the search in different directions and will have a significant impact on the search time, resource utilization, and throughput of the resulting architectures. The current tool flow does not handle any loop carried dependecies when unrolling the data flow graph. Also, the current tool only targets basic blocks present in an application and does not have any support for scheduling of the blocks with conditional branches. These additional features will enable the CP-based scheduler to extract parallel portions of the code across the blocks and provide the ability to determine a global optimal schedule and generate efficient architectures at the system level.

Another aspect of the tool flow that needs to be addressed is the approach taken to generate hardware architectures. In the current approach, a schedule, with minimum WSDP, is determined first, and then based on the produced schedule, neccessary hardware (i.e., number of delay registers and multiplexers) is determined to facilitate the architecture generation process. Thus, resulting generated architeture may not be the optimal one because the area consumed by delay registers and multiplexers was not considered during the search. A simultaneous determination of schedule and hardware allocation will allow the solver to include this overhead during the search when calculating WSDP and try to minimize the overall WSDP to generate optimal architectures.

References

[1] K. Marriot and P. J. Stuckey, *Programming with Constraints: An Introduction.* Cambridge: The MIT Press, 1998.

[2] B. Pangrle and D. Gajski, "Design tools for intelligent silicon compilation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 6, no. 6, pp. 1098–1112, Nov. 1987.

[3] W. Sun, M. J. Wirthlin, and S. Neuendorffer, "Combining module selection and resource sharing for efficient fpga pipeline synthesis," in *Proceedings of the 2006 ACM/SIGDA 14th International Symposium on Field Programmable Gate Arrays*, pp. 179–188. New York: Association for Computing Machinery, 2006.

[4] P. V. Beek and K. D. Wilken, "Fast optimal instruction scheduling for single-issue processors with arbitrary latencies," in *Proceedings of the 7th International Conference on Principles and Practice of Constraint Programming*, pp. 625–639. London, UK: Springer-Verlag, 2001.

[5] A. M. Malik, J. McInnes, and P. V. Beek, "Optimal basic block instruction scheduling for multiple-issue processors using constraing programming," in *Proceedings of the 18th IEEE International Conference on Tools with Artificial Intelligence*, pp. 279–287, Washington, DC, 2006.

[6] C.-T. Hwang, J.-H. Lee, and Y.-C. Hsu, "A formal approach to the scheduling problem in high level synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 10, no. 4, pp. 464–475, Apr. 1991.

[7] K. Kuchcinski, "Constraints-driven scheduling and resource assignment," *ACM Transactions on Design Automation of Electronic Systems*, vol. 8, no. 3, pp. 355–383, 2003.

[8] P. Paulin and J. Knight, "Force-directed scheduling for the behavioral synthesis of asics," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 8, no. 6, pp. 661–679, June 1989.

[9] P. Pauline and J. Knight, "Force-directed scheduling in automatic data path synthesis," in *Design Automation, 1987, 24th Conference*, pp. 195–202, June 1987.

[10] R. P. Dick and N. K. Jha, "Mogac: A multiobjective genetic algorithm for the co-synthesis of hardware-software embedded systems," *International Conference on Computer-Aided Design*, p. 522, 1997.

[11] K. Bazargan, R. Kastner, and M. Sarrafzadeh, "3-d floorplanning: Simulated annealing and greedy placement methods for reconfigurable computing systems," in *Proceedings of the Tenth IEEE International Workshop on Rapid System Prototyping*, p. 38. Washington, DC: IEEE Computer Society, 1999.

[12] B. R. Rau, “Iterative modulo scheduling: an algorithm for software pipelining loops,” in *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pp. 63–74. New York: Association of Computing Machinery, 1994.

[13] N. Bellas, S. M. Chai, M. Dwyer, and D. Linzmeier, “Mapping streaming architectures on reconfigurable platforms,” *Special Interest Group on Computer Architecture, Computer Architecture News*, vol. 35, no. 3, pp. 2–8, 2007.

[14] M. Ahmed and H. Abdel-Malek, “Scheduling in high-level synthesis using a hybrid constraint logic programming /integer programming approach,” in *The 2006 International Conference on Computer Engineering and Systems*, pp. 127–131, Nov. 2006.

[15] K. Wilken, J. Liu, and M. Heffernan, “Optimal instruction scheduling using integer programming,” *Special Interest Group in Programming Languages*, vol. 35, no. 5, pp. 121–133, 2000.

[16] C. Inc., “Handel c compiler,” `http://www.celoxica.com/`, 2009.

[17] A. Dellson, “Programming fpgas for high-performance computing acceleration,” `http://www.mitrionics.com/`, 2005.

[18] I. Impulse Accelerated Technologies, “Impulse c,” `http://www.impulsec.com/`, 2009.

[19] A. Jones, D. Bagchi, S. Pal, X. Tang, A. Choudhary, and P. Banerjee, “Pact hdl: a c compiler targeting asics and fpgas with power and performance optimizations,” in *Proceedings of the 2002 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pp. 188–197. New York: Association of Compting Machinery, 2002.

[20] T. Harriss, R. Walke, and B. Kienhuis, “Compilation from matlab to process networks realized in fpga,” in *Proceedings of the 35th Asilomar Conference on Signals, Systems, and Computers*, pp. 385–403, 2002.

[21] L. M. Reyneri, F. Cucinotta, A. Serra, and L. Lavagno, “A hardware/software co-design flow and ip library based on simulink,” in *Proceedings of the 38th annual Design Automation Conference*, pp. 593–598. New York: Association of Computing Machinery, 2001.

[22] M. C. Areno, “Automated constrain-based hardware architecture generation for reconfigurable computing system,” Master’s thesis, Utah State University, Logan, 2007.

[23] G. Smolka, “The oz programming model,” in *Computer Science Today, Lecture Notes in Computer Science*, pp. 324–343. Heidelberg, Berlin: Springer-Verlag, 1995.

[24] Mozart, “Mozart programming system,” `http://www.mozart-oz.org/`, 2009.

[25] C.-G. Quimper, A. Golynski, A. López-Ortiz, and P. Beek, “An efficient bounds consistency algorithm for the global cardinality constraint,” *Constraints*, vol. 10, no. 2, pp. 115–135, 2005.

[26] P. Baptiste, C. Le Pape, and W. Nuijten, *Constraint-Based Scheduling: Applying Constraint Programming to Scheduling Problems.* Norwell, Massachusetts: Kluwer, 2001.

[27] M. Heffernan and K. Wilken, "Data-dependency graph transformations for instruction scheduling," *Journal of Scheduling*, vol. 8, no. 5, pp. 427–451, 2005.

[28] J. Phillips, *A C to Register Transfer Level Algorithm Using Structured Circuit Templates: A Case Study with Simulated Annealing.* Ph.D. dissertation, Utah State University, Logan. UT, 2008.

[29] H. Samala and A. Dasu, "Methodology to derive resource aware context adaptable architectures for fpgas," *IET/IEE Computers and Digital Techniques*, vol. 4, no. 2, pp. 1–15, 2010.

[30] M. Areno, B. Eames, and J. Templin, "A force-directed scheduling based architecture generation algorithm and design tool for fpgas," *Journal of Systems Architecture*, to appear, 2010.

[31] Xilinx, "Xilinx ise foundation," `http://www.xilinx.com/`, 2009.

Appendices

Appendix A

Architecture Description File

A.1 Architecture Description File for Data Flow Graphs

```
OPERATIONS
inputa 1:0:0:0:0
inputc 1:0:0:0:0
outputa 1:0:0:0:0
outputc 1:0:0:0:0
addf 13:572:579:0:0
subf 13:572:579:0:0
multf 8:124:188:4:0
divf 26:819:709:0:0
compf 3:86:19:0:0
CONSTRAINTS
Latency 10000
Area 1000000
Buswidth 32
Device xc4vlx25
Family Virtex4
Speed -11
Package ff668
NumDeviceDSPs 96
NumDeviceLUTs 178176
NumDeviceBRAMs 336
NumDeviceFFs 178176
```

A.2 Data Flow Graph File for Discrete Cosine Transform

```
NODE 0 inputa
NODE 1 inputa
NODE 2 inputa
NODE 3 inputa
NODE 4 inputa
NODE 5 inputa
NODE 6 inputa
NODE 7 inputa
NODE 8 subf
NODE 9 addf
NODE 10 subf
NODE 11 addf
NODE 12 subf
NODE 13 addf
NODE 14 subf
NODE 15 addf
NODE 16 inputc
NODE 17 inputc
NODE 18 inputc
NODE 19 inputc
NODE 20 multf
NODE 21 multf
NODE 22 multf
NODE 23 multf
NODE 24 subf
NODE 25 subf
NODE 26 addf
NODE 27 addf
NODE 28 subf
NODE 29 subf
NODE 30 addf
NODE 31 addf

NODE 32 inputc

NODE 33 inputc

NODE 34 inputc

NODE 35 inputc

NODE 36 multf

NODE 37 multf

NODE 38 multf

NODE 39 multf

NODE 40 subf

NODE 41 subf

NODE 42 subf

NODE 43 subf

NODE 44 subf

NODE 45 subf

NODE 46 addf

NODE 47 addf

NODE 48 addf

NODE 49 addf

NODE 50 inputc

NODE 51 inputc

NODE 52 inputc

NODE 53 inputc

NODE 54 inputc

NODE 55 multf

NODE 56 multf

NODE 57 multf

NODE 58 multf

NODE 59 multf

NODE 60 subf

NODE 61 subf

NODE 62 subf
NODE 63 outputa
NODE 64 outputa
NODE 65 outputa
NODE 66 outputa
NODE 67 outputa
NODE 68 outputa
NODE 69 outputa
NODE 70 outputa
CONNECTION 0 8 L
CONNECTION 1 8 R
CONNECTION 0 9 L
CONNECTION 1 9 R
CONNECTION 2 10 L
CONNECTION 3 10 R
CONNECTION 2 11 L
CONNECTION 3 11 R
CONNECTION 4 12 L
CONNECTION 5 12 R
CONNECTION 4 13 L
CONNECTION 5 13 R
CONNECTION 6 14 L
CONNECTION 7 14 R
CONNECTION 6 15 L
CONNECTION 7 15 R
CONNECTION 8 20 L
CONNECTION 16 20 R
CONNECTION 10 21 L
CONNECTION 17 21 R
CONNECTION 12 22 L
CONNECTION 18 22 R
CONNECTION 14 23 L
CONNECTION 19 23 R
CONNECTION 20 24 L

CONNECTION 21 24 R

CONNECTION 9 25 L

CONNECTION 11 25 R

CONNECTION 20 26 L

CONNECTION 21 26 R

CONNECTION 9 27 L

CONNECTION 11 27 R

CONNECTION 22 28 L

CONNECTION 23 28 R

CONNECTION 13 29 L

CONNECTION 15 29 R

CONNECTION 22 30 L

CONNECTION 23 30 R

CONNECTION 13 31 L

CONNECTION 15 31 R

CONNECTION 24 36 L

CONNECTION 32 36 R

CONNECTION 25 37 L

CONNECTION 33 37 R

CONNECTION 28 38 L

CONNECTION 34 38 R

CONNECTION 29 39 L

CONNECTION 35 39 R

CONNECTION 36 40 L

CONNECTION 26 40 R

CONNECTION 38 41 L

CONNECTION 30 41 R

CONNECTION 40 42 L

CONNECTION 41 42 R

CONNECTION 37 43 L

CONNECTION 39 43 R
CONNECTION 26 44 L

CONNECTION 30 44 R

CONNECTION 27 45 L

CONNECTION 31 45 R

CONNECTION 40 46 L

CONNECTION 41 46 R

CONNECTION 37 47 L

CONNECTION 39 47 R

CONNECTION 26 48 L

CONNECTION 30 48 R

CONNECTION 27 49 L

CONNECTION 31 49 R

CONNECTION 42 55 L

CONNECTION 50 55 R

CONNECTION 43 56 L

CONNECTION 51 56 R

CONNECTION 44 57 L

CONNECTION 52 57 R

CONNECTION 45 58 L

CONNECTION 53 58 R

CONNECTION 49 59 L

CONNECTION 54 59 R

CONNECTION 55 60 L

CONNECTION 48 60 R

CONNECTION 56 61 L

CONNECTION 47 61 R

CONNECTION 57 62 L

CONNECTION 46 62 R

CONNECTION 60 63 L

CONNECTION 61 64 L

CONNECTION 62 65 L

CONNECTION 58 66 L
CONNECTION 46 67 L

CONNECTION 47 68 L

CONNECTION 48 69 L

CONNECTION 59 70 L
```

Appendix B

Sample FDS and Constraint Solver Scheduler Output

B.1 FDS Generated Generated Schedule for DCT

```
Node 0 operation:  inputa step 0 (baseID, Itr):  (0, 0)
Node 1 operation:  inputa step 0 (baseID, Itr):  (1, 0)
Node 2 operation:  inputa step 0 (baseID, Itr):  (2, 0)
Node 3 operation:  inputa step 0 (baseID, Itr):  (3, 0)
Node 4 operation:  inputa step 0 (baseID, Itr):  (4, 0)
Node 5 operation:  inputa step 0 (baseID, Itr):  (5, 0)
Node 6 operation:  inputa step 0 (baseID, Itr):  (6, 0)
Node 7 operation:  inputa step 0 (baseID, Itr):  (7, 0)
Node 8 operation:  subf step 1 (baseID, Itr):  (8, 0)
Node 9 operation:  addf step 1 (baseID, Itr):  (9, 0)
Node 10 operation:  subf step 1 (baseID, Itr):  (10, 0)
Node 11 operation:  addf step 1 (baseID, Itr):  (11, 0)
Node 12 operation:  subf step 1 (baseID, Itr):  (12, 0)
Node 13 operation:  addf step 1 (baseID, Itr):  (13, 0)
Node 14 operation:  subf step 1 (baseID, Itr):  (14, 0)
Node 15 operation:  addf step 1 (baseID, Itr):  (15, 0)
Node 16 operation:  inputc step 0 (baseID, Itr):  (16, 0)
Node 17 operation:  inputc step 0 (baseID, Itr):  (17, 0)
Node 18 operation:  inputc step 0 (baseID, Itr):  (18, 0)
Node 19 operation:  inputc step 0 (baseID, Itr):  (19, 0)
Node 20 operation:  multf step 14 (baseID, Itr):  (20, 0)
Node 21 operation:  multf step 14 (baseID, Itr):  (21, 0)
Node 22 operation:  multf step 14 (baseID, Itr):  (22, 0)
Node 23 operation:  multf step 14 (baseID, Itr):  (23, 0)
Node 24 operation:  subf step 22 (baseID, Itr):  (24, 0)
Node 25 operation:  subf step 14 (baseID, Itr):  (25, 0)
Node 26 operation:  addf step 23 (baseID, Itr):  (26, 0)
Node 27 operation:  addf step 14 (baseID, Itr):  (27, 0)
Node 28 operation:  subf step 22 (baseID, Itr):  (28, 0)
Node 29 operation:  subf step 14 (baseID, Itr):  (29, 0)
Node 30 operation:  addf step 22 (baseID, Itr):  (30, 0)
Node 31 operation:  addf step 43 (baseID, Itr):  (31, 0)
Node 32 operation:  inputc step 0 (baseID, Itr):  (32, 0)
Node 33 operation:  inputc step 0 (baseID, Itr):  (33, 0)
Node 34 operation:  inputc step 0 (baseID, Itr):  (34, 0)
Node 35 operation:  inputc step 0 (baseID, Itr):  (35, 0)
Node 36 operation:  multf step 35 (baseID, Itr):  (36, 0)
Node 37 operation:  multf step 27 (baseID, Itr):  (37, 0)
Node 38 operation:  multf step 35 (baseID, Itr):  (38, 0)
Node 39 operation:  multf step 27 (baseID, Itr):  (39, 0)
Node 40 operation:  subf step 43 (baseID, Itr):  (40, 0)
Node 41 operation:  subf step 43 (baseID, Itr):  (41, 0)
Node 42 operation:  subf step 56 (baseID, Itr):  (42, 0)
Node 43 operation:  subf step 35 (baseID, Itr):  (43, 0)
Node 44 operation:  subf step 36 (baseID, Itr):  (44, 0)
Node 45 operation:  subf step 57 (baseID, Itr):  (45, 0)
Node 46 operation:  addf step 56 (baseID, Itr):  (46, 0)
Node 47 operation:  addf step 35 (baseID, Itr):  (47, 0)
Node 48 operation:  addf step 40 (baseID, Itr):  (48, 0)
Node 49 operation:  addf step 69 (baseID, Itr):  (49, 0)
Node 50 operation:  inputc step 0 (baseID, Itr):  (50, 0)
Node 51 operation:  inputc step 0 (baseID, Itr):  (51, 0)
Node 52 operation:  inputc step 0 (baseID, Itr):  (52, 0)
Node 53 operation:  inputc step 0 (baseID, Itr):  (53, 0)
Node 54 operation:  inputc step 0 (baseID, Itr):  (54, 0)
Node 55 operation:  multf step 69 (baseID, Itr):  (55, 0)
Node 56 operation:  multf step 56 (baseID, Itr):  (56, 0)
Node 57 operation:  multf step 49 (baseID, Itr):  (57, 0)
Node 58 operation:  multf step 70 (baseID, Itr):  (58, 0)
Node 59 operation:  multf step 82 (baseID, Itr):  (59, 0)
Node 60 operation:  subf step 77 (baseID, Itr):  (60, 0)
Node 61 operation:  subf step 64 (baseID, Itr):  (61, 0)
Node 62 operation:  subf step 69 (baseID, Itr):  (62, 0)
Node 63 operation:  outputa step 90 (baseID, Itr):  (63, 0)
Node 64 operation:  outputa step 77 (baseID, Itr):  (64, 0)
Node 65 operation:  outputa step 83 (baseID, Itr):  (65, 0)
Node 66 operation:  outputa step 82 (baseID, Itr):  (66, 0)
Node 67 operation:  outputa step 69 (baseID, Itr):  (67, 0)
Node 68 operation:  outputa step 52 (baseID, Itr):  (68, 0)
Node 69 operation:  outputa step 56 (baseID, Itr):  (69, 0)
Node 70 operation:  outputa step 90 (baseID, Itr):  (70, 0)
Op inputa 8 instances, 0 total slices
Op inputc 13 instances, 0 total slices
Op outputa 2 instances, 0 total slices
Op outputc 0 instances, 0 total slices
Op addf 4 instances, 2288 total slices
Op subf 4 instances, 2288 total slices
Op multf 4 instances, 496 total slices
Op divf 0 instances, 0 total slices
Op compf 0 instances, 0 total slices
Total area consumed:  5072
Total cycles required:  91
Original CP Length:  91
```

B.2 The Constraint Solver Generated Generated Schedule for DCT

```
Node 0 operation:  inputa step 0 (baseID, Itr):  (0, 1)
Node 1 operation:  inputa step 0 (baseID, Itr):  (1, 1)
Node 2 operation:  inputa step 0 (baseID, Itr):  (2, 1)
Node 3 operation:  inputa step 0 (baseID, Itr):  (3, 1)
Node 4 operation:  inputa step 0 (baseID, Itr):  (4, 1)
Node 5 operation:  inputa step 0 (baseID, Itr):  (5, 1)
Node 6 operation:  inputa step 0 (baseID, Itr):  (6, 1)
Node 7 operation:  inputa step 0 (baseID, Itr):  (7, 1)
Node 8 operation:  subf step 1 (baseID, Itr):  (8, 1)
Node 9 operation:  addf step 2 (baseID, Itr):  (9, 1)
Node 10 operation:  subf step 1 (baseID, Itr):  (10, 1)
Node 11 operation:  addf step 1 (baseID, Itr):  (11, 1)
Node 12 operation:  subf step 1 (baseID, Itr):  (12, 1)
Node 13 operation:  addf step 4 (baseID, Itr):  (13, 1)
Node 14 operation:  subf step 1 (baseID, Itr):  (14, 1)
Node 15 operation:  addf step 3 (baseID, Itr):  (15, 1)
Node 16 operation:  inputc step 0 (baseID, Itr):  (16, 1)
Node 17 operation:  inputc step 0 (baseID, Itr):  (17, 1)
Node 18 operation:  inputc step 0 (baseID, Itr):  (18, 1)
Node 19 operation:  inputc step 0 (baseID, Itr):  (19, 1)
Node 20 operation:  multf step 14 (baseID, Itr):  (20, 1)
Node 21 operation:  multf step 14 (baseID, Itr):  (21, 1)
Node 22 operation:  multf step 14 (baseID, Itr):  (22, 1)
Node 23 operation:  multf step 14 (baseID, Itr):  (23, 1)
Node 24 operation:  subf step 22 (baseID, Itr):  (24, 1)
Node 25 operation:  subf step 15 (baseID, Itr):  (25, 1)
Node 26 operation:  addf step 22 (baseID, Itr):  (26, 1)
Node 27 operation:  addf step 15 (baseID, Itr):  (27, 1)
Node 28 operation:  subf step 22 (baseID, Itr):  (28, 1)
Node 29 operation:  subf step 17 (baseID, Itr):  (29, 1)
Node 30 operation:  addf step 23 (baseID, Itr):  (30, 1)
Node 31 operation:  addf step 17 (baseID, Itr):  (31, 1)

Node 32 operation:  inputc step 0 (baseID, Itr):  (32, 1)

Node 33 operation:  inputc step 0 (baseID, Itr):  (33, 1)

Node 34 operation:  inputc step 0 (baseID, Itr):  (34, 1)

Node 35 operation:  inputc step 0 (baseID, Itr):  (35, 1)

Node 36 operation:  multf step 35 (baseID, Itr):  (36, 1)

Node 37 operation:  multf step 28 (baseID, Itr):  (37, 1)

Node 38 operation:  multf step 35 (baseID, Itr):  (38, 1)

Node 39 operation:  multf step 30 (baseID, Itr):  (39, 1)

Node 40 operation:  subf step 43 (baseID, Itr):  (40, 1)

Node 41 operation:  subf step 43 (baseID, Itr):  (41, 1)

Node 42 operation:  subf step 56 (baseID, Itr):  (42, 1)

Node 43 operation:  subf step 38 (baseID, Itr):  (43, 1)

Node 44 operation:  subf step 36 (baseID, Itr):  (44, 1)

Node 45 operation:  subf step 30 (baseID, Itr):  (45, 1)

Node 46 operation:  addf step 56 (baseID, Itr):  (46, 1)

Node 47 operation:  addf step 38 (baseID, Itr):  (47, 1)

Node 48 operation:  addf step 36 (baseID, Itr):  (48, 1)

Node 49 operation:  addf step 30 (baseID, Itr):  (49, 1)

Node 50 operation:  inputc step 0 (baseID, Itr):  (50, 1)

Node 51 operation:  inputc step 0 (baseID, Itr):  (51, 1)

Node 52 operation:  inputc step 0 (baseID, Itr):  (52, 1)

Node 53 operation:  inputc step 0 (baseID, Itr):  (53, 1)

Node 54 operation:  inputc step 0 (baseID, Itr):  (54, 1)

Node 55 operation:  multf step 69 (baseID, Itr):  (55, 1)

Node 56 operation:  multf step 51 (baseID, Itr):  (56, 1)

Node 57 operation:  multf step 49 (baseID, Itr):  (57, 1)

Node 58 operation:  multf step 43 (baseID, Itr):  (58, 1)

Node 59 operation:  multf step 43 (baseID, Itr):  (59, 1)

Node 60 operation:  subf step 77 (baseID, Itr):  (60, 1)

Node 61 operation:  subf step 59 (baseID, Itr):  (61, 1)

Node 62 operation:  subf step 69 (baseID, Itr):  (62, 1)
Node 63 operation:  outputa step 90 (baseID, Itr):  (63, 1)

Node 64 operation:  outputa step 72 (baseID, Itr):  (64, 1)

Node 65 operation:  outputa step 82 (baseID, Itr):  (65, 1)

Node 66 operation:  outputa step 51 (baseID, Itr):  (66, 1)

Node 67 operation:  outputa step 69 (baseID, Itr):  (67, 1)

Node 68 operation:  outputa step 51 (baseID, Itr):  (68, 1)

Node 69 operation:  outputa step 49 (baseID, Itr):  (69, 1)

Node 70 operation:  outputa step 51 (baseID, Itr):  (70, 1)

Op inputa 8 instances,0 total slices

Op inputc 13 instances,0 total slices

Op outputa 3 instances,0 total slices

Op addf 1 instances,572 total slices

Op subf 4 instances,2288 total slices

Op multf 4 instances,496 total slices

Total area consumed:  3356

Total cycles required:  91

Original CP Length:  91
```

Appendix C

Sample Verilog Generated Files for DCT Data Flow Graph

```
‘timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:  21:21:55 12/6/2009
// Design Name:
// Design Name:  dfg_DCT
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////
module dfg_DCT(
input clk,
input clr,
input enab,
input [31:0] inputa_1,
input [31:0] inputa_2,
input [31:0] inputa_3,
input [31:0] inputa_4,
input [31:0] inputa_5,
input [31:0] inputa_6,
input [31:0] inputa_7,
input [31:0] inputa_8,
input [31:0] inputc_1,
input [31:0] inputc_2,
input [31:0] inputc_3,
input [31:0] inputc_4,
input [31:0] inputc_5,
input [31:0] inputc_6,
input [31:0] inputc_7,
input [31:0] inputc_8,
input [31:0] inputc_9,
input [31:0] inputc_10,
input [31:0] inputc_11,
input [31:0] inputc_12,
input [31:0] inputc_13,
output reg [31:0] outputa_1,
output reg [31:0] outputa_2,
output reg [31:0] outputa_3
);
reg [31:0] dfg_DCT_fp_addf1_a;
reg [31:0] dfg_DCT_fp_addf1_b;
wire [31:0] dfg_DCT_fp_addf1_result;
reg [31:0] dfg_DCT_fp_subf1_a;
reg [31:0] dfg_DCT_fp_subf1_b;
wire [31:0] dfg_DCT_fp_subf1_result;
reg [31:0] dfg_DCT_fp_subf2_a;
reg [31:0] dfg_DCT_fp_subf2_b;
wire [31:0] dfg_DCT_fp_subf2_result;
```

```
reg [31:0] dfg_DCT_fp_subf3_a;

reg [31:0] dfg_DCT_fp_subf3_b;

wire [31:0] dfg_DCT_fp_subf3_result;

reg [31:0] dfg_DCT_fp_subf4_a;

reg [31:0] dfg_DCT_fp_subf4_b;

wire [31:0] dfg_DCT_fp_subf4_result;

reg [31:0] dfg_DCT_fp_multf1_a;

reg [31:0] dfg_DCT_fp_multf1_b;

wire [31:0] dfg_DCT_fp_multf1_result;

reg [31:0] dfg_DCT_fp_multf2_a;

reg [31:0] dfg_DCT_fp_multf2_b;

wire [31:0] dfg_DCT_fp_multf2_result;

reg [31:0] dfg_DCT_fp_multf3_a;

reg [31:0] dfg_DCT_fp_multf3_b;

wire [31:0] dfg_DCT_fp_multf3_result;

reg [31:0] dfg_DCT_fp_multf4_a;

reg [31:0] dfg_DCT_fp_multf4_b;

wire [31:0] dfg_DCT_fp_multf4_result;

reg [31 :0] dfg_DCT_shift_reg1_d;

wire [31 :0] dfg_DCT_shift_reg1_q;

reg [31 :0] dfg_DCT_shift_reg2_d;

wire [31 :0] dfg_DCT_shift_reg2_q;

reg [31 :0] dfg_DCT_shift_reg3_d;

wire [31 :0] dfg_DCT_shift_reg3_q;

reg [31 :0] dfg_DCT_shift_reg4_d;

wire [31 :0] dfg_DCT_shift_reg4_q;

reg [31 :0] dfg_DCT_shift_reg5_d;

wire [31 :0] dfg_DCT_shift_reg5_q;

reg [31 :0] dfg_DCT_shift_reg6_d;

wire [31 :0] dfg_DCT_shift_reg6_q;

reg [31 :0] dfg_DCT_shift_reg7_d;

wire [31 :0] dfg_DCT_shift_reg7_q;
```

```
reg [31 :0] dfg_DCT_shift_reg8_d;
wire [31 :0] dfg_DCT_shift_reg8_q;
integer step;
reg [31:0] mem_1;
reg [31:0] mem_2;
floating_adder dfg_DCT_fp_addf1(clk, dfg_DCT_fp_addf1_a, dfg_DCT_fp_addf1_b,
dfg_DCT_fp_addf1_result);
floating_subtracter dfg_DCT_fp_subf1(clk, dfg_DCT_fp_subf1_a, dfg_DCT_fp_subf1_b,
dfg_DCT_fp_subf1_result);
floating_subtracter dfg_DCT_fp_subf2(clk, dfg_DCT_fp_subf2_a, dfg_DCT_fp_subf2_b,
dfg_DCT_fp_subf2_result);
floating_subtracter dfg_DCT_fp_subf3(clk, inputa_8, inputa_7,
dfg_DCT_fp_subf3_result);
floating_subtracter dfg_DCT_fp_subf4(clk, inputa_6, inputa_5,
dfg_DCT_fp_subf4_result);
floating_multiplier dfg_DCT_fp_multf1(clk, dfg_DCT_fp_multf1_a, dfg_DCT_fp_multf1_b,
dfg_DCT_fp_multf1_result);
floating_multiplier dfg_DCT_fp_multf2(clk, dfg_DCT_fp_multf2_a, dfg_DCT_fp_multf2_b,
dfg_DCT_fp_multf2_result);
floating_multiplier dfg_DCT_fp_multf3(clk, dfg_DCT_fp_subf3_result, inputc_4,
dfg_DCT_fp_multf3_result);
floating_multiplier dfg_DCT_fp_multf4(clk, dfg_DCT_fp_subf4_result, inputc_5,
dfg_DCT_fp_multf4_result);
shift_register11 dfg_DCT_shift_reg1(clk, dfg_DCT_fp_multf_1_result,
dfg_DCT_shift_reg1_q);
shift_register27 dfg_DCT_shift_reg2(clk, dfg_DCT_fp_addf_1_result,
dfg_DCT_shift_reg2_q);
shift_register7 dfg_DCT_shift_reg3(clk, dfg_DCT_fp_addf_1_result,
dfg_DCT_shift_reg3_q);
shift_register1 dfg_DCT_shift_reg4(clk, dfg_DCT_shift_reg4_d, dfg_DCT_shift_reg4_q);
shift_register1 dfg_DCT_shift_reg5(clk, dfg_DCT_shift_reg5_d, dfg_DCT_shift_reg5_q);
```

```
shift_register6 dfg_DCT_shift_reg6(clk, dfg_DCT_fp_addf_1_result,
dfg_DCT_shift_reg6_q);
shift_register2 dfg_DCT_shift_reg7(clk, inputa_1, dfg_DCT_shift_reg7_q);
shift_register2 dfg_DCT_shift_reg8(clk, inputa_2, dfg_DCT_shift_reg8_q);
always @(inputa_1, inputa_2, inputa_3, inputa_4, inputa_5, inputa_6, inputa_7,
inputa_8, inputc_1, inputc_2, inputc_3, inputc_4, inputc_5, inputc_6, inputc_7,
inputc_8, inputc_9, inputc_10, inputc_11, inputc_12, inputc_13,
dfg_DCT_fp_addf1_result, dfg_DCT_fp_subf1_result, dfg_DCT_fp_subf2_result,
dfg_DCT_fp_subf3_result, dfg_DCT_fp_subf4_result, dfg_DCT_fp_multf1_result,
dfg_DCT_fp_multf2_result,
dfg_DCT_fp_multf3_result, dfg_DCT_fp_multf4_result, dfg_DCT_shift_reg1_q,
dfg_DCT_shift_reg2_q, dfg_DCT_shift_reg3_q, dfg_DCT_shift_reg4_q, dfg_DCT_shift_reg5_q,
dfg_DCT_shift_reg6_q, dfg_DCT_shift_reg7_q, dfg_DCT_shift_reg8_q, clr, enab)
begin
if(clr)
begin
step = 0;
outputa_1 <= 32'b0;
outputa_2 <= 32'b0;
outputa_3 <= 32'b0;
end
else if (!enab)
step = step;
else
begin
step = step + 1;
case(step)
1:  begin
mem_1 <= inputa_5;
mem_2 <= inputa_6;
dfg_DCT_fp_addf1_a <= inputa_8;
dfg_DCT_fp_addf1_b <= inputa_7;
```

```
dfg_DCT_fp_subf1_a <= inputa_4;

dfg_DCT_fp_subf1_b <= inputa_3;

dfg_DCT_fp_subf2_a <= inputa_2;

dfg_DCT_fp_subf2_b <= inputa_1;

end

2:  begin

dfg_DCT_fp_addf1_a <= mem_2;

dfg_DCT_fp_addf1_b <= mem_1;

end

3:  begin

dfg_DCT_fp_addf1_a <= dfg_DCT_shift_reg5_q;

dfg_DCT_fp_addf1_b <= dfg_DCT_shift_reg4_q;

end

4:  begin

dfg_DCT_fp_addf1_a <= dfg_DCT_shift_reg8_q;

dfg_DCT_fp_addf1_b <= dfg_DCT_shift_reg7_q;

end

14:  begin

mem_1 <= dfg_DCT_fp_addf1_result;

dfg_DCT_fp_multf1_a <= dfg_DCT_fp_subf1_result;

dfg_DCT_fp_multf1_b <= inputc_2;

dfg_DCT_fp_multf2_a <= dfg_DCT_fp_subf2_result;

dfg_DCT_fp_multf2_b <= inputc_3;

mem_2 <= dfg_DCT_fp_addf1_result;

end

15:  begin

dfg_DCT_fp_addf1_a <= dfg_DCT_fp_addf1_result;

dfg_DCT_fp_addf1_b <= mem_1;

dfg_DCT_fp_subf1_a <= dfg_DCT_fp_addf1_result;

dfg_DCT_fp_subf1_b <= mem_2;

end

16:  begin
```

```
mem_1 <= dfg_DCT_fp_addf1_result;
mem_2 <= dfg_DCT_fp_addf1_result;
end
17:  begin
dfg_DCT_fp_addf1_a <= dfg_DCT_fp_addf1_result;
dfg_DCT_fp_addf1_b <= mem_1;
dfg_DCT_fp_subf1_a <= dfg_DCT_fp_addf1_result;
dfg_DCT_fp_subf1_b <= mem_2;
end
22:  begin
mem_1 <= dfg_DCT_fp_multf1_result;
mem_2 <= dfg_DCT_fp_multf2_result;
dfg_DCT_fp_addf1_a <= dfg_DCT_fp_multf4_result;
dfg_DCT_fp_addf1_b <= dfg_DCT_fp_multf3_result;
dfg_DCT_fp_subf1_a <= dfg_DCT_fp_multf2_result;
dfg_DCT_fp_subf1_b <= dfg_DCT_fp_multf1_result;
dfg_DCT_fp_subf2_a <= dfg_DCT_fp_multf4_result;
dfg_DCT_fp_subf2_b <= dfg_DCT_fp_multf3_result;
end
23:  begin
dfg_DCT_fp_addf1_a <= mem_2;
dfg_DCT_fp_addf1_b <= mem_1;
end
28:  begin
dfg_DCT_fp_multf1_a <= dfg_DCT_fp_subf1_result;
dfg_DCT_fp_multf1_b <= inputc_7;
end
30:  begin
dfg_DCT_fp_addf1_a <= dfg_DCT_shift_reg4_q;
dfg_DCT_fp_addf1_b <= dfg_DCT_fp_addf1_result;
dfg_DCT_fp_multf1_a <= dfg_DCT_fp_subf1_result;
dfg_DCT_fp_multf1_b <= inputc_6;
```

```
dfg_DCT_fp_subf1_a <= dfg_DCT_shift_reg5_q;
dfg_DCT_fp_subf1_b <= dfg_DCT_fp_addf1_result;
end
35:  begin
mem_1 <= dfg_DCT_fp_addf1_result;
dfg_DCT_fp_multf1_a <= dfg_DCT_fp_subf1_result;
dfg_DCT_fp_multf1_b <= inputc_8;
dfg_DCT_fp_multf2_a <= dfg_DCT_fp_subf2_result;
dfg_DCT_fp_multf2_b <= inputc_9;
mem_2 <= dfg_DCT_fp_addf1_result;
end
36:  begin
dfg_DCT_fp_addf1_a <= mem_1;
dfg_DCT_fp_addf1_b <= dfg_DCT_fp_addf1_result;
dfg_DCT_fp_subf1_a <= mem_2;
dfg_DCT_fp_subf1_b <= dfg_DCT_fp_addf1_result;
end
38:  begin
dfg_DCT_fp_addf1_a <= dfg_DCT_shift_reg4_q;
dfg_DCT_fp_addf1_b <= dfg_DCT_fp_multf1_result;
dfg_DCT_fp_subf1_a <= dfg_DCT_shift_reg5_q;
dfg_DCT_fp_subf1_b <= dfg_DCT_fp_multf1_result;
end
43:  begin
dfg_DCT_fp_multf1_a <= dfg_DCT_fp_addf1_result;
dfg_DCT_fp_multf1_b <= inputc_1;
dfg_DCT_fp_subf1_a <= dfg_DCT_fp_multf1_result;
dfg_DCT_fp_subf1_b <= dfg_DCT_shift_reg6_q;
dfg_DCT_fp_subf2_a <= dfg_DCT_fp_multf2_result;
dfg_DCT_fp_subf2_b <= dfg_DCT_shift_reg3_q;
dfg_DCT_fp_multf2_a <= dfg_DCT_fp_subf1_result;
dfg_DCT_fp_multf2_b <= inputc_10;
```

```
end
49:  begin
outputa_1 <= dfg_DCT_fp_addf1_result;
dfg_DCT_fp_multf1_a <= dfg_DCT_fp_subf1_result;
dfg_DCT_fp_multf1_b <= inputc_11;
end
51:  begin
outputa_1 <= dfg_DCT_fp_multf1_result;
outputa_2 <= dfg_DCT_fp_addf1_result;
outputa_3 <= dfg_DCT_fp_multf2_result;
dfg_DCT_fp_multf1_a <= dfg_DCT_fp_subf1_result;
dfg_DCT_fp_multf1_b <= inputc_12;
end
56:  begin
dfg_DCT_fp_addf1_a <= dfg_DCT_fp_subf2_result;
dfg_DCT_fp_addf1_b <= dfg_DCT_fp_subf1_result;
dfg_DCT_fp_subf1_a <= dfg_DCT_fp_subf2_result;
dfg_DCT_fp_subf1_b <= dfg_DCT_fp_subf1_result;
end
57:  begin
end
59:  begin
dfg_DCT_fp_subf1_a <= dfg_DCT_fp_multf1_result;
dfg_DCT_fp_subf1_b <= dfg_DCT_shift_reg3_q;
end
69:  begin
outputa_1 <= dfg_DCT_fp_addf1_result;
dfg_DCT_fp_subf1_a <= dfg_DCT_shift_reg1_q;
dfg_DCT_fp_subf1_b <= dfg_DCT_fp_addf1_result;
dfg_DCT_fp_multf1_a <= dfg_DCT_fp_subf1_result;
dfg_DCT_fp_multf1_b <= inputc_13;
end
```

```
72:  begin
outputa_1 <= dfg_DCT_fp_subf1_result;
end
77:  begin
dfg_DCT_fp_subf1_a <= dfg_DCT_fp_multf1_result;
dfg_DCT_fp_subf1_b <= dfg_DCT_shift_reg2_q;
end
82:  begin
outputa_1 <= dfg_DCT_fp_subf1_result;
end
90:  begin
outputa_1 <= dfg_DCT_fp_subf1_result;
step = 0;
end
endcase
end
end
endmodule
```