5-2008

A Field Programmable Gate Array Based Finite-Domain Constraint Solver

Prasad Subramanian
Utah State University

A FIELD PROGRAMMABLE GATE ARRAY BASED FINITE-DOMAIN CONSTRAINT SOLVER

by

Prasad Subramanian

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

of

MASTER OF SCIENCE

in

Electrical Engineering

Approved:

Dr. Brandon Eames Major Professor	Dr. Aravind Dasu Committee Member
Dr. Jacob Gunther Committee Member	Dr. Byron R. Burnham Dean of Graduate Studies

UTAH STATE UNIVERSITY
Logan, Utah

2008

Abstract

A Field Programmable Gate Array Based Finite-Domain Constraint Solver

by

Prasad Subramanian, Master of Science

Utah State University, 2008

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

Constraint satisfaction and optimization techniques are commonly employed in scheduling, industrial manufacturing, and automation processes where concepts from Operations Research (OR) and Artificial Intelligence (AI) are applied. In embedded systems, Constraint Satisfaction Problem (CSP) finds use in design, synthesis, and optimization. However, most application areas of CSP employ offline solving techniques where design requirements and constraints are captured before the system is deployed online. There is a significant amount of pre-planning and human intervention required.

In embedded systems, online constraint solving techniques are primarily used as onboard control software in order to enable a system that can dynamically adapt to a changing environment (e.g., autonomous mission planning). This is possible because by using constraint-solving techniques, the constraint model is inherently separated from the underlying algorithms employed for solving the constraints. Due to this separation of concerns, a variety of problems can be solved. The domain of dynamic scheduling is considered in this thesis and can be considered a part of onboard control software for embedded systems. Scheduling problems in particular are amenable to CSP techniques since the discrete start times that govern the execution of various tasks can be formulated as the output of several constraints. In its primitive form, the definition of a schedule includes temporal dependency

constraints and resource constraints. Many dynamic schedulers, however, are also required to take configuration constraints of the system into account.

Using CSP techniques for scheduling algorithms provides intelligent scheduling and enables the embedded system to be more adaptable to dynamic changes in the environment. This thesis discusses the development of a parallel finite-domain constraint solver in order to perform online constraint satisfaction for embedded systems. By modeling the scheduling problem as a CSP problem, the embedded system becomes flexible and adaptable to dynamic changes in the environment since it can accommodate a range of constraints apart from precedence and resource constraints. The features of this solver are that it is implemented in a platform with multiple soft-core processors with distributed memory architecture. The constraints for the application problem are captured from a Data Flow Graph (DFG) and solved on this platform. A tool is also developed that automates the partitioning of the given application and also configures the underlying framework for execution of the CSP problem.

(121 pages)

To my Parents — whose love, support, and prayers helped me pursue this program.

Acknowledgments

This research project proved to be an exciting learning experience and I thank my advisor, Dr. Brandon Eames, for all the support and encouragement I received during every step of the way. I sincerely appreciate his patience and efforts in urging me to move forward even during times of frustration and equipping me with the skills to surmount road-blocks. Constant interactions and guidance from him made my journey through this program a valuable one.

I also appreciate the time devoted by my committee members, Dr. Aravind Dasu and Dr. Jacob Gunther. I would like to acknowledge Dr. Aravind Dasu for providing access to the development boards and tools for implementing this project.

The timely assistance and inputs received from my colleagues in the lab proved very useful. I would like to thank Shant Chandrakar for helping me climb the initial learning curve with MicroBlaze; Jonathan Philips for his opinions and providing the sample application to benchmark the tool. To my friends Arvind Sudarsanam, Rohit Saraswat, and Anand Madhusoodanan who have helped and encouraged me in more ways than I can describe.

Nothing could have been achieved without the support received from my parents. Their encouragement has motivated me to pursue my ambitions far away from home.

To all the staff and faculty members in the Electrical and Computer Engineering Department at Utah State University for the excellent support and infrastructure provided all through the master's program. I would also like to thank the Department of Biology at Utah State University for the opportunity to sustain myself as a graduate student.

Prasad Subramanian

Contents

List of Tables

List of Figures

Acronyms

FPGA	Field Programmable Gate Array
CLP	Constraint Logic Programming
FD	Finite Domain
CP	Constraint Programming
SRQ	Self Referential Quiz
MILP	Mixed Integer Linear Programming
ILP	Integer Linear Programming
CBS	Constraint Based Scheduling
DFG	Data Flow Graph
FSM	Finite State Machine
FSL	Fast Simplex Links
BRAM	Block RAM
CSP	Constraint Satisfaction Problem
GDB	GNU Debugger
OPB	On-chip Peripheral Bus
MB	MicroBlaze
IRQ	Interrupt ReQuest
RISC	Reduced Instruction Set Computer
PAR	Place and Route

Chapter 1

Introduction

Constraint satisfaction and optimization techniques are commonly employed in scheduling problems, industrial manufacturing, and automation processes where it borrows concepts from Operations Research (OR) and Artificial Intelligence (AI). Constraint Satisfaction Problem (CSP) also finds use in design, synthesis, and optimization of embedded systems. However, most application areas of CSP employ offline solving techniques where the design requirements and constraints are captured before the system is deployed or comes online. There is significant amount of pre-planning and human intervention required.

In recent years online constraint solving techniques are being employed in embedded and non-embedded systems for dynamic system adaptation and reconfiguration. For instance, the application of online CSP is demonstrated in the selection of various web services and components dynamically based on the Quality of Service (QoS) specified by the user. The user choice is formulated as a constraint satisfaction problem and the solution to the CSP problem is the run-time selection of web components that meet the QoS requirements [1].

In embedded systems, online constraint solving techniques is primarily used as onboard control software in order to enable a system that can dynamically adapt to a changing environment such as smart printer and autonomous mission planning [2]. This is possible because by using constraint solving techniques, the constraint model is inherently separated from the underlying solving techniques and algorithms employed for solving the constraints. By not having to change the solver, the constraints can be captured dynamically and provided to the fixed solver infrastructure.

The domain of dynamic scheduling is considered in this thesis and can be considered to be a part of onboard control software for embedded systems. Scheduling problems in

particular are suitable for employing CSP techniques since the start times that govern the execution of various tasks can be formulated as the output of several constraints. In its primitive form, the definition of schedule includes temporal constraints and resource constraints. Temporal precedence constraints provide the predecessor-successor relationship of the execution of tasks which resource constraints constraint the maximum utilization of the system resources. Many dynamic schedulers however are also required to take configuration constraints of the system into account.

Let us consider systems such as robotic space explorers or unmanned space missions. Such systems have fault tolerance built into them to account for permanent faults that could arise out of sudden impact or radiation when in space. Classical dynamic scheduling algorithms such as Earliest Deadline First (EDF), Rate Monotonic (RM) can be applied to fault-tolerant embedded systems where dynamic configuration changes to system could occur due to permanent or transient faults. However, it is complex to implement such schedulers in a distributed environment taking synchronization and configuration constraints into account. It results in an unpredictable scheduler [3].

Using CSP techniques for scheduling algorithms provides intelligent scheduling and enables the embedded system to be more adaptable to dynamic changes in the environment [2, 4]. For instance, a Ground Processing Scheduling System (GPSS) handles configuration constraints along with precedence and resource constraints in the constraint-based scheduler. The GPSS is an automatic scheduling system that can accommodate frequent configuration changes and output a schedule that also conforms to the temporal and resource restrictions in the system [3].

It is inherently difficult to encode such configuration constraints into a non-CSP based scheduler because its domain is not designed to handle such constraints. By modeling the scheduler as a CSP problem, the embedded system becomes flexible and adaptable to dynamic changes in the environment since it can accommodate a range of constraints apart from precedence and resource constraint. A CSP formulation of scheduling separates the constraint model of the problem from the solving techniques employed to solve the problem.

The constraint model of the problem consists of tasks, resources, and the various constraints that govern the execution of the tasks. The solving techniques are the algorithms used to solve the constraints. A change in the environment represented by the tasks in the system and events interacting with it can be represented by modifying the constraint model. The solving techniques and algorithms need not be modified due to a change in the constraint model. Due to this separation of concerns, a variety of problems can be solved.

This thesis discusses the development of a parallel finite-domain constraint solver in order to perform online constraint satisfaction for embedded systems. The features of this solver are that it is implemented in a platform with multiple soft-core processors with distributed memory architecture. The constraints for the application problem are captured from a Data Flow Graph (DFG) and solved on this platform. A tool is also developed that automates the partitioning of the given application and configuration of the underlying framework and support infrastructure for execution. The implementation followed is a different approach that is applicable to a different class of applications such as mission planning and GPSS [2, 3].

The discussion in this chapter is organized as follows. Section 1.1 provides background material in preparation for the chapters that follow. Section 1.2 introduces the research statement and provides motivation for this research. This is followed by a discussion on the contributions made by this research. Section 1.4 provides a review and summary of related work and experiments carried out in this research area.

1.1 Background

Constraint programming (CP) is a technique for finding solutions to problems by representing them as a set of constraints. CP has roots such as Artificial Intelligence (AI), Operations Research (OR) and Optimization techniques [5–7].

The mathematical model is represented in fig. 1.1 [2]. A problem in CP consists of decision variables v belonging to a domain D, a set of constraints c correlating these decision variables and an objective function h involving the decision variables whose value is to be maximized or minimized. The objective function is not required if it is not at optimization

problem.

The solution to a CP problem involves the assignment of domain-values to variables such that all constraints in c are satisfied. A constraint optimization problem is an extension of the constraint satisfaction problem, which also involves the satisfaction of a given objective function. When the variables involved in the constraints are restricted to the domain of non-negative integers the problem is referred to as a finite-domain constraint problem. A variable in the finite-domain can be represented in two forms:

- $v = [l, u]$, where l represents the lower-bound and u represents the upper-bound indicating the range of values contained in that set inclusive of the lower-bound and the upper-bound, or
- $v = a, b, c, d$, as a set enumerating all the individual values.

However, in this research we follow the first form of representation for reasons which are discussed later. Further explanation of the rationale behind this is discussed in sec. 1.1.3. Finite-Domain constraints have lot of practical applications and it is mentioned in Bartak [6] that 95% of all industrial constraint applications employ FD solvers to resolve problems efficiently in a timely manner. FD constraint solvers are finding increasing use in many problems related to scheduling (job-shop scheduling, time-table scheduling), planning, and resource allocation [5, 6].

1.1.1 Constraint Store

A primary component of a finite-domain constraint solver is the constraint store. The constraint store is a central repository containing all variables in the constraint satisfaction

Given n variables x_i with domain D_i,
m constraints c_j and objective function h,

find a solution	$< x_1, ..., x_n >=< v_1, ..., v_n >$	
with minimal	$h(v_1, ..., v_n)$	
subject to	$v_i \in D_i$	$i = 1, ..., n$
	$c_j(v_1, ...v_n)$	$j = 1, ..., m$

Fig. 1.1: Mathematical model of CP.

problem. Each variable in the constraint store is associated with a domain D. The domain imposes an upper and lower bound on the variable.

1.1.2 Propagation

A propagator is a concurrent computational agent that implements a constraint [5, 7]. Propagators retrieve variable state from the constraint store, perform constraint computation and update the constraint store on completion. Figure 1.2 [7] shows the relation between constraint store and propagators. Propagation refers to the process of domain reduction of the variables involved in a constraint.

Figure 1.3 provides a brief constraint satisfaction problem. We employ a constraint graph to model the CSP. The nodes in a constraint graph represent the variables, while the edges model the constraints connecting dependant variables. Initially, the constraint store contains the variables $B1, x \in [0, 10]$ and $B2, y \in [0, 10]$. The two variables are connected by the constraint $C, x < y$ which imposes through propagation the new bound on the variables: $x \in [0, 9]$ and $y \in [1, 10]$.

The primary goal of propagation is reduction of cardinality of each variable's domain. A propagation step will always lead to one of the following results:

- Removal of values from domain variable. For example applying $x < 5$ on $x \in [0, 10]$ results in $x \in [0, 4]$.

- Entailment: Propagator cannot add new information to the constraint store. For instance, applying the constraint $x < 5$ on $x \in [0, 4]$ qualifies for entailment because the domain of x already satisfies the given constraint.

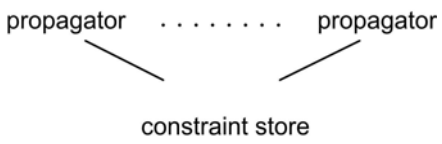

Fig. 1.2: Constraint store.

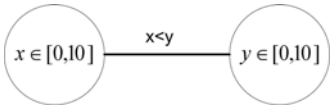

Fig. 1.3: Constraint graph.

- Failure: A constraint imposed in the propagator conflicts with information in the constraint store. Applying $x < 5$ on $x \in [6, 10]$ results in failure since no value of x simultaneously satisfies the domain criteria and the constraint.

A space refers to an information of a constraint satisfaction problem including propagators and the constraint store. The space can be any of the following four states:

- Failed: When a propagator fails due to an encountered propagation, the space is said to have failed.
- Stable: When no further information can be added to the constraint store through propagation, the space is said to be stable.
- Solved: When a solution has been found for the constraint problem, the resulting constraint store is said to represent a solved space.
- Undetermined: The propagators are actively pruning domains and the space is not yet failed or solved.

1.1.3 Pruning Methods - Domain vs. Interval

Interval pruning refers to the situation where propagation attempts to examine the upper and lower bound assigned to each variable. In contrast, domain pruning attempts to prune all inconsistent values from the domain. Domain pruning is computationally more expensive than interval pruning because it requires the examination of the entire list of values in the domain. Experimental studies by Henz et al. [5] and Fromherz [2] have shown that interval-pruning when combined with search yields solutions faster than domain-pruning which does not require any search.

1.1.4 Distribution

Interval propagation does not present the ability to completely determine a solution to general CSPs. When propagation leads to a a stable but unsolved space, the solver must take action to restart propagation.

Distribution consists of the creation of two complementary CSPs. The solver first creates two copies of the stable space. It then selects a single variable from the constraint store and binds it to a value from its domain. In the second copy, a complementary constraint is inserted. Each space is then separately searched for a solution. This is done through variable assignment followed by value assignment [2, 5, 6]. It involves choosing a variable from the list of FD variables present in the constraint store and posting a constraint on that variable. Choosing the variable depends on the variable assignment scheme followed where as choosing the constraint to post on that variable depends on the value assignment algorithm followed [7].

Propagation is triggered through the posting of new information to the constraint store in the created spaces. The process of propagation in conjunction with distribution is iteratively repeated until a solution is found. An unsolved constraint store is called a choice point.

Through distribution the solver creates a tree of choice points, solutions and failures [5, 7, 8]. The distribution step needs to be guided by a search algorithm. Solvers often provide several search algorithms that users can select. A systematic search involves the incremental exploration of the full search tree [2]. A heuristic search technique on the other hand explores only a portion of the tree. Search can involve finding an optimal solution based on a user-specified objective function.

When the search tree is being explored, choice points in the tree are visited in depth-first order. There is a need to revisit a choice point to explore other paths either when an unsatisfactory solution is encountered or when a distribution step fails. This is commonly known as backtracking.

Backtracking involves revisiting a node in the tree, which requires the restoration of

the state of the constraint store or undoing the operations performed on a state. There are various restoration or reconstruction techniques that are followed.

The commonly used reconstruction techniques are copying, trailing, and recomputation. In the current context, a node refers to a choice point in the search tree. Copying involves saving a copy of the entire tree node in memory before changing its state. In trailing, a record of the modified nodes only is saved before a state change. Recomputation does not have any past record of the state changes undergone. Therefore, reconstruction occurs by repeated computation by application of constraints starting from a previously saved state information. The number of computations depends on depth-interval between savings. There are many advantages and disadvantages to this technique. Copying and trailing require significant memory in order to save the constraint store. Recomputation is computationally intensive but does not require any memory for saving constraint states. A balance between performance and memory consumption can be reached by using recomputation in combination with trailing or copying. For instance, a fixed-recomputation technique, which is a combination of copying and recomputation, offers better performance than copying or recomputation executed as a standalone operation [9–11].

However, it must be noted that even though several studies exist which compare one method to another, the results are rather subjective and depend strongly on the nature of the constraint problem. Performance results depend on the size of the problem (number of variables involved, number of constraints) and formulation of the constraints (weak/strong propagation). A solution to a constraint problem consists of an assignment to all the variables in the constraint store which satisfies all the constraints.

1.2 Motivation

The goal of this thesis is to demonstrate that a flexible, scalable finite-domain constraint solver can be implemented in parallel targeting COTs FPGA for online constraint satisfaction applications in embedded systems. The motivation for this research is presented in this section.

1.2.1 Offline Constraint Solving

The theory and benefits of using finite-domain constraints is quite established and it has been used extensively in modeling and solving several problems related to scheduling, resource management, and optimizations [5, 6, 12]. However, a vast majority of current applications of FD solvers can be categorized as offline constraint solving mechanisms. This is because these decisions are not done in a reactive manner. The constraints and the decision variables involved in these constraints are derived using only design time information. With offline search response time is important but not mandated by the environment in which these solvers are used, and hence these systems are not reactive. Constraint solvers [13] employed in these systems are generic and are extremely demanding in memory, power, and processor utilization and cannot be executed on embedded systems. This design focuses on problem scalability rather than deterministic execution time.

1.2.2 Online Constraint Solving

Embedded systems are typically reactive systems which interact with their surroundings in real-time. A reactive system is one whose response and behavior changes in accordance to the external events which act as inputs to the system. Such systems react and provide output in time to meet the deadlines set forth by their environment. By modeling the events in a dynamically changing environment as a constraint satisfaction problem, the response of the systems adapts to previously unaccounted events that influences its behavior. This potential flexibility that is offered in modeling events as constraints encourages its use in embedded systems.

In Fromherz [2] the author demonstrates the benefits, flexibility, and viability that online constraint solving provides for computer systems embedded in printers. Another instance of online constraint solver is shown in Williams et al. [14] where constraint solving mechanisms are used in for autonomous decision making in space exploration. However, offline constraint solving techniques cannot be employed in embedded systems since they present significant challenges. Such techniques lead to a combinatorial explosion in the search space which increases the memory requirement of the system. Also, the increase

in search space leads to an exponential increase in the search time. There is a need for constraint solving techniques that can be employed in embedded systems for effectively executing online search problems.

Since constraint solving techniques can be computationally intensive process as stated by Fromherz [2] and Fernández [13], performance of a constraint solver in an embedded system can be potentially improved by parallel processing. FPGAs offer a unique mix of parallel processing capability and flexibility. They are increasingly being used as computational platforms for embedded systems. The primary contribution of this thesis is the development of an FPGA-based FD solver which can be used for online constraint solving problems.

1.2.3 Reasons for Choosing FPGA

An FPGA provides ample resources to execute applications in parallel. FPGAs provide a fabric primarily containing functional units such as Look-Up Tables (LUT), Multiplier units, Block RAM (BRAM) and routing and switching matrices for inter-connecting all the functional units within the digital fabric. Vendor provided tools used in configuring an FPGA to implement a given design translate the hardware design into a bit stream through the steps of synthesis and Place and Route (PAR). The input design for configuring the FPGA can be present in the form of a Hardware Description Language (HDL) or as a vendor library in the form of an IP core.

Design implementations on a FPGA offer several advantages. Firstly, there is more room for concurrent executions in hardware as opposed to a microprocessor. Secondly, it offers the ability to reconfigure the FPGA fabric as opposed to an ASIC whose logic is hardwired once the chip is taped out. In addition, FPGAs also provide the flexibility for rapid-prototyping and testing of the hardware design early during the design process. Due to the multiplicity of resources offered by the reconfigurable fabric, FPGAs qualify as natural candidates for parallel processing.

1.3 Contribution of this Research

The contribution of this thesis is the development of a flexible and scalable parallel architecture for online finite-domain constraint solver for an embedded platform. The results were obtained by successfully implementing propagation and online search on multiple soft-core processors in an FPGA. The implementation framework is generic such that additional form of constraints can also be supported by the system in future.

The capabilities of the online FD solver for online scheduling is shown through means of a potential online mission planning application. Mission planning consists of the allocation of a set of tasks to a fixed set of resources. Dependencies between tasks restrict the order of their execution which is implied by means of a directed graph. This dependency is modeled as a precedence constraint. Only precedence constraints were extracted from the application modeled as a task graph [12]. The schedule must be accomplished subject to the precedence constraint. As shown in the results, Chapter 6, the solver successfully handled precedence constraints and provided a solution. While resource constraints are also required in deriving an accurate schedule, it fell out of scope for this thesis and is definitely a path for future work for reasons discussed in Chapter 6.

1.4 Related Work - Literature Review

In this section, we examine related work and contributions in the area of finite-domain constraints. We categorize and summarize the contributions by other researchers in each sub-section that follows.

1.4.1 Comparison of FD Solvers

There are several constraint programming languages in existence. But the scope of the discussion is restricted to the features of ILOG Solver [15], ECL^{i}PS^e [16], Mozart-Oz [10] and JaCoP [17]. ILOG is a commercial tool while the others were born out of academic pursuits. The criteria used for comparison are their expressiveness, efficiency, and the different search and distribution strategies used by the language.

The implementation language used in these solvers differs from each other. Language of implementation is not a barrier for implementing constraint solvers or formulating a problem as a constraint [6]. Of course, the implementation language chosen limits the expressiveness of the constraint formulation. Mozart for instance, is an implementation engine based on the Oz language. ECL^{i}PS^e is a Constraint Logic Programming system that is implemented using the language Prolog. ILOG solver has is solving engine implemented in C++, and finally JaCoP provides a CLP environment that is implemented using Java.

In Fernández et al. [13], the authors evaluate the expressiveness and efficiency of several constraint languages of which ILOG, Oz and ECL^{i}PS^e are relevant to this discussion. Expressiveness of the language is defined as the closeness of the language representation of the constraint to the mathematical representation and efficiency of the solver is the time taken to reach a solution. In these measurements, only the times taken to reach the first solution were considered. The authors mention that the time taken to search all solutions was nearly the same for all solvers and was hence not the focus of comparison for efficiency comparison. The authors conclude that Oz is more expressive than ILOG and ECL^{i}PS^e because the solution in Oz is very close to the mathematical constraint formulation of the problem. The reasons stated were:

- The use of reified constraints allowed concatenation of propagators, and
- Propagators could directly operate over Boolean variables and constraint expressions.

ECL^{i}PS^e does not support reified constraints. ILOG solver on the other hand supports reification but the direct application of disjunction($\vee$), conjunction($\wedge$) and negation($\neg$) over Boolean variables is not allowed.

Performance benchmarks [13] were provided for several problems for the first-solution search using the first-fail labeling and the naive labeling. For the first-fail labeling for all the programs used for benchmarking (SRQ, magic square, n-Queens) ILOG was the fastest followed by Oz and ECL^{i}PS^e. On the magic square problem with 130 variables ILOG had a speed-up of 216 over ECL^{i}PS^e. Oz, on the other hand, had a speed-up that averaged around 3 over ECL^{i}PS^e for all the benchmarks mentioned in Fernández and Hill [13]. However,

this report is dated 1998 and many algorithms and libraries used in these solvers could have evolved since then. CPU and memory performance have improved drastically to this date. Therefore, the performance benchmarks quoted this paper may not hold true in the current scenario and would have to be revaluated.

The performance comparison for the first-solution using the naive labeling scheme in Fernández and Hill [13] revealed that ILOG was still the fastest followed by Oz and ECL^{i}PS^e. However, the variation in speed-up on the benchmarks in ILOG did not vary as much as in first-fail labeling.

This comparison on the performance between the first-fail labeling and naive variable enumeration method showed that the labeling method influences the performance of the solver. The results of performance measurements for first fail labeling varied among various solvers due to differences in the underlying implementation of the first-fail heuristic. The naive labeling approach does not provide any ambiguity in choosing the variable for value assignment because the ordered list of variables to perform assignment is always fixed. The left most variable in the enumerated list is always chosen and the lowest value of its domain is assigned.

In summary, the authors conclude that ILOG proved to be the fastest system and most robust with capability to solve constraint problems with more variables than the other two systems—Oz and ECL^{i}PS^e. Oz was faster than ECL^{i}PS^e for first and all solutions. ECL^{i}PS^e was the slowest.

Another insightful comparison on the performance of the solvers based on reconstruction techniques used is discussed in Schulte [9]. The reconstruction techniques of copying, trailing and recomputation are used in the exploration of the search tree by constraint solvers for finding a solution. It is shown that the performance offered by Oz in the flexibility offered in choosing between copying and/or fixed-recomputation techniques provides an advantage over ILOG [11] and ECL^{i}PS^e [9] which always follows a fixed trailing technique.

JaCoP has its applications [17,18] in the synthesis of embedded systems. The features of JaCoP [17] as mentioned by the authors are as follows. Applications of this solver are

discussed in Wolinski and Kuchcinski [18] and Kuchcinski and Wolinski [19], and are also discussed in sec. 1.4.2, Offline Constraint Solvers. No performance comparison of the JaCoP solver against other solvers exist at this point.

- It is a FD constraint programming system implemented in Java.
- It supports the use of reified constraints.
- The solver can search for all solutions or first-solution.
- Various systematic and heuristic search methods such as Limited Discrepancy Search, Credit search, hierarchical search and credit search are provided by the solver.

Many concepts used in the implementation of this project were borrowed from the Mozart-Oz system. As a tool Mozart also offers a visual explorer tree [8] which displays the search tree traversed when performing a distribution with the state of the constraint store during each step of the distribution.

1.4.2 Offline Applications of FD Constraints in Design of Embedded Systems

In this section we focus on the applications of finite-domain constraints as an offline constraint solver in the design of embedded systems. Discussions in Kuchcinski [12] and Wolinski and Kuchcinski [18] prove that several problems on scheduling, partitioning, and resource allocations related to the synthesis of embedded systems can be modeled and solved as a set of FD constraints.

In Kuchcinski [12] it is shown that optimal results in reduced time can be achieved over other solving techniques such as MILP and B&B for certain applications when finite-domain constraints are used to model the mapping and execution of tasks for a distributed embedded system. Efficiency of searching is also improved when search techniques are combined with the FD constraints. Through experimental results it is shown that FD solver provides a competitive advantage over (M)ILP solvers in the design of some applications. The model is flexible and comprehensive. It can be used to model task precedence constraints,

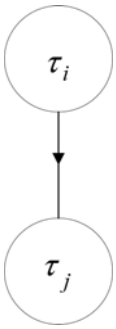

Fig. 1.4: Task graph.

resource constraints, power, and energy constraint, communication resources and memory constraints. This form of modeling is extensible.

FD constraints also offer an advantage in that they can be used along with any search technique for searching the tree of solutions. This technique provides more flexibility in the use of different techniques for finding an optimal solution. A combination of constraint solving techniques and optimization methods can be used together with finite-domain constraints to arrive at the solution. For instance, optimization methods such as Branch-and-Bound (B&B) algorithm, heuristic search methods can be used.

To introduce the concept and to demonstrate the efficacy of this methodology, a FD constraint model is developed and a solution representing the schedule is obtained for a task graph with a limited set of resources. Precedence and resource constraints are used in the formulation of the constraint model. A precedence constraint is a $\leq$ inequality which is used to indicate the order of execution of tasks in the task graph. It is also used to sequence the execution of communication. In that respect communicating events are also modeled as a task. A task is modeled as a tuple $< \tau, \delta, \rho >$ where τ is the start time, δ is the duration of execution and ρ is the resource occupied during execution. An example of precedence constraint is $\tau_i + \delta_i <= \tau_j$ for the task graph shown in fig. 1.4. The precedence constraint model mentioned in Kuchcinski [12] was leveraged in this research.

Resource constraints are modeled as multi-dimensional rectangles with each rectangular block occupying a resource for the given duration. A 2-dimensional rectangle is denoted by $R_i =< \tau_i, \rho_i, \delta_i, 1 >$. When the resource constraint is supplied with a list of such rectangles,

the solver spaces the rectangles such that no two rectangles overlap another. This implies no two tasks can simultaneously occupy the same resource.

Another contribution of Kuchcinski [12] is that varying execution times for tasks depending on the type of resource it is mapped to can also be modeled using FD constraints. Much of the principles presented in this paper are carried forward by the author to demonstrate a more practical synthesis problem in embedded system design.

In Wolinski and Kuchcinski [18], a hardware architecture for a pipelined schedule is derived using finite domain constraints. The uniqueness lies in having to use finite domain constraints to derive the pipelined schedule from a Hierarchical Conditional Dependency Graph (HCDG) [18] which also takes into consideration the inter-loop dependency.

1.4.3 Applications of FD Constraints for Online Constraint Solving

We now provide a summary of research work related to the implementation of FD constraint solvers as an online solver for embedded applications. The challenges encountered and design considerations that have to be taken into account are mentioned in this section of the survey.

In Fromherz [2], the author discusses the successful design of an online scheduler in a printer that reacts to events in a dynamically changing environment by providing a modified schedule. The printer is designed to be a smart printer with multiple user-configurable options for printing documents while it is online. For instance scanned documents can be printed in a double-sided manner or on a single side and for multiple copies of that document for various order of sorting of the printed pages. The user also has the options to throttle the pages per minute that are printed out. The response of the printer to these options should be such that it meets the objectives of the user.

The design of this printer is complex and involves controlling the operations performed by various mechanical and electrical components present in its system. The successful execution of a print job involves the execution of several sub-tasks that involve triggering the components in the printer in the right order at the right time. This in effect translates into a scheduling problem. The inputs from the user mark the various events that influence

the output of the scheduler. For instance throttling the page output of the printer involves controlling the speed at which papers are pulled into the print engine and flow through the "assembly line" for printing.

The main control software or scheduler for this printer is modeled using finite-domain constraints. In this printer, the occurrence of external events, the start time of the control instructions required to trigger the execution of the components, resources available in the printer are all as finite-domain constraints and the scheduling problem is solved using FD solving techniques. By using FD constraint constraint solving techniques to an online application such as printer, the behavior of the printer becomes adaptable to run-time changes to its configuration and is able to respond accordingly. The author justifies the feasibility of this approach due to its successful implementation in Xerox printers.

In Fromherz [2], the author also mentions that online constraint solving techniques for scheduling problems have been used onboard NASA's Remote Agent Planner [2] - a depth-first search constraint solver on-board the spacecraft Deep Space One and Scheduler (RAX-PS) and JPL's Aspen Planner and scheduler [2] - an online constraint solver which provided real-time response using iterative repair technique.

The scheduler implemented in the printer in Fromherz [2] is based on finite-domain constraints and is called a Constraint Based Scheduler (CBS) that reacts to changing constraints in real-time. The CBS model is formulated from the tasks, constraints, resources, and the objective function.

In offline scheduling, all tasks and constraints are known apriori and are posted to the constraint solver. Online scheduling requires a mechanism to incrementally add/modify tasks, constraints, and resources dynamically to the controller responsible for deriving the schedule with responsiveness of a real-time system in a changing environment. The system reacts to changes in the tasks/constraint to generate a modified schedule. The state of the system could change at run-time due to unavailability of resources from a fault, additional resources due to fault tolerance and other unpredictable events that impose restrictions on the behavior of the scheduler.

The design of a CBS discussed in Fromherz [2] was achieved by modeling the system in a hierarchical composition manner with a component containing sub-components. Each component announces its parameterized values that define the characteristics of its operation to its parent. Every parent can derive the larger constraint from the constraints of the parts from which it is composed. A custom constraint language known as CDL is used for this purpose. Based on the composition constraints, the scheduler in the system controller derives the schedule which states the timed execution of individual control commands to control the sub-components.

The strength of this approach is the scheduler that dynamically reacts to the changing environment by modifying and updating the constraints that define the execution of the tasks on a set of resources generating an optimal schedule.

1.4.4 Distributed Constraint Solving - Parallel Search Technique

In this section, we discuss an implementation of a distributed search on a network of computers and also draw similarities with the implementation adopted in this research. These discussions are intended to show that concurrent search provides benefits in performance.

In Jaffar et al. [20], the authors discuss an implementation of a parallel combinatorial depth-first search on a distributed network of computers. Each computer forms a node in the cluster of computers. The branching algorithm followed is the first-fail heuristic where two sub-spaces are created by imposing complementary constraints.

The cluster for performing a distributed search is of the shape of a binary tree and new nodes can join or leave the cluster at run time. The search tree forks from a single master at the root which controls the assignment of the various search problems and also records the exploration of various solutions. The worker node is assigned a subproblem of the entire search tree. Each worker node that is assigned a search subproblem can in turn fork more search problems to its child nodes. The solutions found by each worker is reported to the master which based on the type of search exploration decides to explore more solutions or not.

The authors demonstrate in their results that a linear speed-up can be achieved in their architecture through a plot of speed-up vs. number of workers for different sized problems (number of variables in the constraint problem) for both backtracking and branch and bound search algorithms. Advantages of this architecture are that it is a scalable distributed architecture and dynamic load balancing is achieved by maintaining a balanced binary tree.

In Jaffar et al. [20], a worker node is assigned the entire constraint store representing that search subproblem, whereas in this thesis the master and each worker node solves a dedicated portion of the constraint store and communicates their results to the master at the root. Second, worker nodes can participate or leave the distributed computing network anytime during run-time while in this thesis the tree infrastructure and the number of processors are fixed.

1.4.5 Design Overview

In order to gain a broader understanding of the designed system used to solve the given problem we refer to fig. 1.5. The different functional layers implemented on the FPGA platform are described with reference to this figure.

The lower most layer is the hardware platform on which the multi-processor interconnection network is implemented. Above the hardware layer is the software or firmware layer that controls and drives the underlying hardware infrastructure. The firmware and the hardware layer together form the communication infrastructure for the platform. This is explained in detail in Chapter 2.

The layer indicating constraint solver contains algorithms that implement propagation, distribution and search. Chapter 3 discusses the implementation of a propagator while Chapter 4 explains the distribution and search steps. As denoted by the top-most layer, the constraints are decoupled from knowledge of the underlying platform and algorithm for implementing the constraints. In that respect, the application interface is design to suit this abstraction.

The tool flow for automated decision making was also implemented in this research.

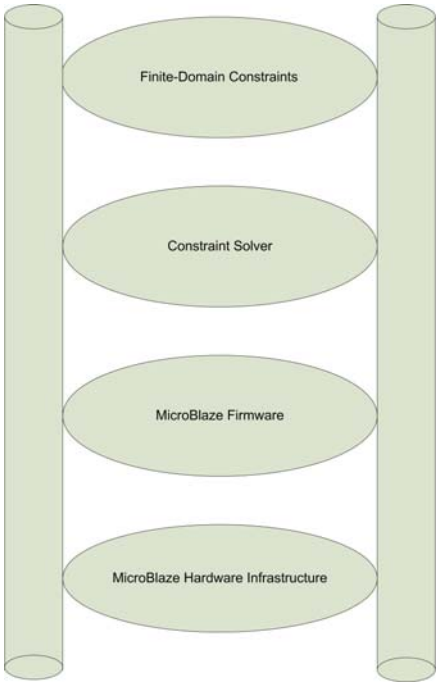

Fig. 1.5: Functional layers of the computing unit.

This is illustrated in fig. 1.6. The input to the tool is a Data Flow Graph (DFG) representation of the given application problem. The tool parses this graph and partitions the nodes contained in DFG into clusters. For each cluster that is formed a set of finite-domain constraints contained in those clusters are generated. Also, firmware configuration for the hardware platform is generated. The configuration information assumes the existence of a valid hardware design. The partitioning of the graph is discussed in Chapter 5. Finally, an evaluation of the design is performed for sample applications and its results are discussed in Chapter 6.

This thesis proceeds as follows. In Chapter 2 we discuss the MicroBlaze infrastructure that is used for the implementation of a distributed memory architecture. Chapter 3 discusses the formulation of the constraint model and the implementation of a propagator. This is followed by a description of the distribution and search mechanism implemented specifically for a multi-processor network. Chapter 5 explains the clustering algorithm followed in this project. Finally, we conclude by explaining the results and discussing the

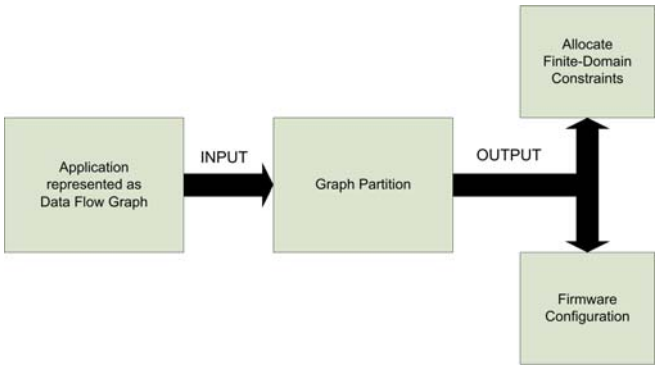

Fig. 1.6: Tool flow.

scope for future developments in this research.

Chapter 2

Communication Infrastructure

The architectural foundation on which the FD solver is based is composed of a network of parallel processors with fast point-to-point communication links. This chapters discusses the substrate targeting a Field Programmable Gate Array (FPGA). The hardware platform used for testing was a Xilinx Virtex-II Pro FPGA on the ML310 development board from Xilinx.

2.1 System Description

Let us consider a model of constraint store with a set of concurrent propagators operating upon it as shown in fig. 2.1. This model implies the need for a globally shared memory accessible by multiple computational units. Only through sharing information about constraint variables are propagators able to jointly make progress toward solving a given constraint satisfaction problem.

From a practical implementation viewpoint, such a globally shared memory accessible by several computational devices is not practical due to limitations on the number of read and write ports available to a memory device. Further, the selected implementation platform, an FPGA, offers multiple, small internal memory banks, each of which is separately accessible. It offers no native global memory structures. Therefore the concept of shared memory was emulated, making use of the native distributed memory structures, as illustrated in fig. 2.2. When compared with fig. 2.1 it can be said that memory associated with each processor represents a partition of a single large globally shared memory. The partitioning and distribution of the constraint store results in the allocation of finite domain variables to a distributed set of processors, each with their own local memory. This effectively divides the constraint store into a set of partitioned constraint stores. Access to

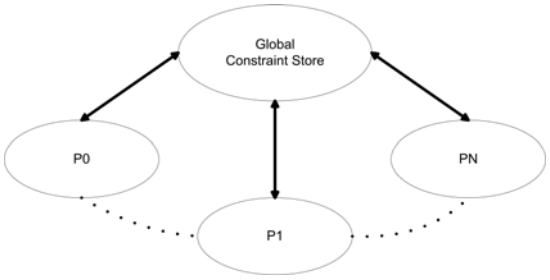

Fig. 2.1: Propagators accessing a global constraint store.

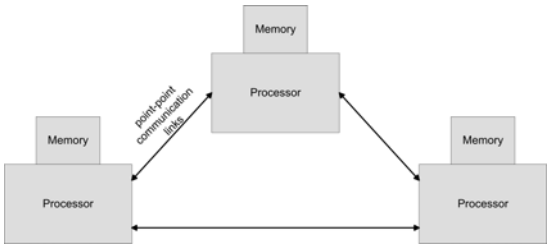

Fig. 2.2: Distributed memory architecture.

non-locally held finite domain variables is implemented via point-to-point communication links.

The major considerations in the design of this communication infrastructure were:

1. To provide provision for parallel processing;
2. Providing a pseudo-implementation of a shared global memory through distributed memory architecture and thereby increasing overall memory bandwidth;
3. Reduce communication overhead;
4. Facilitate global communication and control.

We provide a brief introduction of the hardware concepts in fig. 2.3. A processor with its local memory is referred to as a node. Each node represents an instantiation of a soft-core microprocessor provided by Xilinx MicroBlaze. Memory was allocated from on-board block RAMs, for both instructions and data. Peripherals were configured and added to each MB for supporting inter-node communication. FSL links were used in the hardware communication infrastructure.

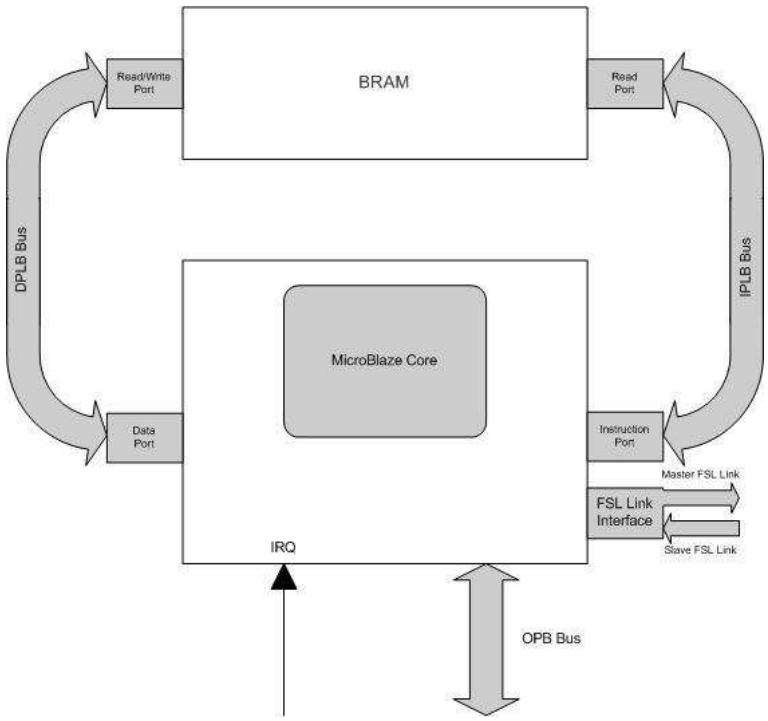

Fig. 2.3: A single MicroBlaze and its interfaces.

A Microblaze (MB) is a 32 bit single-issue pipelined RISC processor IP core which can be instantiated in an FPGA design. Such processors are also referred to as soft-core processors. It employs a Harvard architecture which implies it contains a separate instruction and data address space. It can be clocked at a maximum frequency of 100 MHz on current FPGA platforms.

2.2 Hardware Peripherals

A simplified representation of the MB and its peripherals is provided in fig. 2.3. A MicroBlaze with fixed on-chip memory in the form of Block RAMs (BRAM) is used for each processing node in the design. The memory size of the allocated BRAM is fixed during design time. In order to achieve a distributed memory architecture the size of BRAM allocated to each MB is limited. Each BRAM in a processor node is configured to contain a single read-port and a read/write port during design time. The read-port is used for instruction fetch while the read/write port is used for data reads and data

writes. These ports are interfaced to the MB through the Instruction Processor Local Bus (IPLB) and the Data Processor Local Bus (DPLB). The MB can communicate with external peripherals or other MBs through Fast Simplex Links (FSL). An FSL establishes point-point communication link between a MB and another peripheral or MB. Each processor in the network is also associated with an Interrupt Controller in order to support asynchronous message passing.

2.2.1 Memory Configuration

As shown in fig. 2.2, each processor has access to a limited amount of BRAM memory which is local only to that processor. In order to reduce memory access latency, the memory configuration adopted for this architecture was a flat on-chip only approach, where all memory accesses were performed using the processor's local bus. The local memory used for each processor was allocated from the pool of on-chip memories present in the FPGA known as Block RAM (BRAM). These on-chip memory modules have fast access times and reduce latency from off-chip memory access. The size of the application program and the problem definition for the purpose of this research was restricted such that they could be executed within the limited amount of local memory. The discussion of the impact of this limitation is included in the Chapter 6.

Unlike a shared-memory architecture which offers a single, global memory address space accessible to all processors, in a distributed-memory architecture each processor only has direct access to its local memory.

2.2.2 Communication Links

The point-to-point communication links shown in fig. 2.2 are implemented using FSL links. Since a single FSL offers only unidirectional communication, two FSL links are required to achieve full-duplex communication. The number of FSL link interfaces that can exist in a MB can be configured for each MB during system design. A MB can support at most eight FSL links. A MB provides two FSL bus interfaces for every FSL interface that is activated—a Master FSL bus interface and a Slave FSL bus interface. Data can only

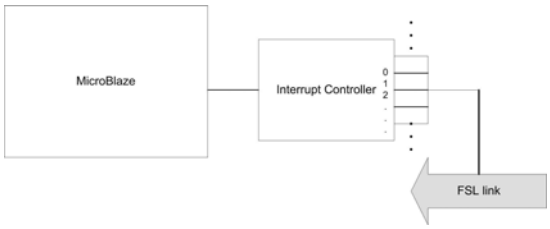

Fig. 2.4: FSL-interrupt controller interface.

be sent on the Master bus and only received on the Slave bus interface. The number of FSL bus interfaces provided in the MB is dependant on the number of MBs with which it directly communicates.

An FSL is implemented as a FIFO whose depth can be configured during design time. In this implementation the data width for all FSL links was configured to be 32 bits wide. Data is inserted or written to the FSL by the source processor, synchronously and read (or dequeued) asynchronously by the destination processor. The asynchronous reads at the destination processor are achieved through interrupts where an Interrupt Service Routine (ISR) handles the incoming read requests.

The FSL link is configured to interrupt the MB. The corresponding MB ISR reads the received data from the FSL buffer and dispatches it appropriately. Software execution on the MB in this implementation need not explicitly wait for the receipt of data on an FSL link.

2.2.3 Interrupt Controller

As shown in fig. 2.4, each MB is associated with an interrupt controller whose lines are tied to the control lines of the input FSL links. An MB can have multiple FSL links each with the ability to interrupt the processor. An interrupt controller is required to multiplex multiple interrupt requests onto a single interrupt request line of the MB. Interrupt requests are serviced based on assigned priority. When data is present in the FSL link, the FSL asserts a control line. The control line is connected at design time to an interrupt line on the incident MB. This assertion leads to the triggering of the ISR on the MB. Pending

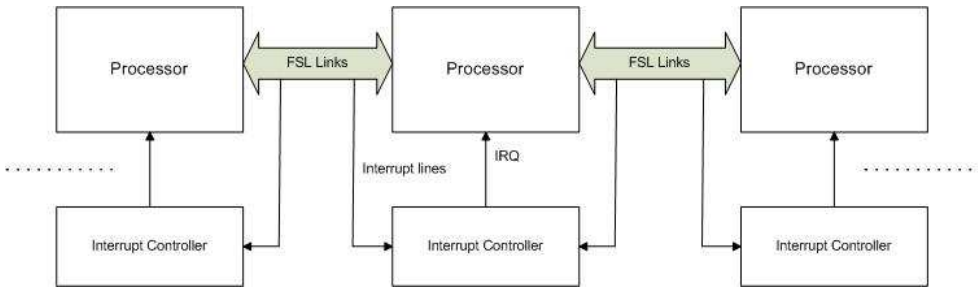

Fig. 2.5: System overview.

interrupt requests in an interrupt controller are serviced based on the priority assigned to each interrupt line on the interrupt controller. The priority is essentially a fixed, unique integral number assignment starting from 0 marking each interrupt line. Lower-numbered interrupt lines have higher priority than higher-numbered lines. The interrupt controller acts as a dispatcher when multiple interrupt requests are pending. Since only one interrupt can be serviced by the processor, assigning priorities decides the order in which pending interrupts are to be serviced.

2.2.4 Processor Inter-Connection Network

In order to facilitate global synchronization and control of processors a subset of the inter-node FSL links are selected to form a minimum spanning tree over the network. This tree is used to broadcast "system-level" messages and global synchronization words across each node in the network. A detailed discussion of this broadcast tree infrastructure is provided in Chapter 4.

Figure 2.5 depicts the complete processing framework, illustrating processors, interrupt controller, and FSLs.

2.3 Data Partitioning

To achieve parallel processing in a distributed-memory architecture, the data set to be processed for computation must be partitioned and distributed among individual processors in the network. The goal of distributed computing is to achieve faster processing of

the data set through parallel execution than what is otherwise observed on a single computational unit. However, correct parallel processing almost always requires inter-processor synchronization. Synchronization often implies the exchange of partial data sets between neighboring processors.

A parallel FD solver is not immune to synchronization. Distributed processing requires node to share information on the current state of the shared FD variables. Data partitioning distributes FD variables among processors, but the architecture must support rapid synchronization and update of FD variables shared across processor boundaries. In this implementation, each FD variable is assigned to a particular partition. Each MB is allocated a single partition. Inter MB sharing requires a well-defined approach to memory addressing in order to allow nodes of different address spaces to access a saved variable.

2.3.1 Partitioning of Address Space

Data partitioning restricts the range of natively addressable memory locations for a processor. This data partitioning inherently causes a partitioning of the global address space. The partitioned data is called a cluster. An addressing scheme is required in order to deterministically identify data dependant variables that are allocated to various clusters.

A uniform addressing scheme is applied for addressing every variable in the system. Each variable in any cluster can be identified by its unique global address. The variables within a cluster are assigned a contiguous sequence of global addresses with no intermittent gaps. Unused address space can only exist at the end of all assignments within a cluster. Further, the range of the addresses allocated to each cluster is equal across clusters and is rounded to the nearest higher power of two.

Restricting the range of address assigned to clusters to be a power of 2 helps in deterministically mapping any valid global address to a unique cluster. This address translation scheme used in mapping a given global address to a variable in a cluster is explained in sec. 2.3.2 with the help of an example.

2.3.2 Address Translations

The various steps involved in the accessing of variable with a given global address are explained by means of a pseudo code as shown in fig. 2.6 with related computations involved in this process. Most of the algorithm is self-explanatory. This algorithm is explained with the help of an example. For the partitioning shown in fig. 2.7, the following address translations apply based on algorithm shown in fig. 2.6.

- $ClSize = 8; ClBits = 3$
- Let global address $GlAddr = 10$
- Substituting in Step(1) we get, $ClID = \frac{10}{8}$

This achieves the mapping of a global address to a cluster. At compile time the size of the cluster and the number of nodes in each cluster remains constant during run-time and is determined by the clustering algorithm. A node distinguishes between access to locally owned or variables owned elsewhere via lookup table.

- Local access: When a given global address is determined to exist natively within a cluster, then the variable is accessed though a simple global to local address translation and indexing into a local table as described in steps (4)-(6) in fig. 2.6.
- Remote access: When it is determined from the global address that a variable does not natively reside in the requesting processor, then the access of that variable is classified as a remote access. Since it requires inter-processor communication, a remote access incurs more processing overhead and access latency, than a local variable. A message routing table is generated for each MB at design time and included in the runtime software.

  An access request message is sent over the FSL link instructing the destination processor to access this remote variable (from the perspective of the source processor).

  Sections 2.4 and 2.5 refine the message based inter-processor communication.

Given:

Number of clusters in the system $\rightarrow N_C$
Uniform cluster size(power of 2) $\rightarrow ClSize$
# of bits required to represent cluster size $\rightarrow ClBits$
$\Rightarrow ClBits = log2(ClSize)$
Current cluster identifier: $CurrID$
Starting Global address of current cluster $\rightarrow StartGlbl$
Table of FD variable $\rightarrow fd\_table_{1..ClSize}$
Table which maps cluster ID to a processor: $clusterProcMap_{1..N_C}$
Table which maps processor ID to an FSL: $procFSLMap_{1..N_C}$

Input:

Global Address $\rightarrow GlAddr$

Output:

Type of access required:Local/Remote access

Main Algorithm:

(1)Cluster identifier $ClID = GlAddr/ClSize$
$\Rightarrow ClID =$ right bit-shift $GlAddr$ by $ClBits$ positions
(2)**if** $ClID = CurrID$ then
perform **Local Access** of $GlAddr$
(3)**else**
perform **Remote Access** of $GlAddr$

Local Access:

(4)Local address $\rightarrow LA$
(5)Global to local address translation: $LA = GlAddr - StartGlbl$
(6)Access local variable from FD table: fd_table_{LA}

Remote Access:

(7)Cluster to processor mapping:
Processor ID $\rightarrow procID$
$procID = clusterProcMap_{ClID}$
(8)Processor to FSL mapping:
FSL ID $\rightarrow fslID$
$fslID = procFSLMap_{procID}$
Communicate over FSL link $fslID$

Fig. 2.6: Algorithm to map a global address.

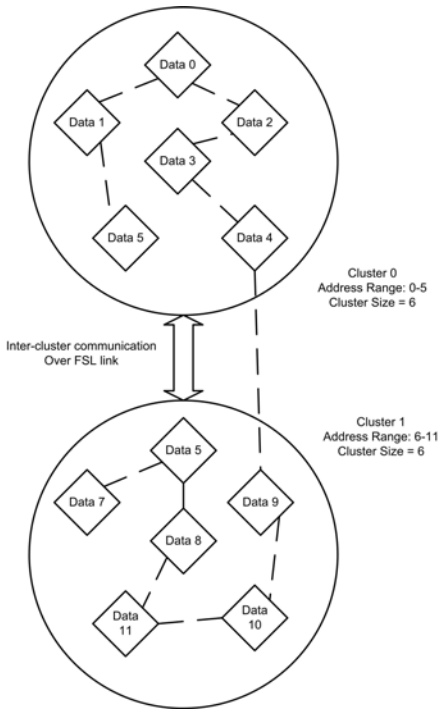

Fig. 2.7: Initial data partitioning.

- Mapping: On a remote access, the proper FSL link to access is determined by indexing the routing table based on global address. Since the global address space is small, the routing table is manageable. This is routing occurs in two steps. First, the mapping of a given global address to a processor is performed using a lookup table to identify the processor on which that address resides. The processor index is then used to index into another lookup table to determine the FSL link to use. In summary there is a:
  - A cluster to processor mapping,
  - A processor to FSL mapping.

2.4 Shadow Copy

One approach to performing remote data access (read/write) that was abandoned followed a persistent data read request mechanism. A message requesting a read of the remote data is sent over the FSL link to the destination processor. The source processor blocks on

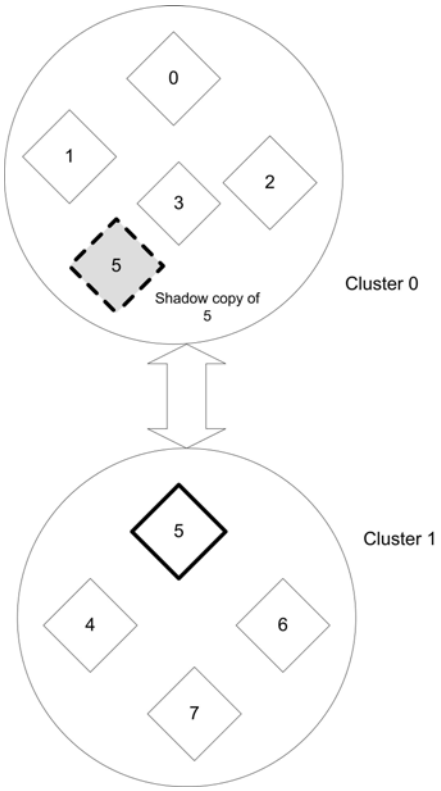

Fig. 2.8: Clustering with shadow node.

this read request and awaits a response on the FSL link from the remote processor. The following observations were made due to which this approach was not feasible.

1. Frequent remote read requests when there is no new information to convey on the shared data due to the iterative nature of propagation. But relatively infrequent remote write accesses were observed.

2. IPC overhead increases due to two message exchanges for a single read request. This decreases processor utilization for useful computation.

Hence to reduce FSL utilization, local cached copies of remotely stored variables are maintained on each node. We refer to these local copies as shadow copies. In fig. 2.8, when cluster 0 attempts to access data with address 5, it is determined after address translation that this data resides in cluster 1. Instead of blocking on a read request sent over the FSL

link to cluster 1, the content of variable 5 is read from its duplicate copy residing in cluster 0. This duplicate variable is referred to as a *Shadow Copy*. Similar access occurs during a write to a remote data.

The advantage of having a shadow copy is that remote data accesses of a node are now done locally, reducing the latency of access of the node. This reduces the frequency of interrupts originating from the FSL, both on the parent cluster owning the node and the surrogate-parent containing shadow copies of the node. This mechanism most importantly reduces overhead incurred due to frequent servicing of interrupts and improves CPU utilization for the main thread of computation.

Maintaining shadow copies implies that data coherency issues need to be addressed. When the cluster hosting a shadow copy of a remote data modifies the value of the shadowed variable, the original content of the data present in the host cluster must also updated. This update is performed via remote access messaging. Similarly when the host-parent updates a value which is shadowed on remote processors, a message to update the corresponding shadow copies present on other nodes is broadcast. In this manner, the contents of both the original and shadow copies remain in sync with each other.

Providing shadow copies decouples two MBs. This is because each MB independently carries out its computations without being blocked to receive the original value of the remote data. Synchronization between shadow data and real data occurs asynchronously. This delay in syncing does not impact the correctness of the collaborative computing achieved by means of propagation because the application is tolerant to such delays. This is explained in Chapter 3 where we discuss propagation in more detail. However, as will be discussed, inter-node synchronization can impact time to convergence.

The shadow variables residing within a node occupy a separate address space than the real variables and are indexed using a different lookup table than the one followed for accessing real variables. In fig. 2.6 the procedure to access the real variable based on a given global address was explained. The algorithm in fig. 2.9 explains that a given global address is translated into an index into a table distinctly maintained for shadow variables.

Begin:
Global address $\rightarrow GlAddr$
Shadow index $\rightarrow SI$
$SI = MapGlobalShadow(GlAddr)$
Access shadow variable:
$shadow\_table_{SI}$

Fig. 2.9: Algorithm to access shadow variables.

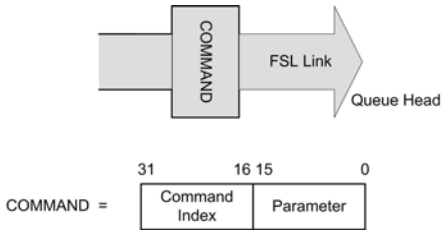

Fig. 2.10: Generic command encapsulation over FSL link.

2.5 Message Passing Interface

We now discuss the message passing interface used for inter-processor communication. The above discussions have loosely revolved around the exchange of data between processors using messages. These messages represent commands that are sent over the FSL links. Figure 2.10 illustrates the format of a command. The command consists of two 16 bit half-words—the command index and another parameter associated with this command. The parameter field may be left blank depending on the command. The command index is used to index into a table of instructions as shown in fig. 2.11 which contains a list of function pointers. The acceptable commands are fixed at compile time and is uniform for all processors.

The command index represents an instruction to be performed at the destination node. To speed the instruction dispatch, a table is constructed at design time containing function pointers, pointing to the appropriate instruction handling code. On receipt of a command, the index is used to look up the appropriate pointer from the table, which is then indexed to handle the command. All MBs employ a table of instructions as shown in fig. 2.11.

Inter-processor communication occurs when the source processor sends one of the commands along with additional parameters if required over the FSL link.

This table based scheme is generic enough such that modifications to the command interpretation and ability to support more commands exist. A message based communication was extensively used for implementing the steps of propagation, distribution and search to achieve synchronization and control.

2.6 Remote Update Command

A remote update using the message passing interface could involve either the real FD variable or a shadow FD variable. The encoding of the write-command over the FSL link is illustrated in fig. 2.12. The write-command is encoded by a fixed index that is used to index into the command handler table (fig. 2.11). It also requires additional parameters to process the command such as the global address of the variable within the destination cluster and the value of the FD variable which is represented by a lower bound and an upper bound. This generic write-command interface can be used to write real variables as well as shadow variables by using different command indices to encode the command.

The accomplishments of this design may be summarized as follows. Firstly, it emulates distributed shared memory architecture by allowing a MB to access a set of memory locations from adjacent MicroBlaze. This facilitates distributed computing to be performed which is the central idea behind the realization of a parallel architecture. A data partitioning and addressing scheme is developed for this purpose. Accessing critical sections

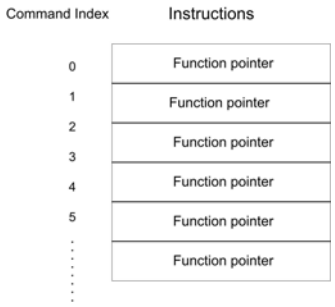

Fig. 2.11: Generic command table.

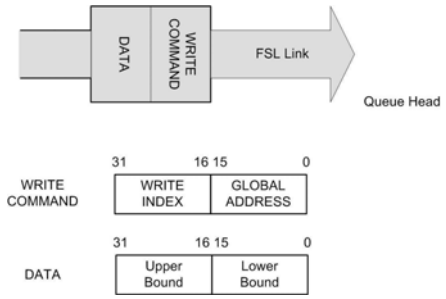

Fig. 2.12: Remote write request format.

and achieving data synchronization are the challenges posed by such architectures and have been addressed in Chapter 3.

Secondly, a message passing interface is developed for inter-processor communication. These messages encompass instructions and data exchanged between processors and can trigger a change in the execution flow of the processor.

Finally, the frequency of inter-processor communication is reduced by introducing shadow copies which reduce the utilization of the FSL links.

Chapter 3

Implementation of Propagator

Online planning and scheduling algorithms can take many forms, from determining the optimal resource mix for a computation graph, to determining a binding of tasks to computational elements in a resource constrained environment. The re-mapping tasks to parallel resources due to device failure could also be considered a candidate for online search.

3.1 Modeling the Problem

Finite-domain constraints are employed in online constraint solving for scheduling applications in an embedded system. As mentioned in sec. 1.2, the primary motivation for using online constraint solvers was to schedule events during run-time in a dynamically changing environment. When the problem of scheduling is modeled as a constraint satisfaction problem it is seen that the scheduler is more flexible and adaptable to events, tasks, resources and other constraints that were previously unaccounted for before deploying the scheduler. In a sense, the behavior of the embedded scheduler is redefined with modified parameters without bringing down the system it controls. For instance the scheduler in an embedded mission planning for space applications has to deal with unanticipated failure of devices, temporary unavailability of a component and redundancy in components while the system is online. When such events are captured as constraints and presented to the online finite-domain constraint solver, it is seen that behavior of the system is easily extensible and flexible.

The application to be scheduled is represented in the form of a Data Flow Graph (DFG). A DFG is graphically represented as a directed graph containing nodes and edges. The nodes in the DFG represent an operation being executed on a resource and edges connecting the nodes denote flow of data and set the precedence of operations in the graph.

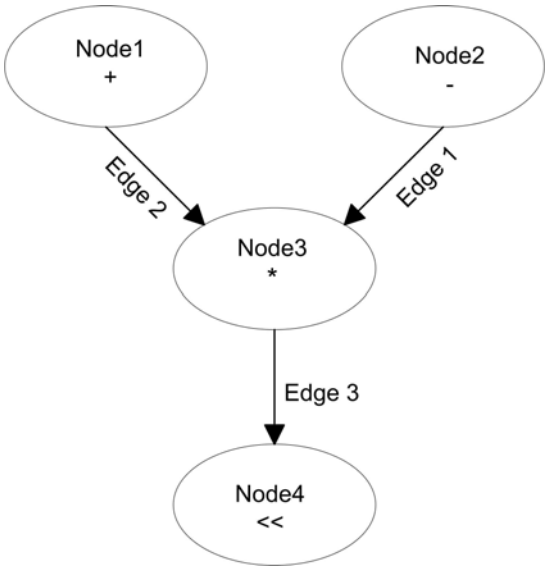

Fig. 3.1: Sample data flow graph.

The formulation of the finite-domain model were leveraged from the work described in Kuchcinski [12], where an FD constraint model is deduced from task graphs. The finite domain constraint representation of a dataflow graph is identical to that of a task flow graph, when considering precedence and latency.

A sample DFG used for constructing the FD model is illustrated in fig. 3.1. By definition, a schedule is the process of assigning valid start times to tasks for occupying a limited set of resources [2]. The DFG that is considered in this research has no notion of resource availability where as scheduling assume resource availability. However, in order to prove the feasibility of parallel online constraint solving, the scope of this research is only limited to precedence relationship set forth by edges of the DFG and resource occupancy is not currently considered.

The nodes in the DFG shown in fig. 3.1 are labeled and are associated with an operation. Edges in the graph represent a communication path for information flow from the source to the destination node. The DFG model specifies that every node produces output data only at the end of its execution and a node can start executing only after the availability of input data it requires. Every node executes for a fixed duration, the task latency after it fires. An

input edge of a node in a DFG represents data inputs upon which the node depends. For instance, $Node3$ must wait for the arrival of data on both its input edges in order to start execution. Valid sequences of execution for this graph are {$Node1$,$Node2$,$Node3$,$Node4$} or {$Node2$,$Node1$,$Node3$,$Node4$}.

3.2 Mathematical Formulation of the Constraints

For the formulation of the finite-domain constraint model for the DFG, we leverage the concept of precedence constraints of task graphs, as mentioned in Kuchcinski [12]. Precedence constraints specify the requirement that data dependencies must be satisfied in the schedule.

Let us consider a task T, represented by the tuple $< \tau, \delta >$, where τ represents the start time of execution of a task and δ represents the latency or duration of execution of a task. The precedence constraint for $Edge2$ in the DFG shown in fig. 3.1 is represented by an inequality:

$$\tau_{Node1} + \delta_{Node1} \leq \tau_{Node3}. \tag{3.1}$$

Often DFG scheduling is associated with some fixed deadline which specifies that the generated schedule take a total of X cycles or fewer; this limit is referred to as the makespan. The constraint imposed by the makespan implies that the execution of the last node in the given graph must be complete before the deadline imposed by the design. To satisfy this constraint it would be sufficient if just the sink node in the graph complete its execution before the specified deadline. This constraint can also be represented as a <= precedence constraint $[\tau_D, 0]$ with zero latency:

$$\tau_{Node4} + \delta_{Node4} \leq \tau_D. \tag{3.2}$$

In general, both precedence and latency constraints equations may be represented in the form:

$$A + B \leq C, \tag{3.3}$$

where $A \equiv \tau_{source\ node}$, $B \equiv \delta_{source\ node}$ and $C \equiv \tau_{dest\ node}$.

3.3 Constraint Solving - Propagation

As mentioned in Chapter 1, FD constraint solving consists of propagation, distribution, and search. The implementation of a propagator is discussed in this section. To demonstrate the process of propagation, the FD constraints that were deduced from the edges in the DFG in sec. 3.2 are used as a reference example. Constraint solving only requires the set of constraints to operate upon and does not have any knowledge of the application.

3.3.1 Propagation Principle

A propagator is a concurrent computational agent which implements a particular constraint [8] and is used to prune the domain of variables held in the constraint store. A propagator embodying the constraint specified in eq. 3.3 is implemented in this research. Each parameter in the propagator is a finite-domain variable. A FD variable is represented by a pair [lb,ub], where lb represents the lower bound of the FD variable and ub the upper bound of the variable, lb, ub are non-negative integers. The domain of the FD variable is restricted to the interval bound by lb and ub.

3.3.2 Interval Pruning

The effects of propagation of the inequality given by eq. 3.3 is explained with an example. Intially A, B and C have the finite-domain values $(0, 10)$, $(4, 4)$ and $(2, 12)$ assigned to them, respectively. Substituting the values in eq. 3.3, the constraint reads

$$(0, 10) + (4, 4) <= (2, 12).$$

The inputs to this constraint are feasible for propagation and domain pruning leads to following set of values being assigned to the variables A, B and C, respectively: $A = (0, 8)$, $B = (4, 4)$ and $C = (4, 12)$.

Observe in this constraint that the value of the constant variable B has not been

modified after the propagation of this constraint. However, the values of A and C have been modified to satisfy the constraint.

Propagation is now performed for the precedence constraints in eq. 3.1-3.2 which were formulated from the edges in DFG given by fig. 3.1. Propagation results in the following lower and upper bound assignments:

$$Node1 = [0, 2],$$

$$Node2 = [0, 3],$$

$$Node3 = [5, 7],$$

$$Node4 = [8, 10].$$

3.3.3 Reduction in Number of Propagation Steps

The convergence of the propagation towards a fixed point for the set of constraints eq. 3.1-3.2 is illustrated in figs. 3.3-3.6. Propagation for each edge in the DFG is performed in the direction in which graph traversal occurs and in the sequence in which each edge in the DFG is encountered.

From these figures, it may be observed that the direction of traversal for propagation affects the number of iterations required to converge to a solution. A single iteration consists of a single pass of the graph traversing it in the same direction either from the root to the leaf nodes or vice versa. A single pass of the graph could either be a traversal from the root node to the leaf node in which case it is termed as a forward pass or from the leaf nodes to the root nodes in which case it is termed as a reverse pass. When propagation is performed in the forward direction, it is observed that the number of iterations for propagation to proceed to completion is equal to the number of levels in the graph. For the DFG illustrated in fig. 3.2 the number of levels is 3. Hence, it takes three forward passes before propagation stands completed. The pruning that occurs during each pass is also illustrated in figs. 3.3-3.5. On the other hand, a forward pass followed by a reverse pass takes only two iterations for propagation to converge to a solution. This was observed irrespective of the depth of the input graph traversed. The reduced number of iterations for the forward-reverse case can

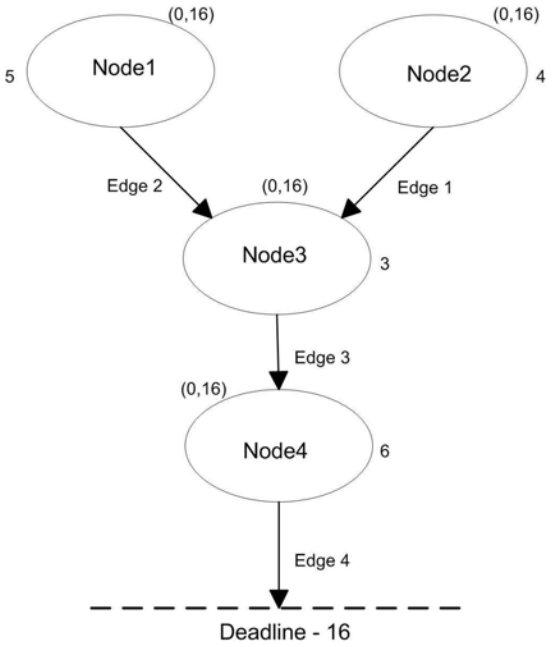

Fig. 3.2: Initial node assignments.

be explained as follows.

At the end of the initial forward pass, the leaf node $Node4$ converges to completion since it is the parent node of the constant $Deadline$, which is a finite-domain variable that is constant. Subsequent forward passes will not prune the value of $Node4$ since it has converged to a solution. In fact, during every subsequent forward pass, the non-converged finite-domain variable which is the immediate parent of a converged finite-domain variable converges to completion. It takes three passes equal to the number of levels in the DFG for all nodes represented by finite-domain variables to propagate to completion. When performing a reverse pass after a forward pass, every immediate parent-node of a completely converged node also propagates to completion during every step of the reverse pass. At the end of the reverse pass the propagation of all nodes in the DFG proceed to completion. This is applicable for any directed acyclic graph.

It remains to be seen as to how a single propagator representing the constraint $A+B <= C$ was actually implemented. The algorithm used for pruning the finite-domain variables presented to this propagator is explained and illustrated in sec. 3.4.

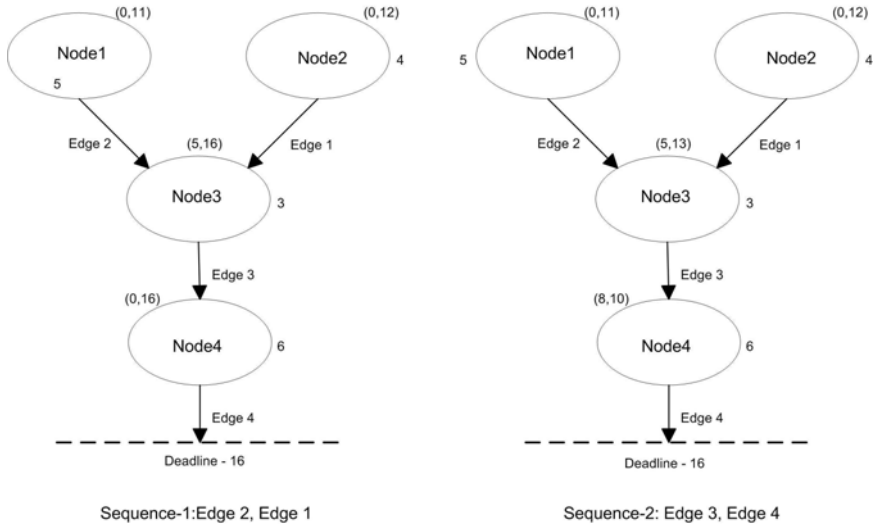

Fig. 3.3: Forward pass-I.

3.4 Propagator Implementation

A propagator is an implementation of a constraint and accepts finite-domain variables from the constraint store. Propagation may result in pruning the domain of the finite domain variables, but the output of the propagator should confirm to the constraint imposed on the range of the variable by the constraint store. The constraint that is implemented for the analysis of the DFG problem is of the form $A+B <= C$ where A, B and C are finite-domain variables. Conditions to be satisfied by the input parameters for propagation to be feasible and the internal architecture of the propagator are explained in this section.

Since A, B and C belong to the domain of non-negative integers, the constraint can be better explained by illustrating it with the aid of a number line. Assume that A, B and C have the following values assigned to them initially:

$$A = (1, 3),$$

$$B = (1, 6),$$

$$C = (1, 5).$$

When the constraint $A+B <= C$ is applied to this set of variables, it can be seen from

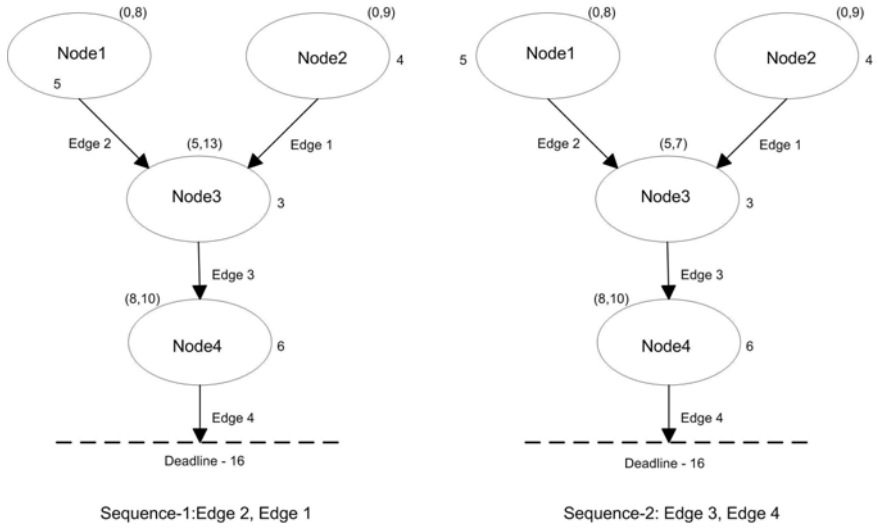

Fig. 3.4: Forward pass-II.

the number line shown in fig. 3.7 that propagation is feasible due to some region of $A + B$ overlapping with the line C. This implies that upper bound of the sum $A + B$, 9 is greater than the upper bound of C which is set to 5 and the lower bound of the sum $A+B$ is greater than the lower bound of C. To satisfy the constraint imposed the propagator prunes to a value 5, which is equal to the upper bound of C and raises the lower bound of C to meet the lower bound of $A + B$. The lower bound of $A + B$ cannot be reduced further to meet the lower bound of C because that would conflict with the constraint on the interval of A and B imposed by the constraint store. The results of pruning are shown as a number line in fig. 3.10. It remains to be seen as to how the pruning of $A + B$ results in pruning of the individual variables A and B. Since interval-based pruning is adopted, pruning should result in an interval which is true for all possible combinations of variable assignments without violating the interval constraint imposed by the constraint store. When the minimum value of A is assigned then the maximum value of B that can be assigned without exceeding the upper limit of C is 4. Similarly, for the minimum possible assignment to B, the maximum value that can be assigned to A is 4. But assigning this value violates the bounds set on A by the constraint store. Hence, A can only assume 3 and not 4. B is pruned to 4 and A requires no pruning resulting in these assignments after propagation. The values assumed

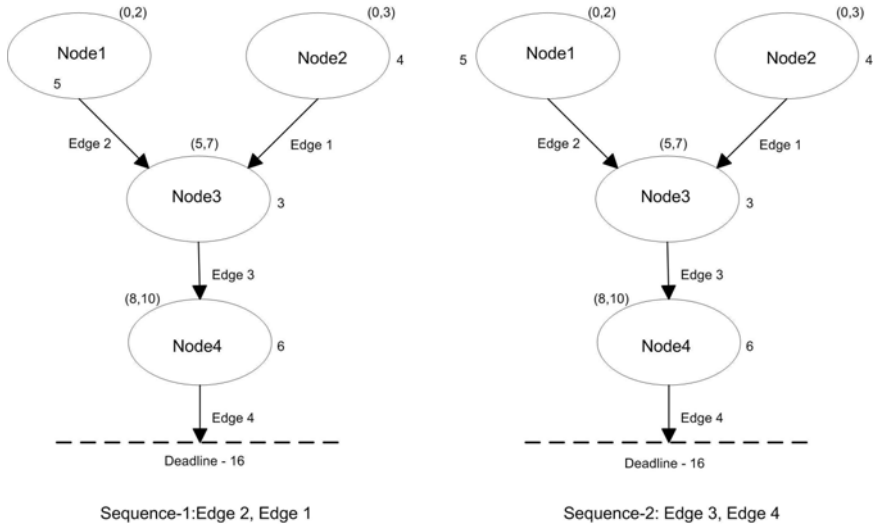

Fig. 3.5: Forward pass-III.

by the variables as a result of propagation are mentioned in eq. 3.4.

$$\begin{aligned} A &= (1, 3) \\ B &= (1, 4) \\ C &= (2, 5) \end{aligned} \tag{3.4}$$

Another scenario is shown in fig. 3.8, where there is only one point at which the two number lines coincide. This may be observed at point 4 on the scale. Upon subjecting these parameters to pruning, the line reduces to a single point as illustrated in fig. 3.11. The implementation of the propagator also considers the case for entailment, failed propagation, and propagation involving constant parameters.

- **Entailment**: When an attempt to propagate a set of variables that has already been pruned is done, no further propagation occurs since there is no new information available to the propagator for propagation to occur. In that case the propagator is said to be an entailed propagator. If the values of A, B and C from eq. 3.4 are provided as input parameters to the propagator, propagation will not occur.

- **Failed propagator**: Consider the input scheme to the propagator illustrated in

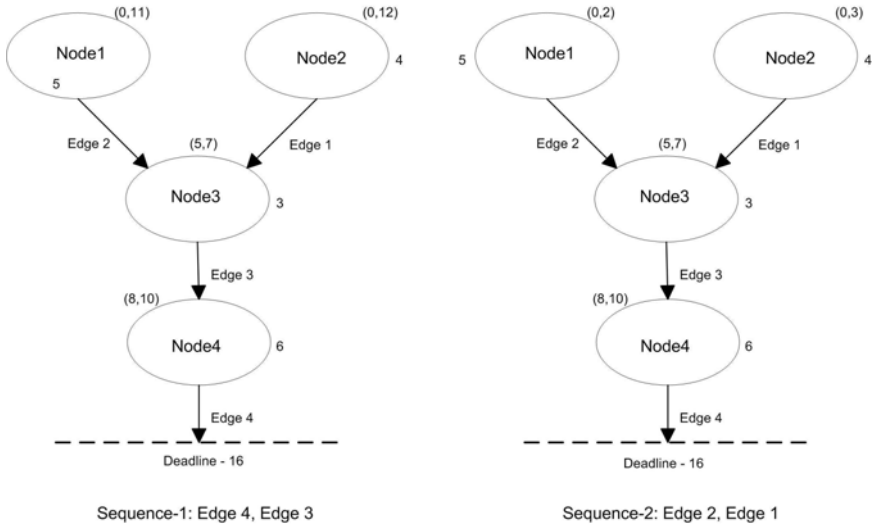

Fig. 3.6: Reverse pass-I.

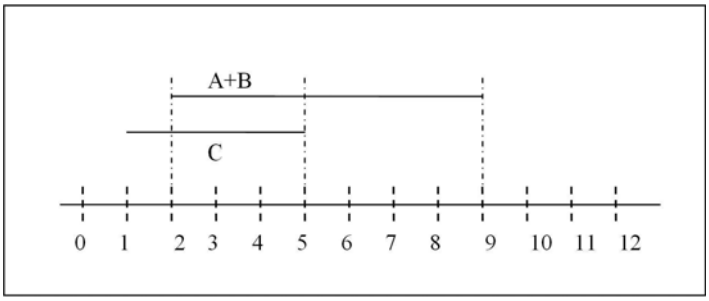

Fig. 3.7: Feasible number line.

fig. 3.9. It may be observed that there is absolutely no overlapping region satisfying the constraint imposed by the propagator. This leads to a constraint violation.

- **Propagation of a constant parameter**: A finite-domain variable in which the lower bound is equal to the upper bound is known as a constant or grounded finite-domain variable. When a constant FD variable is passed through a propagator it remains unmodified.

3.5 Algorithm for Implementation of Propagator

We now present the algorithm used in the implementation of the propagator for the constraint given by eq. 3.3. The pseudo-code for propagation that follows is a translation of

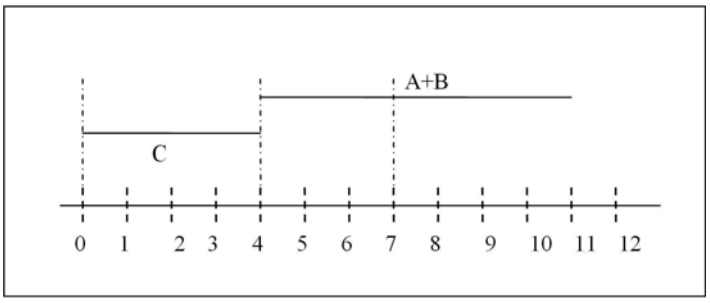

Fig. 3.8: Boundary scenario.

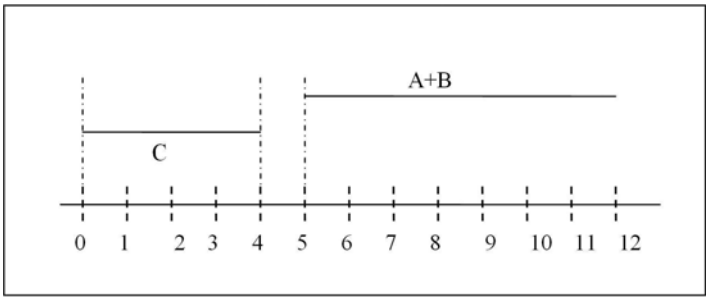

Fig. 3.9: Infeasible number line.

the logical explanation provided in the previous sections. The implementation follows the principle of partial-arc consistency checks for interval pruning as explained in Fromherz [2], Henz [5] and Bartak [6].

Pseudo Code for Complete Propagation

The algorithm for the implementation of a complete propagation step is illustrated as a pseudo code in fig. 3.12.

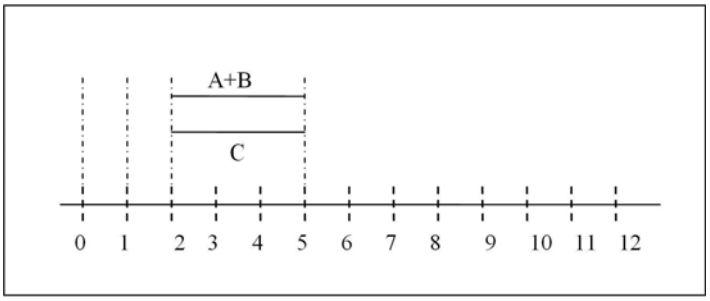

Fig. 3.10: Results after pruning feasible number line.

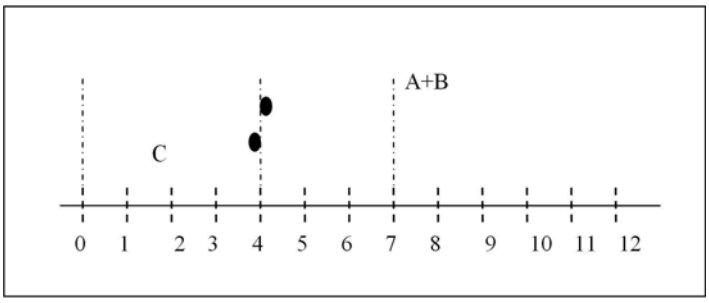

Fig. 3.11: Results after pruning the boundary scenario.

While [new information can be conveyed to the constraint store]
  For [all propagators incident upon the constraint store]
    (1)Read input parameters A,B and C
    (2)Check input parameters from (1) for validity
    (3)**If** (2)= VALID
      (3.1)Prune left-hand-side parameters of the $A + B \leq C$—A, B
      (3.2)Prune right-hand-side parameters of the $A + B \leq C$—C
    End If
    (4)**If** (3.1) OR (3.2) [update the constraint store]
      assert the condition: new information conveyed to constraint store
    End If
  End For
End While

Fig. 3.12: Pseudo code for complete propagation.

3.5.1 Pseudo Code for Pruning LHS Parameters

Figure 3.13 provides pseudo code and encodes the algorithm for Step (3.1) of for pruning the parameters A and B of the constraint $A + B \leq C$.

3.5.2 Pseudo Code for Pruning RHS Parameters

Figure 3.14 explains the pseudo code for Step (3.2) for pruning the parameter C of the constraint $A + B \leq C$.

3.6 Contribution - The Consolidator

In this section we discuss the design for performing the propagation step in parallel. Finite domain constraints are inherently amenable to parallelization. By definition, a

Upper Bound $\rightarrow UB$
Lower Bound $\rightarrow LB$
(1)Determine $SUM = UB(A) + UB(B)$
(2)**If** $SUM > UB(C)$ then
    $SUM = UB(C)$
End If
(3)$DIFF = SUM - LB(B)$
(4)**If** $DIFF < UB(A)$,then
    $DIFF = SUM - LB(A)$
End If
(5)$DIFF = SUM - LB(A)$
(6)**If** $DIFF < UB(B)$, then
    $UB(B) = DIFF$
End If

Fig. 3.13: Algorithm for pruning LHS parameters.

Upper Bound $\rightarrow UB$
Lower Bound $\rightarrow LB$
(1)Determine $LB_{SUM} = LB(A) + LB(B)$
(2)**If** $(LB(C) < LB_{SUM})$ and $(UB(C) > LB_{SUM})$
    $LB(C) = LB_{SUM}$
End If

Fig. 3.14: Algorithm for pruning RHS parameters.

propagator is a computational agent that operates concurrently with other propagators to amplify the constraints in the constraint store [7]. For concurrent operations data consistency of the constraint store is required. In order to ensure this, it is imperative that all writes to the constraint store be synchronized and that no race conditions occur during an update to the constraint store. This is achieved by means of a consolidator. The design and purpose of the consolidator is explained.

The consolidator is essentially a comparator that acts as a check point for all updates to the constraint store. The consolidator only accepts updates which improves the current FD value stored in the constraint store. Only if the lower bound of the update request is greater than the lower bound in the constraint store or if the upper bound of the update request is lower than the upper bound in the constraint store is the update accepted. This check on the bounds is correct because propagation fundamentally only leads to domain

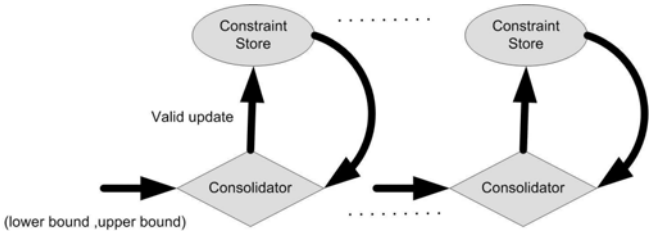

Fig. 3.15: Consolidator.

pruning and narrowing of bounds.

The consolidator check prevents a pruned FD variable from being over-written by values which negate the effect of the pruning. In the absence of a consolidator, inconsistent and redundant updates to the constraint could occur. The consolidator is illustrated in fig. 3.15.

To achieve parallel processing on multiple processors, the constraint store was partitioned and distributed onto several processors on the distributed-memory architecture discussed in Chapter 2. Distributing the constraint store onto several processors eases the communication bottle neck and synchronization issues encountered with having numerous accesses to a single data repository. This required a consolidator to be associated with every partitioned constraint store to ensure data consistency.

3.7 FPGA Implementation of Propagator

The principles and algorithms that were discussed in sec. 3.5 are used in the implementation of the propagator. The two different implementations that were performed on the FPGA were based on the same algorithm. The first approach was a hardware implementation of the constraint propagators while the second utilized multiple soft-core processors as a platform for software-based parallel constraint solving.

3.7.1 Hardware Implementation of Propagator

Our first attempt at a propagator implementation revolved around an hardware implementation using Hardware Description Language (HDL) that was used to implement the propagator on an FPGA. A single instance of a propagator in HDL followed the algorithm

discussed in fig. 3.12.

The hardware implementation of the propagator was designed by means of a state machine. The finite state machine design driving this implementation is detailed in this section. The several versions designed and tested included a single propagator unit, two propagator units accessing a single constraint store and a pipelined implementation of the propagator. The design of the single instance of the propagator is explained in this section. The other implementations are based on the same design principles and and are omitted for brevity.

The main elements for a single instance of a propagator unit are the memory, the propagator itself, which implements the less-than-equal-to constraint and the controller. The system level design shown in fig. 3.16 shows how all the pieces are interfaced together for a single propagator unit.

- **Memory**: A single propagator unit consists of a memory module which contains the addresses of the tuple of finite domain variables used for propagation. In other words, constraint allocation is done in the memory. Memory is also required for the constraint store. Block RAMs present in the FPGA are used for this purpose.

- **Propagator**: A combinational design of a propagator performs a fixed "ALU" operation in this design. This computational unit is an implementation of the constraint $A + B <= C$. The results of propagation are provided back to the controller which updates the constraint store.

- **Controller**: The controller implements the control logic necessary to fetch data, propagate and write data back to the constraint store. The control logic is implemented as a state machine. The design of the state machine governing this controller follows this section.

Figure 3.16 provides an overview of the hardware design consisting of the controller, a memory unit to store the address of the FD variables to propagate, the memory unit that is used to store the FD variables and the propagator. The memory containing the address of

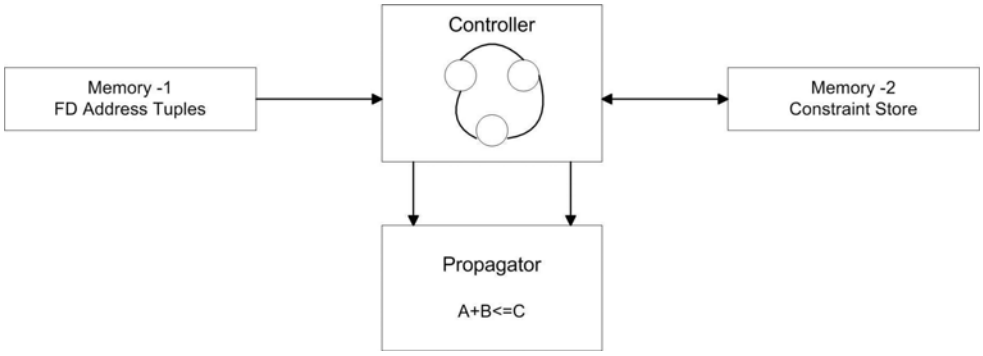

Fig. 3.16: System design of a single instance of a propagator unit interfaced to constraint store.

the FD tuples is equivalent to the instruction memory while the other memory for storing FD variables is equivalent to the data memory.

An illustration of the FSM with its state transition diagram is shown in fig. 3.17. This FSM contains seven states and the role of each state is indicated by a label. Some state transitions are guarded while the unguarded transitions occur at the beginning of a clock cycle.

3.7.2 Hardware Design of Parallel Propagators

The top level interface of the hardware block of the propagator is shown in fig. 3.18. The inputs to the propagator include a set of FD variables A, B, and C and the outputs of the propagator are the pruned FD variables. The update line is asserted only when any of the input FD variables are pruned else it remains low.

The previous section covered the hardware design of a single propagator. In this section, an overview of the design of two propagators operating concurrently is explained. As shown in fig. 3.19, the design consists of two propagator units working concurrently and operating on a single constraint store. The dual port capabilities of the BRAM memory module were utilized in order to increase the memory throughput which was configured to contain one Read only port and another Read/Write port. A bus arbiter was designed to resolve memory access contentions since both propagator units access a single constraint store.

The memory controller avoids contentions when read and write operations occur at

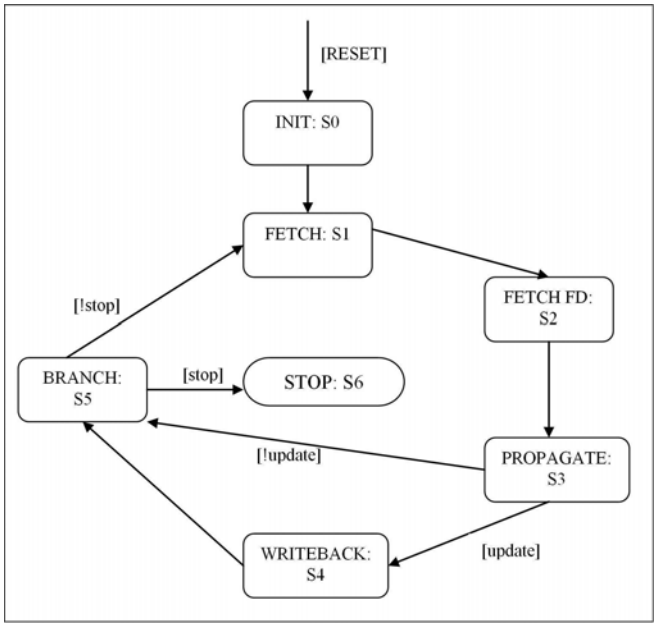

Fig. 3.17: FSM for the controller.

an address location during the same clock cycle. When such a situation arises, the read operation from the memory is stalled until the write operation is completed. The memory controller also contains the consolidator which discards updates to a finite domain variable which does not prune the existing finite domain variable present in the constraint store.

3.7.3 Pipelined Implementation of a Single Propagator Unit

A pipelined implementation of a propagator was also implemented. Resorting to pipelining improved the memory utilization significantly with approximately one read/write access occurring every clock cycle. The controller for a single propagator unit was redesigned to accommodate pipelining wherein simultaneous read and writes from/to the constraint store memory could now occur in comparison to the sequential state machine implementation.

3.7.4 Memory Performance Bottleneck with Concurrent Propagators

Since pipelining results in higher memory utilization, operating multiple instances of

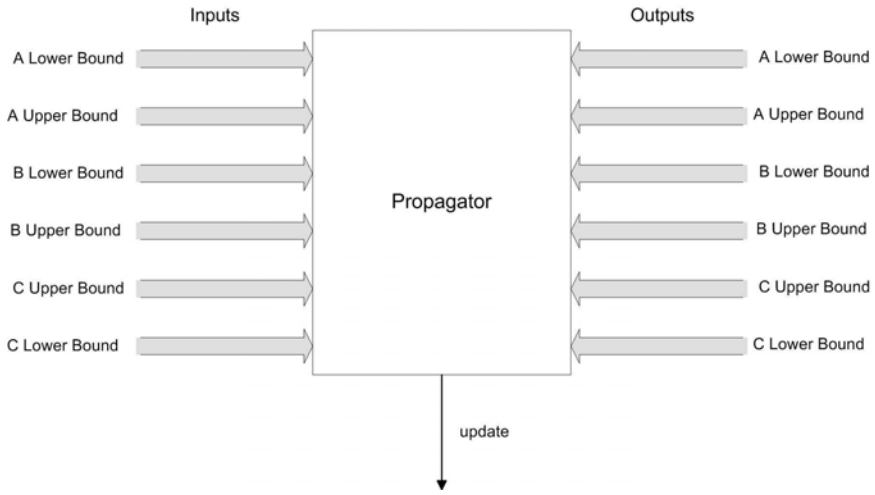

Fig. 3.18: Hardware block of a propagator.

this unit in parallel requires addressing the bus contentions when accessing a single constraint store. Accessing a single constraint store would therefore decrease the throughput due to the delay cycles introduced by the bus arbiter.

A highly scalable parallel architecture of propagators could contain single units of pipelined architectures with each unit restricted to access only one constraint store. In such a design when a large data flow graph needs to be pruned using sets of parallel propagator units, certain nodes of the DFG get allocated to a particular constraint store only. Being a connected graph, modification of the FD variable for any node in one constraint store needs to be communicated to the other nodes that are its successor or predecessor and there is significant inter-memory communication that is involved in this approach.

3.7.5 Results of Hardware Implementation

The synthesis report from Xilinx tools and the results of benchmarks for the constraints obtained from the LU-decomposition graph (shown in fig. 5.6, page 87) are provided in this section by executing them on the various hardware designs.

- **Single non-pipelined propagator unit**
  - Slice utilization in the FPGA is 129 out of 5472 = 2%

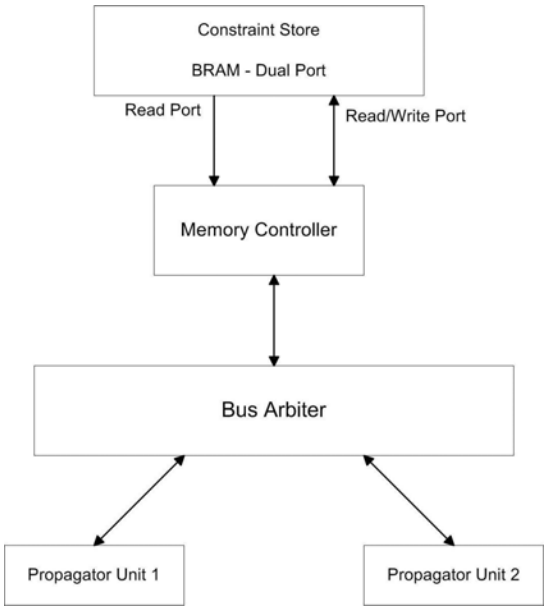

Fig. 3.19: Structure of two propagator units.

  - Synthesized timing: 76.4 MHz
  - Time for complete propagation: 5.69μs

- **Two non-pipelined propagator units**
  - Slice utilization in the FPGA is 332 out of 25280 = 1%
  - Synthesized timing: 194 MHz
  - Time for complete propagation:
    1. Two propagator units with forward traversal only—21.84μs,
    2. Two propagator units with reverse traversal only—19.85μs,
    3. Two propagator units with forward and reverse traversal—4.42μs.

- **Single pipelined propagator unit**
  - Slice utilization in the FPGA is 393 out of 25280 = 1%
  - Synthesized timing: 363 MHz
  - Time for complete propagation: 3.03μs

3.7.6 Software Implementation of Propagator

The algorithms discussed in sec. 3.5 were used in the parallel execution of propagators on multiple soft-core processors. Each soft-processor in the FPGA contained a portion of the partitioned constraint store. The computational agents executing in each processor collaborate using the underlying infrastructure discussed in Chapter 2 to achieve parallel processing.

From the perspective of multiple processors, the input to the consolidator could also represent data from a computational unit that is external to the processor. Therefore, the consolidator acts as a check-point through which internal and external updates to the constraint store occur.

3.8 Hardware vs. Software Implementation

We compare the advantages and disadvantages offered by an hardware implementation over software.

3.8.1 Infeasibility of Hardware Implementation

The hardware implemented appeared promising in terms of providing better performance than software but was not a viable option due to the following reasons:

1. **Scalability of the architecture**: The hardware design of a pipelined implementation required several design overhauls in order to make it scalable to support a parallel architecture. Issues that required to be addressed in the design included communication and data synchronization between the various propagator units. The encoding of this synchronization is highly topology dependant and difficult to modify once expressed in a hardware-based design.

2. **Flexibility**: Future revisions of the constraint solver might require the implementation of a different propagator. Making these changes to the hardware design presents significant design challenges due to rigidity of hardware design.

3. **Design time**: The time involved in modifications to the hardware interface, behavior, and testing increased as the complexity of the design increased.

3.8.2 Advantages of a Software Implementation

The implementation of the propagator proved feasible on a soft-core processor known as MicroBlaze from Xilinx with its supporting infrastructure. This approach followed a hardware-software codesign approach.

1. Design time: The design time was reduced due to the use availability of soft-core processors in the form of IP cores. The software implementation consisted of compiling the algorithm using the C compiler provided by the tool chain.

2. Scalability: Scaling up the design by increasing the number of computational units required for parallel processing was possible due to the support structure provided by the hardware peripheral IP cores. The IP cores address several bus contention issues that were previously discussed. However, software support was still required to address various data synchronization issues. The hardware/software effort involved in scaling the design was significantly reduced. Once the firmware code for one computational unit is developed, with minor configuration changes the same code can be replicated onto other computational units.

3. Flexibility: New propagators can be introduced into the existing design without having to change the underlying hardware design because the propagators are implemented in software. Once the design is scaled up in hardware, any future revision to the problem would more likely involve a software change.

Hardware implementation certainly shows better performance metrics than a software implementation. But these are metrics that were obtained for a propagator running in isolation, as opposed to it running and sharing data with other propagator units. Further, there is no notion in this implementation of global synchronization and control in preparation for a distribution step, so when a hardware component is integrated with other software components or other global synchronization and control, its performance will degrade.

There are several other issues that need to be addressed when implementing the constraint solver completely in hardware such as shadow copies, sharing constraint stores, global synchronization and inter-processor communication. While these features are possible to implement in hardware, they will add overhead, both in time and area and this is an area for further research. There is significant potential for a mixed hardware/software codesign finite domain constraint solver which makes use of the FPGA fabric as well as the MircoBlaze processor, but addressing such issues fell out of scope for the research reported in this thesis.

Chapter 4

Distribution

This chapter provides an overview of distribution and explains the parallel implementation of the distribution algorithm adopted in this research project. In review, a solution to a constraint problem constitutes a variable assignment that satisfies all the constraints in the constraint store as well as those imposed by the propagators. Propagation by itself only prunes the domain of variables in the constraint store and does not yield a solution. Distribution is the technique employed by the solver to introduce new information to the solution which cannot be formally proven through propagation alone. However, this new information, in the form of added constraints, is partially speculative, and may lead to a conflict. Checkpointing and backtracking search are employed to ensure that the search space is adequately covered, and that a correct solution if one exists can be found. The process of distribution involves branching followed by search. The subsequent solutions detail the distribution approach and the implementation employed in this project.

4.1 Branching

Branching presents the core operation of a distribution step. Branching is performed by subjecting one of the variables from the constraint store to an additional constraint. The constraint store is modified by the application of this new constraint which ideally triggers additional propagation. This resulting propagation step will halt in one of the following states:

- A solution, where a variable assignment occurs to all domain variables in the constraint store;
- A failure, due to inconsistency between the propagators and values in the constraint store; or

- A stable but unsolved constrained store.

A stable but unsolved space is termed a choice point since the solver must choose new information to "tell" the constraint store in order to proceed. We apply branching only when coming to a choice point. In order to guard against the possibility of branching leading to failure, the solver creates and stores a copy of the current constraint store, prior to executing the branch. This allows the solver to come back to this point if it determines that the branch yielded an incorrect result (i.e., a failure). Successive branching steps yields a search tree whose nodes represent choice points, and edges represent the inserted constraints. The constraint inserted during branching determines the pattern of the search tree that develops. There are many algorithms which can be used to guide the branching process. Branching involves two steps:

1. Variable selection,

2. Constraint selection.

Naive and first-fail are two heuristic variable selection algorithms. The naive heuristic involves simply selecting the next unbound variable to distribute on in an arbitrary or random order. In contrast, the first-fail heuristic involves unbound variable selection from an ordered list. The unbound variables are ordered based on the cardinality of their domain—from smallest (most constrained) to largest. First-fail selects variable with the smallest domain. The first-fail heuristic works on the principle that likelihood of the application of a constraint is most likely to fail for a variable that is most constrained.

Generic constraint selection approaches include min-value, max-value or mid-value (where the solver selects the minimum, maximum, or middle value, respectively from the domain of the selected variable, and binds that variable to the selected value). However, in typical solvers these can be replaced by domain-specific extensions as well.

Once variable has been selected for branching, the solver determines a constraint to tell the store involving the selected variable. This usually takes the form of $v := val$, where v is the selected variable and val is the value assigned to v selected from the domain of v.

The complement of the constraint $v \neq val$ is inserted into a saved copy of the constraint store for use in backtracking.

Since the selected constraint represents a guess, at each distribution step, the solver creates two complementary constraint stores from a single choice point. The search algorithm employed dictates the order in which branching is performed.

4.2 Searching

While creating a search tree by branching from choice points, the solver employs a search algorithm to visit the nodes in the tree to arrive at a solution. The order in which the tree is to be traversed and the type of solution required is defined by the search algorithm that is used. For instance, Mozart employs a depth-first search technique borrowed from Prolog to traverse the search tree. The notion of depth-first search is also used by branch-and-bound algorithms used in linear programming. Studies have shown that in general depth-first search requires fewer computational resources in terms of memory usage and time required to arrive at a solution.

When there are numerous solutions available to a given problem, there are various ways of reporting a solution. One can choose all solutions which would require the traversal of the complete search tree. Alternatively, one could look for the best solution or just the first solution that is encountered during traversal. Choices made for searching has an impact on the amount of memory that is required by the solver and the amount of computation effort that is involved. We currently only consider a first-solution based search.

Associated with the process of searching, is backtracking. With backtracking the solver reverts to a previous state or node of traversal and proceeds with the traversal of the nodes in the search tree. For the search tree depicted in fig. 4.1, each node represents an additional choice point, an entailed space or a failed space. There are four choice points, three failed spaces and two solved spaces depicted. Traversal from one node to another occurs by applying all the propagators for the given problem on the constraint store. So as one traverses deeper into the tree, the values of the constraint store changes due to the methods of variable assignment performed in branching and subsequent propagation. To traverse all

the nodes in a search tree requires knowledge of the state space of the node that is being branched from.

Regaining knowledge and restoring the state space of a node upon revisiting it requires memory storage of the entire state space. Storage of state space can be avoided by recomputation which involves arriving at a given state space by successively performing branch and search from a previously stored state. The number of recomputations involved in this process depends on the number of previously stored nodes. Consider the search tree depicted in fig. 4.1 [10]. If the only stored state was that of the root node, then to reach the deepest solution node in the tree would require three recomputations. On the other hand, recomputation can be completely avoided by storing the state space before moving to another node. But this approach would require increased memory storage. Therefore, a trade-off between computational effort and memory storage is required when designing the backtracking algorithm.

The backtracking scheme implemented in this project avoided recomputations by storing the contents of the constraint store at each distribution point. The size of the problem was small enough to meet the memory restrictions imposed by the computation units. Of course, the depth of the search tree that could be traversed was dictated by the amount of memory available for storage. Having, provided an overview of the implementation decisions made for distribution, the next section discusses the design and implementation of this algorithm on a multi-processor MicroBlaze system.

4.3 Implementation of Distribution on a Multi-Processor Platform

This section details the implementation of the distribution step for a finite-domain on the MicroBlaze infrastructure outlined in Chapter 2.

As discussed in Chapter 3, the constraint store is partitioned and distributed among the processors in the system. Propagation in one constraint store can be triggered either due to information update as a result of an internal computation or due to an update to the constraint store occurring from an external entity. This is shown in fig. 4.2 by means of a state-transition diagram which is applicable to each processor.

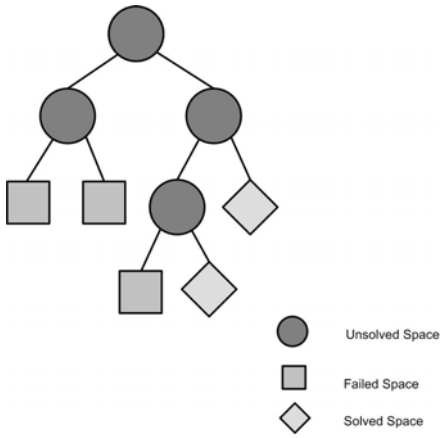

Fig. 4.1: Example search tree with circles representing unsolved space, square representing failed space and diamond representing a solved space.

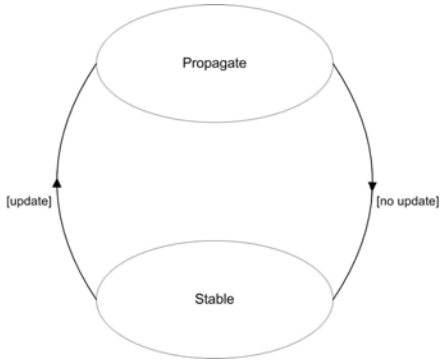

Fig. 4.2: Transition to stable state.

In contrast to the propagation step, for concurrent distribution, data synchronization among all the constraint stores must be ensured during every step of the distribution so that they globally represent one unified constraint store. When synchronization during distribution fails between processors, the state of one constraint store becomes inconsistent with the state of another constraint store. In other words the set of finite-domain variables in the constraint store no longer holds the same value that an equivalent unified, global constraint store would for the same distribution step.

4.4 Measures Taken to Ensure Synchronous Behavior

In order to ensure this synchronous behavior among the multi-processor network, a

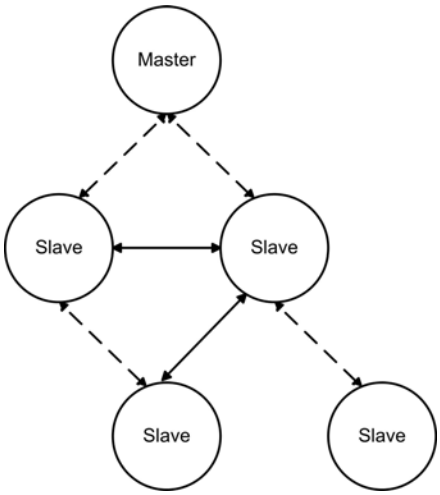

Fig. 4.3: Broadcast tree.

broadcast communication pathway structured as a tree was established with the processor at the root of the tree being the one responsible for decisions impacting the behavior of the entire network. Having a single controller for the entire network of processors ensured that all processors synchronized to the same state as directed by the root processor. The processor at the root node of the broadcast tree infrastructure was designated as the master. All other child processors responded to instructions from the master. An example of a broadcast tree structure configured for this purpose is shown in fig. 4.3. The solid lines indicate communication links that are solely used for propagation. The dotted lines indicate inter-processor communication links that can be used for propagation as well for broadcasting synchronization commands and other instruction required during distribution. Observe in this figure that there is a unique traversal to reach any slave (child) processor from the master processor at the root. Similarly, there is only one way in which one can traverse from any child processor to the master.

4.4.1 State Machine Design Philosophy for Individual Processor

The basic behavior of every processor is governed by a Finite State Machine (FSM) and the process of distribution itself is designed as a state machine. The state transitions in each processor are performed in response to either an event generated natively from within

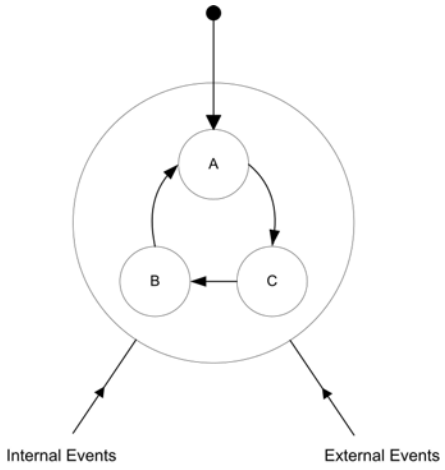

Fig. 4.4: Controller.

the processor or due to external events. To achieve a uniform and consistent distribution, all child processors sync-up to a state as instructed by the master or the root node of the tree before the start of a new distribution step. When the master receives confirmation of this synchronization, it transitions to the next state in its state machine. Figure 4.4 illustrates a finite state machine that responds to both internal and external events. Internal events are generated within a processor whereas external events are those that are generated by other processors in the network that arrive asynchronously. Since the state machine was driven by both internal and external events, transition guards had to be carefully designed. Since multiple events could occur simultaneously, priorities had to be assigned when making decisions for state transitions. More details on the state machine design are explained subsequently.

4.4.2 Communication of Instructions

Communication among the processors occurs by means of message passing. The arrival of a message at a processor signals the occurrence of an event. How the event is processed is modeled by the state machine. The events trigger appropriate state transitions in that processor. There are different types of messages that can be received by a slave processor.

- Message without confirmation: The slave processes this request message received from its immediate parent without sending any confirmation.
- Message with confirmation: This event is processed by sending a confirmation back to the parent processor. Upon receiving a confirmation message, each processor in turn confirms to its immediate predecessor until the message reaches the master processor.
- Broadcast message: A broadcast message from the master is intended for distribution over the entire tree. The message travels from the master processor to all slave processors in the tree. Each processor upon receiving the broadcast message simply broadcasts the message to its immediate successors.

4.4.3 Reasons for Choosing Tree Structure

There are several reasons for choosing a tree structure for the communication path for distribution. The purpose of the broadcast tree is to establish a mechanism for global synchronization and communication. Its goals are:

- A fixed tree structure is easier to debug in case of a problem since the number of inter-connections for each node is limited and the flow of information through the broadcast topology is known.
- The number of branches for any node is at most two. So a node has to broadcast to at most two other nodes or wait for confirmation from at most two other nodes. This creates a form of load balancing so a particular node is not loaded with too many broadcasts or confirmations increasing the communication latency of that node. This decreases the communication overhead in any node.
- The pattern of the binary tree structure fans out from a single root node facilitating the use of a single point of control which is designated as the master. This achieved global synchronization and control.
- Existing infrastructure may be reused. For instance, a single FSL link may be used for distribution as well as for propagation.

4.5 State Machine Description

The description of the state machine determining the behavior of each processor is described below and is as depicted in figs. 4.5, 4.6 and 4.7. It should be noted that none of the state transitions occur in a time-step synchronous manner in any of the processors. It is significant that while the state transitions on different processors do not synchronize on common events, due to the presence of a single master which controls these transitions, the overall effect of pseudo-synchronous transitions is achieved. Observe that the FSM for the master and its children have minor differences in their behavior. The master always takes the lead in the transition and the rest follow the instructions from the master.

4.5.1 Initializations Performed

The constraint store which is distributed among all the computational units is initialized in each of the nodes before entering the state machine.

4.5.2 Propagate State - State A

All computing units enter this state after initialization. It is assumed when entering this state after initializations, that the constraint store is feasible for propagation. Propagation of all the constraints that are allocated to a particular computational node is triggered in this state. Due to the part-whole relationship of the constraint store in each processor, propagation in one computational unit may trigger propagation in other units in the multi-processor unit. All processors simultaneously propagate until their propagators reach a stable state when no new information strengthening its constraint store can be added. This refers to *State A* in figs. 4.5, 4.6 and 4.7.

4.5.3 No Update - State B

The transition from propagate to No Update is based on information local to the processor. If given the current state of its local constraint store, no propagation can occur after a brief period of time, then the processors transitions to the No Update state. However, if new data arrives from another processor, this immediately causes a transition back to the

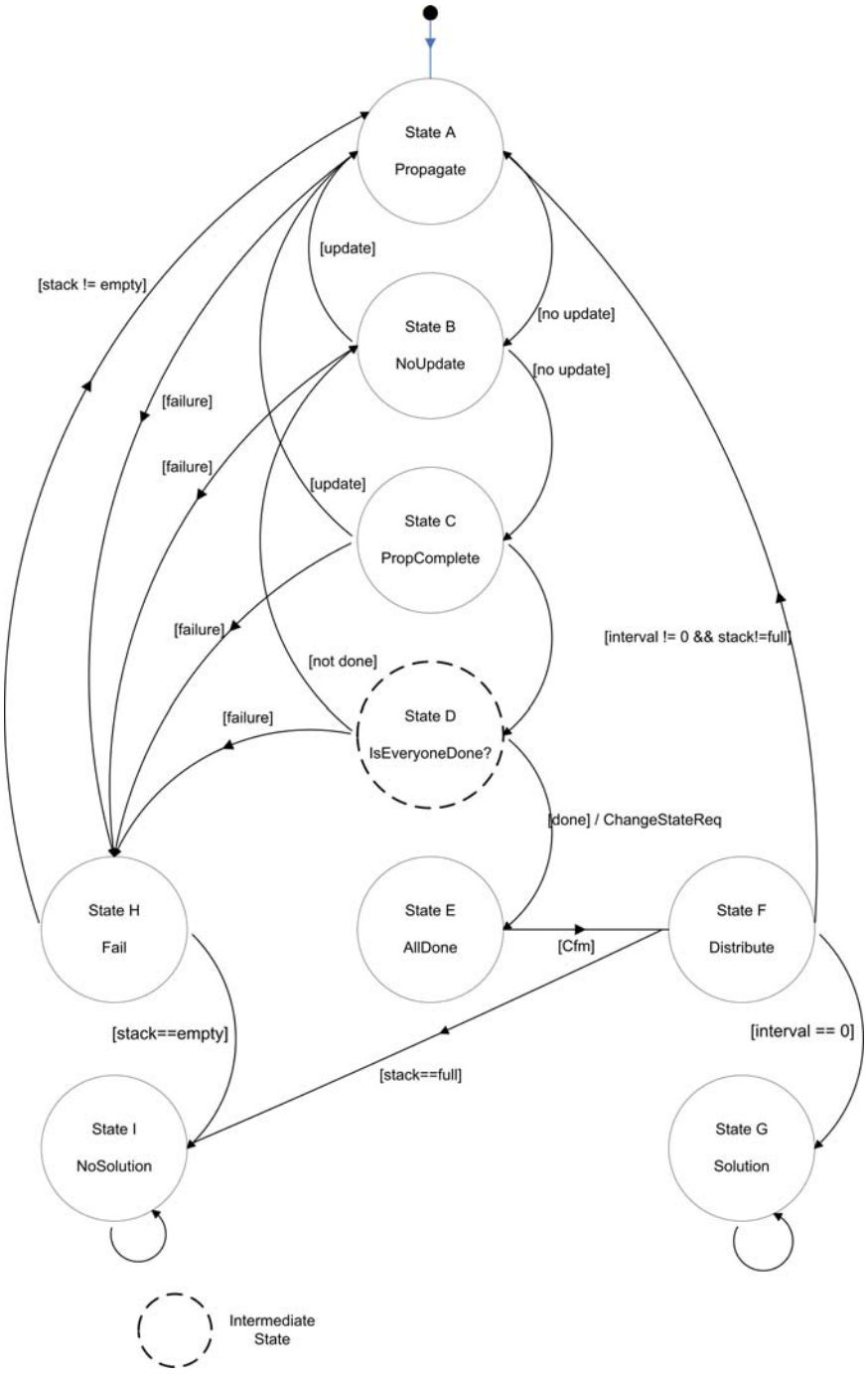

Fig. 4.5: Master state.

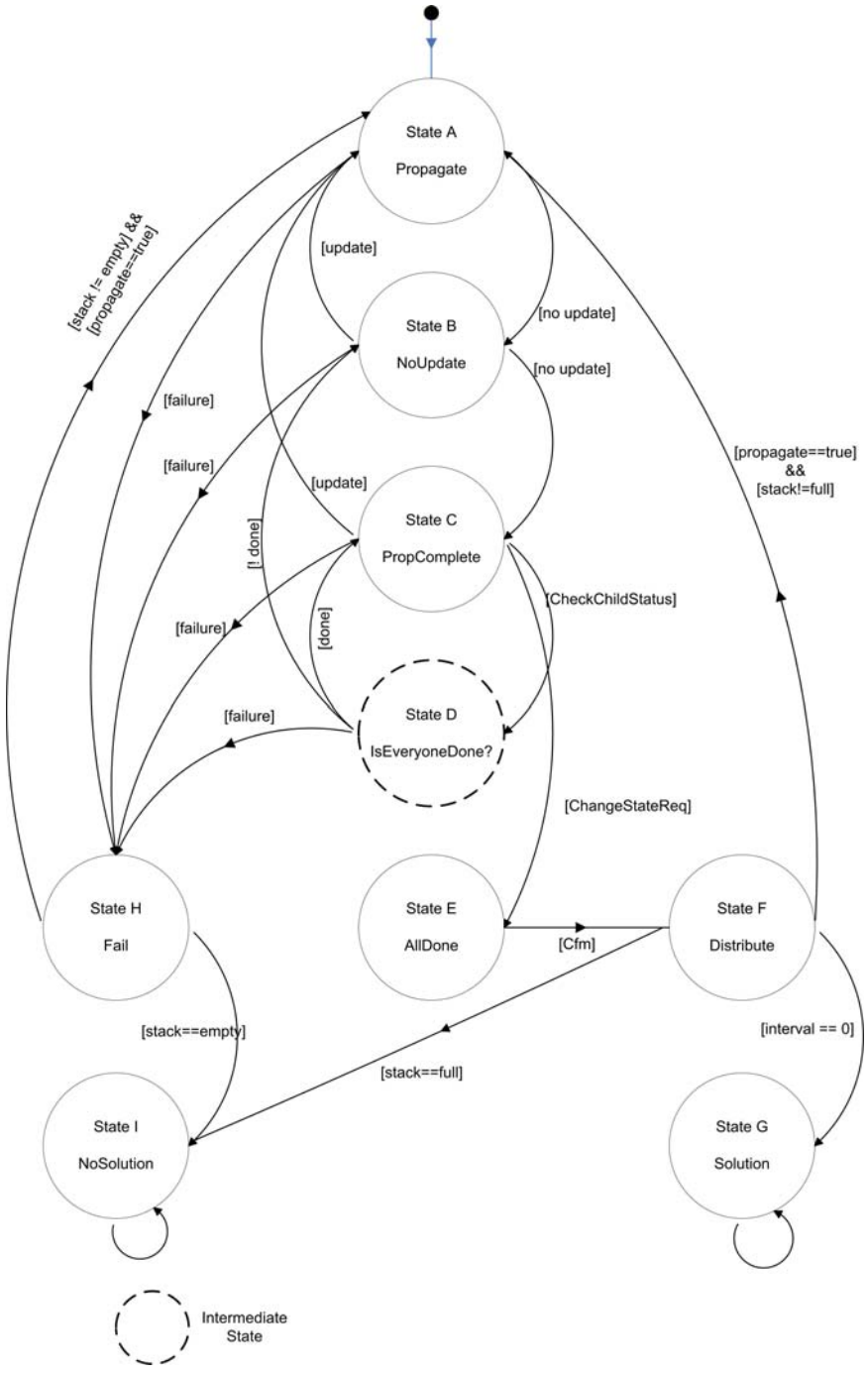

Fig. 4.6: State machine non-leaf processor.

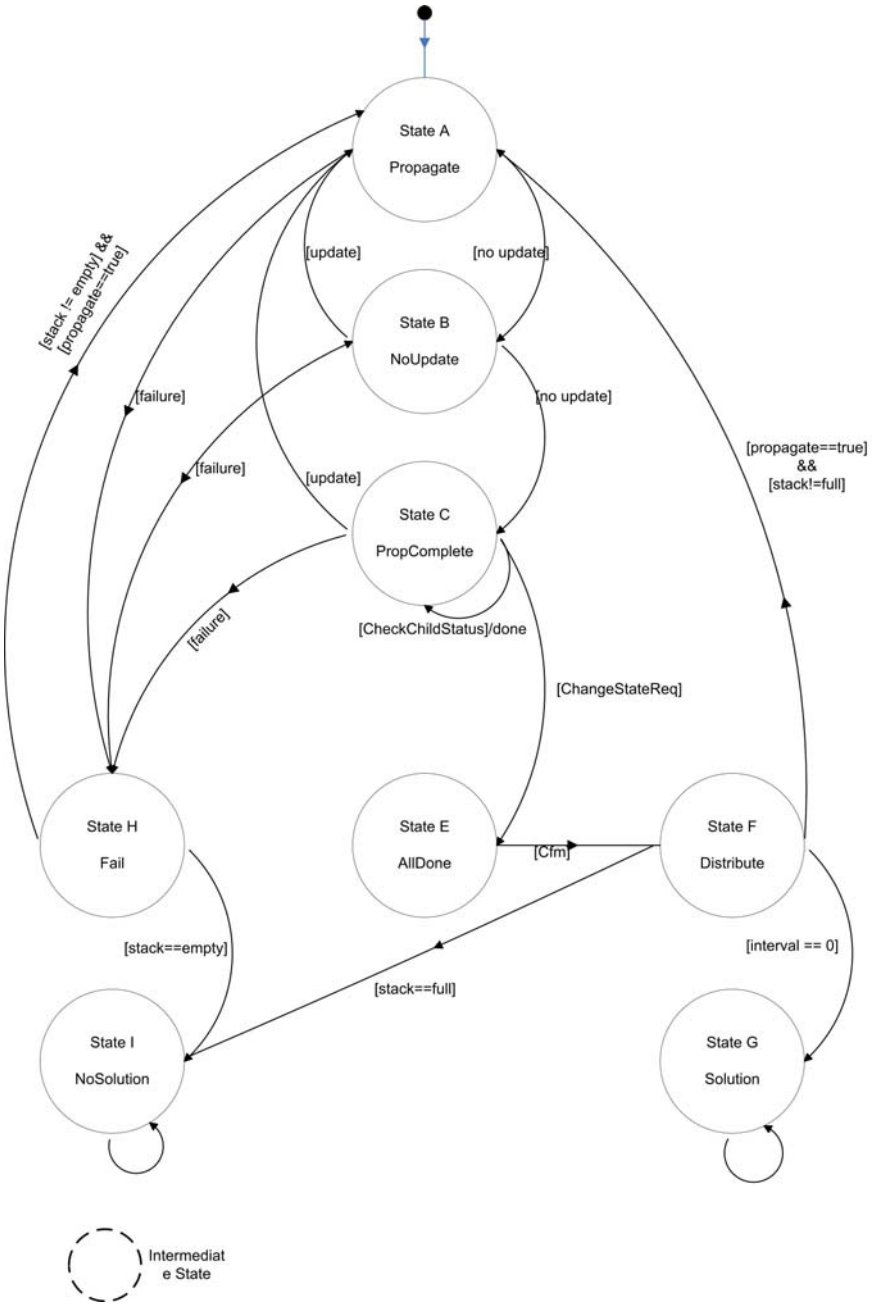

Fig. 4.7: State machine child processor.

Propagate state, where the propagators are invoked in an attempt to perform propagation. This refers to *State B* in figs. 4.5, 4.6 and 4.7.

4.5.4 Propagation Complete - State C

When updates to the constraint store do not occur even in the No Update state, it is decided that propagation is complete. If an update to the constraint store occurs in this state, propagation is restarted and the state machine transitions to *State A*. When any node reaches the No Update state and when no updates to its constraint store occur, it makes a transition to State C. The master and all slave nodes complete their share of propagation and reach their respective State C and remain in this state awaiting instruction from the master processor.

The transitions until State C starting from State A occur independent of the master in all processing units.

4.5.5 Intermediate Wait for Child Status - State D

Once the master cannot propagate any further based on its constraint store, it must check to see of all other nodes are similarly "stuck". It does this by issuing a broadcast message and waiting for a reply. Only when propagation in each of the processors is complete, can the propagation in the entire system be said to be complete. Upon entering this state, the master sends a message to its immediate children requesting their current state. A processor receiving this request from the master confirms completion of the propagation only if it and all its children have in turn completed propagation. Therefore, each interior, non-master tree node on receipt of this message must first broadcast it to its children and receive a reply prior to the request.

In this manner the message from the master processor at the root trickles down to a leaf processor. When a leaf processor receives the request, it simply responds with a message indicating its current state. It need not wait for other messages; and therefore needs no "wait" state. As shown in figs. 4.5, 4.6 and 4.7, only the master and all non-leaf nodes contain this intermediate transition state, State D.

4.5.6 All Propagation Done - State E

When the master receives confirmation that all its children have completed propagation, it is inferred that propagation in the entire system is stalled, either due to completion or pending further information from distribution. At this point, only the master is aware of this fact. To make all the computing units cognizant of the master's inference, the master next broadcasts another message to all its children indicating that the propagation in all computing units stands complete. Prior to processing with distribution, the master waits for acknowledgements from all nodes, to ensure lock-step synchronization.

Every slave node transitions to State E upon receiving this message from its parent. Being a broadcast request requiring confirmation, each slave node broadcasts the message to its children and waits for confirmation. Each processor in its turn sends a confirmation to its immediate parent only after reaching this state and upon receiving a confirmation from its descendant. The confirmation for this request serves two purposes.

- First, it indicates to a parent processor that all processors which descend from it have transitioned to State E, thereby achieving synchronization.
- Second, an additional message is sent along with the confirmation to the parent. This message contains information on a potential FD variable distribution candidate. This information is funneled to the master. This information helps the master processor in choosing the variable for distribution.

Once the master has determined that all nodes are paused waiting for more information, the master must select a variable and constraint in order to implement a distribution step.

4.5.7 Distributed Determination of First-Fail Heuristic

The process involved in determining which FD variable in the distributed constraint store has the smallest domain is discussed here.

Each processing unit keeps track of which variable local to it in its constraint store that has the smallest undetermined domain. A leaf node simply forwards this information to its parent in response to the confirmation message that all are done.

An interior node receives potential distribution candidates from each of its children and compares them with its own selected variable. The variable with smallest domain out of these candidates is then forwarded to the parent node.

Finally, when the master receives this information it can determine the variable to distribute upon. At this point the master transitions from State E to the distribution step State F.

4.5.8 Distribution Step - State F

Distribution involves branching and exploration and the scheme followed was explained previously. The branching scheme followed is the first-fail heuristic where two contradictory subproblems are created of the form $x = n$ and $x \neq n$ where x is a FD variable in the constraint store with the lowest interval which is determined by the master in the previous State E and n is the lower bound of x.

The exploration scheme works as follows. The first path explored for a potential solution is for the assertion $x = n$. The alternative choice is the complement of the constraint $x \neq n$. The alternative choice is explored when backtracking, when the initial path fails. To facilitate backtracking, the constraint store corresponding to the constraint $x \neq n$ is pushed onto the stack. This constraint is posted by deleting the value n from the constraint store.

The implementation of the distribution step in State F is now explained. First, a command to push the contradictory constraint is embedded in a message that is broadcast to all processors by the master. The processor that owns the variable corresponding to this constraint pushes its constraint store onto its stack after processing this constraint by directly deleting the value n from the variable x. This corresponds to incrementing the lower bound of the variable x. All other constraint stores that do not own this variable simply push their current constraint store as is without any modifications.

Second, after the master receives a confirmation of a successful save from its children, it proceeds to the variable assignment step. In this step, a command to perform variable assignment on the variable with the given global address is broadcast to all processors by the master. Every processor upon receiving this message checks to see if it owns that variable's

global address in its constraint store. If the variable's address resides in that constraint store, a min-value assignment is done by assigning the lowest bound of the variable to the upper bound of that variable making it a constant. The variable assignment triggers propagation in that constraint store. Other processors that do not own the distribution address do not modify their constraint store and restart propagation on their respective constraint stores.

These two steps are recursively repeated until a solution is found. The amount of memory available on each processor to save its constraint store onto the configuration stack certainly limits the depth of traversal to find a solution. If a problem requires a large number of distribution steps when traversing depth-first, then that would require multiple savings of the constraint store with each save corresponding to a choice point. This implies that the deeper the traversal, more the memory required for saving.

Saving the constraint store for every choice point may not be the best option in terms of memory efficiency but it avoids re-computation [8,9] and demonstrates proof of a distributed constraint solver on an embedded platform with memory constraint. Regardless of the amount of memory available for saving, the solver can always fail if it is thrown a sufficiently large problem involving many distribution steps.

4.5.9 Failure - State H

When there is inconsistency between the basic constraints imposed on the FD variables in the constraint store and the constraints imposed by the propagators, the constraint store is said to be in failed state. This is true even if only one of the propagators fail. Since the constraint store is distributed, a failed space for part of the constraint store implies that the whole space has failed. Hence, there should must be a mechanism to convey this information to the entire distributed constraint store.

This is implemented through synchronization with the master. A failure is communicated to the master which broadcasts it. When the master receives a message indicating a failure, it transitions to a failure state irrespective of its current state. After transitioning to the failure state, the master broadcasts a message to all the processors indicating that

the constraint store is now a failed space and waits for a confirmation message.

Every processor that receives this message transitions to the failure state. After reaching the failure state, each processor restores the previously stored constraint store by popping the value off the configuration stack. This implements backtracking. The constraint store that is popped represents the values for the alternative constraint that was pushed in the distribution state. When restoring the context, the values of the shadow variables residing in other constraint store also need to be restored. After these procedures, the slave processors send a confirmation back to its parent. Each parent in turn sends a confirmation back to its parent and finally it reaches the master.

After the master finally receives the confirmation, it broadcasts another message to all its children requesting them to restart propagation. Every processor, including the master, then restarts propagation and runs through the transitions of the state machine illustrated in figs. 4.5, 4.6 and 4.7.

4.5.10 Solution Found - State G

When a variable assignment occurs for all the variables in the constraint store satisfying all the constraints imposed by the propagators and that of the constraint store, it is inferred that a solution to the finite-domain constraint problem is found.

The master determines whether or not a solution is found. This is determined from the interval of the candidate FD variable that it receives from its child processors in preparation for distribution. The master and every processor keep tabs on the lowest interval FD variable within its constraint store. The master compares the received variable domain with the value that it contains and chooses the one with smaller lower interval. If the smallest interval turns out to be zero, it implies that there are no variables remaining to be distributed on. Hence, it is concluded that a valid variable assignment has occurred for all variables in the distributed constraint store. The master processor then transitions to the solution found state.

4.5.11 Solution Not Found - State I

This state indicates a failed and unsolved space for the constraint store. The solver reaches this state when there are no alternative paths to explore, all processors can no longer propagate.

When a failure occurs, each processor pops the save constraint store representing the alternative choice to be explored. In the event of a failure when the configuration stack is empty there is no alternative choice point to be explored. It is then concluded that the constraint store will remain in the failed space and unsolved. The master then transitions to the state indicating that no solution can be found.

This concludes the discussion on the implementation of distribution on the multi-processor network.

Chapter 5

Clustering

Clustering can be defined as the grouping of similar sets of objects. It is a collection of objects with similar traits. The term objects and traits are loosely defined. The term object refers to any data sample that needs to be compared for the purpose of grouping Traits are the criterion used for grouping. For instance, the distance between the two objects in the data set can be used as a criterion for grouping. If each data set represents a customer's purchasing trend, then the grouping could place customers with similar buying trends together. This trait is basically a conceptual criteria used for grouping.

In the context of parallel processing and computer architecture, by aggregating similar objects, computations and comparisons can be performed on the groups rather than individual data elements, reducing the computational effort involved in this process. In order to achieve parallel processing, the data set needs to be partitioned and allocated on different computational units and clustering achieves that.

In an embedded environment, a single processor is limited in memory and processing power in the amount of data that it can process. Data partitioning and parallel processing alleviates this bottleneck by distributing the computations across several processors effectively increasing the net memory available to solve the given problem and providing speed-up in processing.

5.1 Need for Clustering

The model of a finite-domain constraint solver contains a single constraint store storing large number of FD variables which are simultaneously accessed by any computational units called propagators. Storing all FD variables in a single globally shared memory location limits the total memory bandwidth available for concurrent execution, there by defeating

the purpose of concurrent execution. To over this memory bottleneck, the finite domain variables are partitioned into smaller groups that are allocated in the distributed memory.

The central goal of clustering in this project is to partition and allocate FD variables on the distributed memory to achieve improvement in processing time by parallel processing. The clustering is performed for a given graph which needs to be modeled using finite-domain constraints. When a given large set of FD variables is partitioned, inter-processor communication occurs at the cluster boundaries due to the data dependency present between nodes in the graph across cluster boundaries. A balance needs to be achieved between the granularity of parallelism and the computations performed [21]. On the one hand, fine-grain parallelism distributes the computations on multiple processors there by increasing the inter-cluster dependency. If the latency of inter-processor communication is high and frequent, then the communication latency effectively increases the total processing time thus defeating the very purpose of parallelizing. On the other hand, decreasing the granularity of parallelism will increase the amount of computation per processor but decreases the frequency of inter-processor communication. To determine the right size of partition a trade-off analysis needs to be performed.

5.2 Output of Clustering

The clustering algorithm implemented in this project automates the data partitioning, configures the processors for distributed-memory access and the process of distribution. The data to be partitioned is presented in the form of a Data Flow Graph (DFG) where every node is associated with an operation whose execution latency is also provided. The graph is parsed and precedence constraints are generated for every edge in the graph. This form of graph representation is a deviation from the generic representation of a constraint graph used for CSP problem [2]. In a constraint graph, nodes model FD variables and edges tell constraint relations between variables. In contrast, the edge in the DFG is restricted to represent only precedence constraints. Since this research used a domain-specific front-end which interfaced to a previously developed tool DFGs were used for clustering. But a more generic clustering algorithm could easily be created based on what is presented here.

Nodes in the DFG are grouped together to form a cluster. The size of each cluster is decided by the number of clusters that are requested by the user at the beginning of this process. The algorithm for clustering is described in the next section.

When all the clusters are formed, configuration information for distributed memory access and for the distribution process is generated. The first configuration information involves the routing tables that map individual clusters onto processors. The other mapping is the FSL link connections emanating from each source processor and terminating in every destination processor. This connectivity information requires knowledge of the hardware specification file used in the hardware instantiation of the MicroBlaze and its associated peripherals and is not currently implemented. It is therefore manually provided. Second, the data sets residing in cluster boundaries have inter-cluster dependant data sets. Shadow variable information corresponding to such data sets are generated as part of the configuration information. Third, global address allocations are automatically generated for each cluster. Each allocation defines the range of contiguous global addresses that are native to a processor in the network of distributed processors. The size of the range is based on the number of variables of each cluster. As mentioned in Chapter 2, the starting address for each processor in the cluster is discrete and is an integral multiple of the cluster-identifier. Finally, configuration information for the broadcast tree used for distribution is also generated by the clustering algorithm. This information communicates parent/child relationships to each cluster for use in a untrue sync.

5.3 Clustering Algorithm

We now describe the clustering algorithm used for partitioning the graph in this research. The algorithm itself is provided in fig 5.1. The graph traversal scheme followed in this algorithm is that of a simple depth-first traversal with some variation. Traversal starts from a root and continues depth-wise adding successor nodes. Successor nodes that share the same parent are also traversed from each node. All untraversed predecessors of every node that is traversed is also performed. This recursive path is continued for every root node in the DFG until all nodes in the graph are covered.

Initialization:

Input: Number of clusters required by user
Form initial list of of root nodes.
handle=Create new cluster
current_cluster_index = *handle*

Main algorithm:

For all root nodes
current_node = node
If current node ≠ visited
Add to current cluster
Mark node as visited
End If
If current cluster ≠ FULL,then
For all children
Add unmarked node to current cluster
Mark nodes as visited
Include all predecessor nodes if not already included
End For
If current cluster=FULL,then
handle=Create new cluster
current_cluster_index = *handle*
End If
For all children
current_node=node
If current cluster ≠ FULL
For all children of *current_node*
(1)*current_node*=node
(2)**if***current*$_n$*ode* ≠ marked
Include node in *current_cluster_index*
(3)Include all predecessor nodes of *current_node*
(4)**If** *current_cluster_index* = FULL,then
handle=Create new cluster
current$_c$*luster*$_i$*ndex*=*handle*
End If
(5)Repeat (1)-(5) recursively until no other node can be traversed.
End For

Fig. 5.1: Clustering algorithm.

5.4 Results of Clustering

We now look at the output of clustering for the algorithm described in fig. 5.1 by examining two sample graphs. One of them is a linear graph containing nine nodes and the other is a representative of the DFG generated for the LU-decomposition algorithm (discussed in more details in Chapter 6). Let us consider the linear DFG shown in fig. 5.2. A request to partition the graph into four clusters based on the clustering algorithm previously discussed results in a partitioning as denoted by the dotted lines in fig. 5.2. The dotted line marks the boundary of a cluster. Each cluster contains a set of nodes and edges; the edges denote constraints. The set of constraints contained in a processor are to be allocated to that processor for solving. Edges that cross over cluster boundaries indicate inter-cluster data dependency requiring inter-processor communication. The mapping of a cluster to a processor is decided by the user after the formation of the clusters and is shown in figs. 5.3 and 5.4. The interpretation of the figures is as follows.

5.4.1 Linear Graph

In fig. 5.5 the inter-connection of the processor network to form the broadcast tree for distribution is shown. The solid lines in this figure indicate the communication link used for the purpose of distribution and may also be used for the communication link during propagation between two processors. Cluster 0 of fig. 5.3 is mapped onto Processor 1 in fig. 5.5. Similarly Cluster 1 onto Processor 0, Cluster 2 onto Processor 2 and Cluster 3 onto Processor 3. This mapping is reflected in fig. 5.3. Observe closely that the path followed by the mapped clusters have communication links that are in line with the links already established by the broadcast tree network in fig. 5.5. Hence, no additional communication links for propagation are required.

The other form of mapping the clusters in a linear graph is reflected in fig. 5.4. Clusters are mapped to similarly numbered processors. For instance Cluster 0 is mapped to Processor 0 and Cluster 3 is mapped to Processor 3. In this case inter-cluster communication between Cluster 0 and Cluster 1 is required for propagation and this is established by the dotted line in fig. 5.4 between Processor 1 and Processor 2.

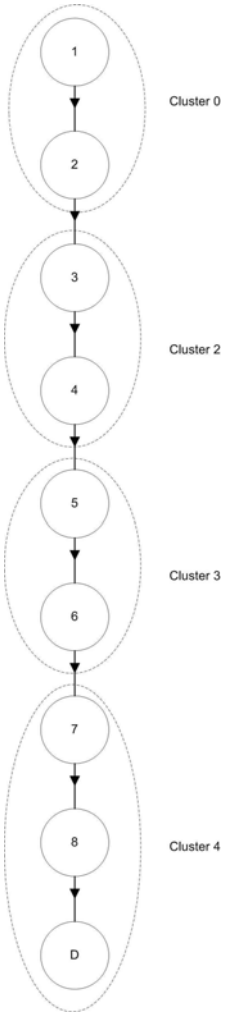

Fig. 5.2: Clusters for linear DFG.

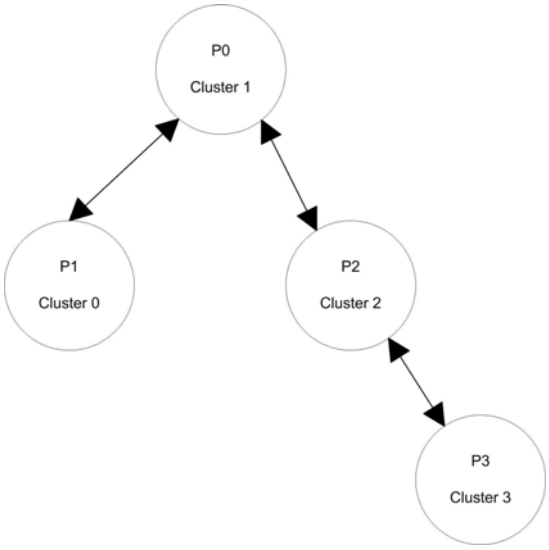

Fig. 5.3: Connections after mapping-1.

The constraints generated in each cluster correspond to the edges contained within each cluster and also the ones originating in that cluster but terminating in a different cluster. For instance the constraints for Cluster 1 are as follows:

$$T(3) + L(3) \leq T(4), \tag{5.1}$$

$$T(4) + L(4) \leq T(5), \tag{5.2}$$

where T indicates the start time of a node and L its latency. Observe that the edge denoted by eq. 5.2 crosses the cluster boundary of Cluster 2. Similarly constraints for other clusters are also generated from the edges. Cluster 3 in fig. 5.5 contains a node named D. This node is the variable denoting the deadline for completing the execution of the entire graph or the time by which the last node in the graph needs to finish its execution. This node is represented as a finite-domain variable with a lower bound and an upper bound indicating the deadline to be met for the required schedule. Since this variable is part of the computations, it is included in the graph to be clustered. The information about this node is provided as an input to the graph parser which populates the graph before invoking the clustering algorithm on that graph.

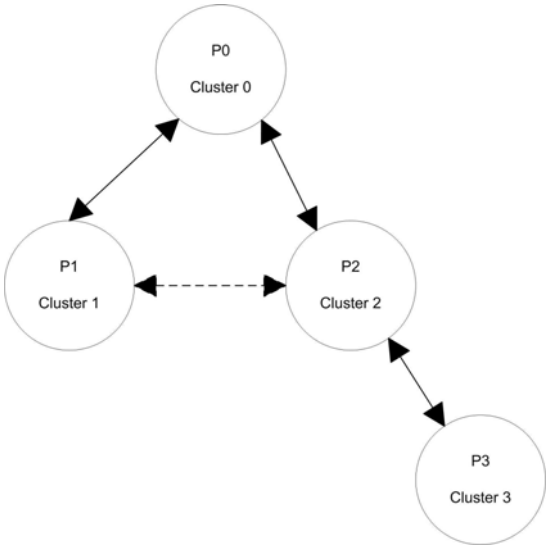

Fig. 5.4: Connections after mapping-2.

5.4.2 Address Allocation

The clusters are mapped onto processors to form a distributed memory architecture. This requires the allocation of dedicated address range to each cluster. The design of the distributed memory architecture was discussed in Chapter 2. It was decided that a uniform address range which is a power of two will be allocated to each cluster which will be a part of the complete global address space. Considering the clusters formed in fig. 5.2, the size of the largest cluster (Cluster 3) is three since it contains three nodes. This size when approximated to the nearest higher power of 2 results in an approximated size of cluster being four.

With four being the uniform size of all clusters, each cluster is allocated an address space beginning with a discrete multiple of the approximated cluster size. Cluster 0 for instance has an address range beginning at 0 and ending at 3. Similarly, Cluster 2's address range begins at:

$$(ClusterID * SizeofCluster) = (2 * 4) = 8,$$

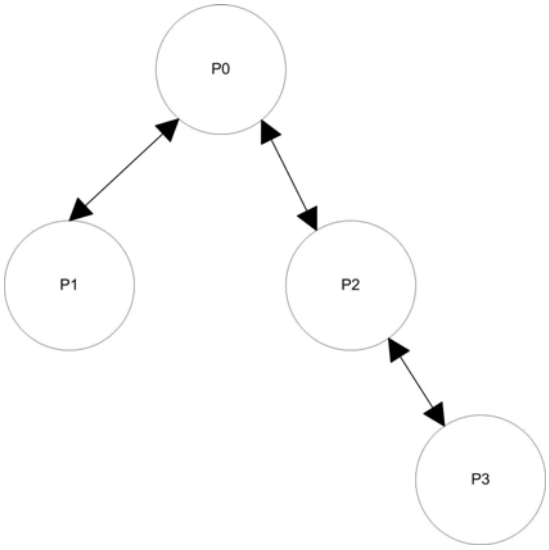

Fig. 5.5: Processor connection before mapping.

and ends at the address given by:

$$8 + ClusterSize - 1 = 11.$$

In this manner all clusters are allocated their own address space from the global pool. Each node within the cluster represents a finite-domain variable and is assigned a unique address from this range in the sequence in which it is encountered during the address assignment step. The address allocations for the nodes in fig. 5.2 are shown in Table 5.1. The first column represents the original node-id and the second column represents the transformed node address. Observe that there is a break in the Node-address numbering for Node-id 2,3 and similarly for Node-id 4, 5 and 6, 7. This is because these node-ids mark the end of one cluster and the beginning of the next cluster. The uniform cluster size is four and since each cluster is not filled to its limit there are gaps/breaks in the translated address.

5.4.3 Sample Graph-2: LU-Decomposition

We previously demonstrated the efficacy and the purpose of the clustering algorithm by

Table 5.1: Address translation.

Node-id	Node-Address
1	0
2	1
3	4
4	5
5	8
6	9
7	12
8	13
9	14

means of simple linear DFG. The robustness of the algorithm is also investigated for another DFG (fig. 5.6) which is a representative of the operations occurring in an LU-decomposition algorithm. The number of clusters requested by the user is three and the various clusters are labeled. After forming these clusters the procedures of mapping, address allocation and constraint assignment are applied to the partitioned graph as mentioned in the case of the linear graph.

5.5 Limitations of the Clustering Algorithm

In this section we discuss the efficiency of the clustering algorithm and its limitations in achieving its goal of load balancing in parallel processing. For the simple linear graph, clustering partitioned the graph into uniformly sized clusters achieving even work load distribution on each processor. In the graph for LU-decomposition, clustering partitioned the graph into unevenly distributed clusters of size 8, 8 and 3 (fig. 5.6). This resulted in uneven distribution of the constraints and did not achieve optimal load balancing. The performance results in the Chapter 6 reflect this point. The clustering algorithm used here was a naive implementation which only demonstrates partitioning of nodes. The algorithm falls short of achieving optimal load balancing. In trying to achieve this trade-off, the amount of processing work load per processor against the communication latency accrued due to inter-processor communication is not taken into account. There is much scope for improvement in this respect. Clustering techniques that consider inter-processor communication costs into account and adjust the granularity of parallelism to achieve effective processor

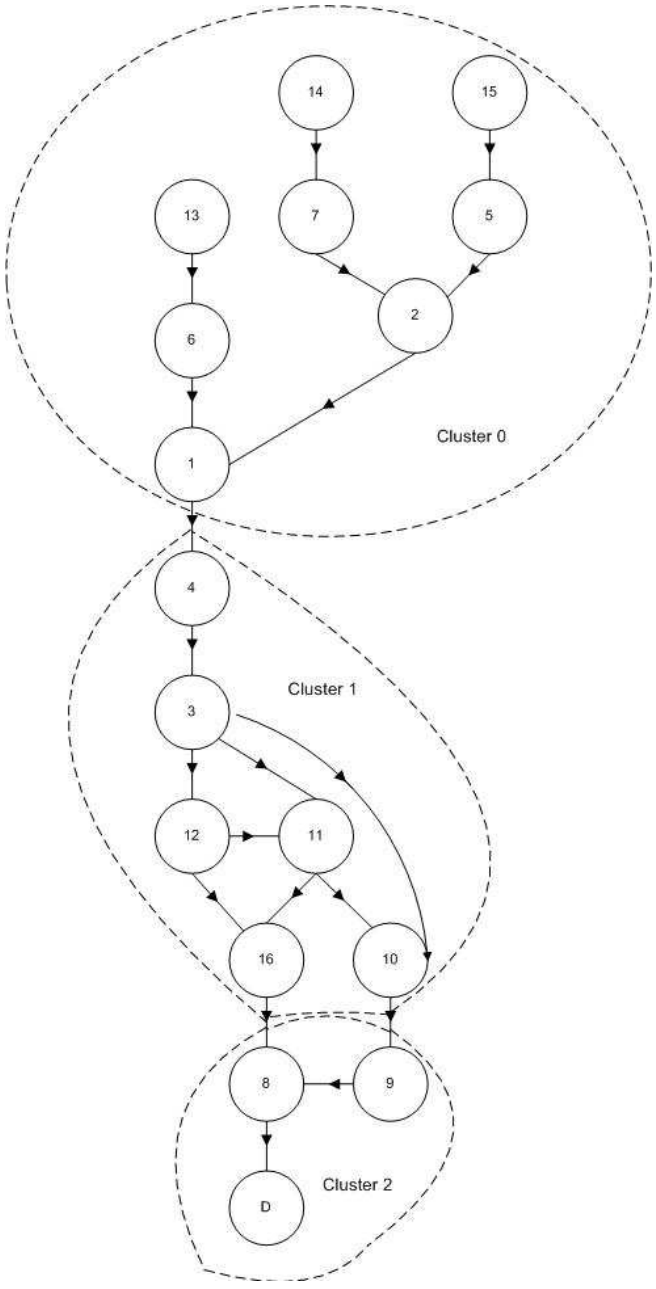

Fig. 5.6: Clusters for LU-DFG.

utilization need to be employed [22] so that the benefit of performance speed-up is not lost in increased communication latency.

Chapter 6

Results and Conclusion

The results stated in this chapter illustrate the effectiveness of the FPGA-based parallel finite-domain constraint solver that was proposed in this research. Measurements from the FD solver implementation executing a class of problems demonstrate the potential performance benefits that can be achieved by parallel processing on a distributed-memory platform in an embedded system. The goals of the experiments presented in this chapter are two fold:

- Feasibility of a parallel implementation of a FD solver on an embedded platform;
- Illustration of significant performance benefits due to parallel processing over that of a single processor implementation.

The metrics used in the experiments were the time to propagate, number of distribution steps, time for convergence to first solution, memory and resource utilization. These metrics highlight the performance and scalability of the implemented design for different applications and problem size. Measurements were performed for multi-processor MicroBlaze network implemented on a Xilinx Virtex-II Pro FPGA that was clocked at $100MHz$.

The performance benefits of using constraint solving techniques for online search are shown by employing this technique to the class of event scheduling problems. Scheduling is a complex problem to solve and by modeling online scheduling of events as a constraint satisfaction problem, the system is adaptable and is able to reschedule dynamic occurrence of events in a changing environment.

The real-world justification for online constraint solving is explained by its need in autonomous mission planning applications. Mission planning applications involve self-guided objects deployed in space that explore and navigate without human intervention. The object could be a vehicle moving in space or a robotic inter-planetary exploration system.

Communication latency between the object in space and the ground station on earth is very large and moreover a reliable communication path cannot be guaranteed all the time.

Such a robotic space exploration system is equipped with equipments and complex controls to sense its surroundings, make measurements and navigate. It is constantly subjected to a dynamically changing environment. For instance, permanent equipment failure could occur, transient faults could be imposed due to radiation and collisions with other objects in space. The system also needs to deal with a redundant component being activated or being deactivated. The scheduler in such a complex system has to respond to such dynamically occurring events at runtime by rescheduling the operations to be performed. Therefore online scheduling/search is required, as justified in Chapter 1.

We consider the FD constraints that are extracted from the task flow graph representation of the application problem. A task flow graph representation inherently encodes the notion of precedence and resource constraints. The edges set precedence in the order of execution of the nodes. Every node in the task flow graph represents an operation that is performed on a resource.

An event graph representing the possible events occurring in an autonomous space mission is illustrated in fig. 6.1 [23]. It is a hypothetical scenario of events for a space application stated by the authors. The nodes in this event graph use one type of resource and there are four instances of each resource type. Each resource type is further denoted by a uniquely shaded node. Each event node has single cycle latency and the deadline for finishing the execution is set to 32 time steps.

As a search problem, the scheduling of the event graph presented in fig. 6.1 is a complex problem to solve considered the explosion in the search space. In Dasu and Philips [23], the authors propose a custom hardware architecture implementing a search heuristic based on iterative repair which is used for dynamically deriving an optimal schedule for events occurring in space missions.

An iterative repair method speculates an initial solution by performing a complete arbitrary variable assignment from the domain of the variables. As long as the solution

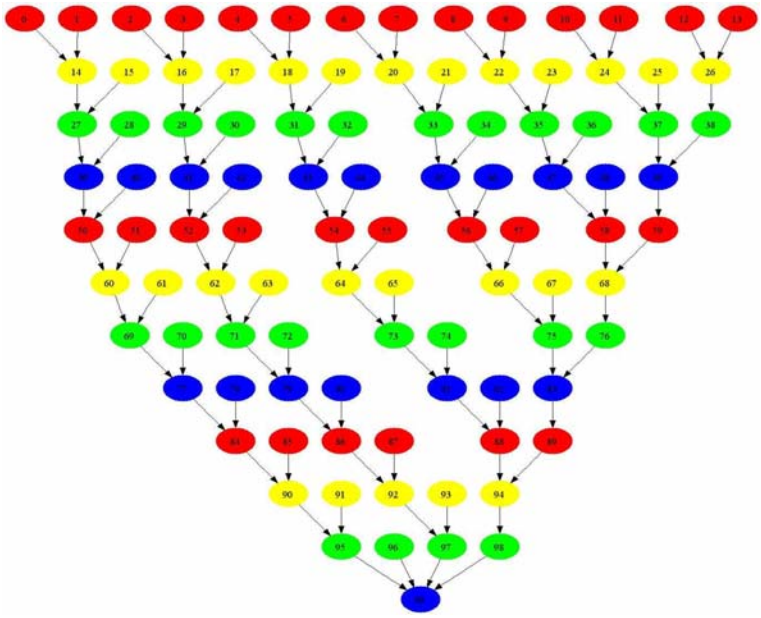

Fig. 6.1: Task flow graph of potential space application.

is invalid, the algorithm iteratively selects a variable and performs value assignment until the constraint violation is removed. Iterative repair technique can also be combined with optimization search techniques such as simulated annealing or hill climbing to minimize an objective function [2].

The event graph denoted in Dasu and Philips [23] is transformed into a task flow graph representation and precedence constraints extracted from the graph are provided as inputs to the parallel implementation of the finite-domain constraint (FDC) solver. We extract a finite domain constraint model that is semantically equivalent to that of the DFG model discussed in previous chapters, consisting of precedence constraints. The finite domain constraint model is extracted from the task flow graph, clustered with the clustering algorithm, and issued to the parallel FDC solver.

We also evaluated other DFGs for the sake of comparison and evaluation of the solver, and that results are presented in the forthcoming charts. In all, three different DFGs were used for benchmarking the feasibility and performance of the design. One was a very simple linear DFG similar to the one shown in fig. 5.2; another DFG represented

Clusters	prop completion(ticks)	# distrb. Steps	First solution(ticks)	Cluster size	Prop speed-up	Distrb. Speed-up
1	110697	1	217084	64	1.00	1.00
2	106058	32	1190349	32/32	1.04	0.18
3	93971	19	568595	22/22/20	1.18	0.38
4	86202	15	429853	16/16/16/16	1.28	0.51

Fig. 6.2: Performance measurements for a linear 64-node linear DFG.

Clusters	prop completion(ticks)	# distrb. Steps	First solution(ticks)	Cluster size	Prop speed-up	Distrb. Speed-up
1	46706	10	228337	17	1.00	1.00
2	40064	9	158334	8/9	1.17	1.44
2	39293	8	154029	9/8	1.19	1.48
3	31740	5	103998	8/6/3	1.47	2.20
4	22367	4	102098	5/4/4/4	2.09	2.24

Fig. 6.3: Performance measurements for 17-node LU-decomposition DFG.

the operations performed in a kernel extracted from an LU-decomposition algorithm as illustrated in fig. 5.6; while the third represents the task flow graph from a possible mission planning problem for an autonomous space application and is illustrated in fig. 6.1. The measurements obtained from executing these applications follows.

6.1 Performance Results

The first set of results shown in figs. 6.2-6.4 compare the time taken to propagate and distribute for the various applications in various configurations. The first column indicates the number of clusters to partition the graph into, where each cluster was allocated to a different MicroBlaze processor. The second column indicates the time taken for completion of the first propagation step in units of clock cycles. The third column states the number of distribution steps required to reach the first solution. The number of nodes contained in each cluster is provided in the next column. The sixth column states the speed-up in propagation obtained by the multi-processor implementation with respect to propagation time for a single processor implementation. The last column provides the speed-up for finding the first solution against the single processor implementation.

The number of backtracks performed during distribution is not reported in the results because the constraint solver does not perform backtracking for any of the problems. This

Clusters	prop completion(ticks)	#distrb. Steps	First solution(ticks)	Cluster size	Prop speed-up	Distrb. Speed-up
1	310209	60	FAILS	50/51	1.00	NA
2	159608	47	2632804	34/34/33	1.94	1.00
3	109971	43	1668505	26/26/26/23	2.82	1.58
4	85914	65	2360598	26/23/26/26	3.61	1.12

Fig. 6.4: Performance measurements for 101-node possible space application DFG.

is because resource constraints from the problem are discarded in determining the solution. Resource constraints further restrict the search space of valid solutions when combined with precedence constraints. When tasks are to be scheduled on limited resources, the number of valid schedules is significantly reduced. The suppression of resource constraints is equivalent to the model assuming an infinite availability of resources. Therefore, the precedence constraint can always be satisfied and every distribution step results in successful propagation.

6.1.1 Propagation

From the results provided in figs. 6.2-6.4, the following observations may be made. An increase in the speed-up for the initial propagation step is noticed as the number of MicroBlaze processors involved to the solve the problem is increased. This measure for initial propagation time is important because the process of propagation and distribution are carried out iteratively until a solution is found. Therefore, reduction in propagation time ultimately translates into reduction in time taken to find the solution. Initial propagation time is particularly measured because maximum number of propagation steps is bound to occur then. This is because all FD variables in the constraint store are initially in the unpruned state. As distribution and propagation progresses, constraint stores are gradually pruned.

The positive speed-up over the single processor implementation for the DFGs tested in these examples appears to indicate that the multi-processor approach to constraint satisfaction taken in this research is both feasible and results in faster computations as opposed

to a single processor implementation. The linear DFG in fig. 5.2, and the one for LU-decomposition in fig. 5.6, showcase only a slight increase in propagation speed-up as the number of processors is increased. It could be said that it is sub-linear speed-up.

On the other hand a near linear speed-up in propagation time is observed for the mission planning DFG as the number of processors increases. This space application DFG by far exhibits more amenability to parallel processing than the other DFGs. This could be attributed to the nature of the graph and manner of partitioning the graph.

The mission planning DFG represented in fig. 6.1 is larger and broader than the other DFGs considered and hence the partitioning scheme works well to achieve very good results due to parallel processing. The graph represented by figs. 5.2 and 5.6 are narrower and longer in comparison. Therefore, partitioning these graphs does not provide as much benefit in parallel processing. This could be attributed to an increase in the ratio inter-processor communication to useful native computation that is performed. Inter-processor communication overhead depends on the level of inter-cluster dependency that occurs at cluster boundaries. While more explanation is necessary, it appears that a broader graph that converges at the bottom when partitioned into clusters tends to offer more paths of concurrent computation to perform than communication. On the other hand a linear DFG containing 64 nodes is not as amenable to parallel processing despite having least number of inter-cluster communication because of the linear dependency imposed by the graph. The graph contains only path of computation. The fact that it is a trivial problem to solve combined with the fact that each cluster is dependent on its successor to converge before it can converge further reduces the parallelism extracted.

This variation in initial propagation time is better summarized through a trend line whose gradient indicates the amount of speed-up. This is shown in fig. 6.5 where the time for propagation measured in clock ticks is plotted against the number of processors used. The solid line indicates the measurements for potential space DFG, the dashed line indicates the LU-decomposition DFG and the dotted line indicates that of a linear DFG.

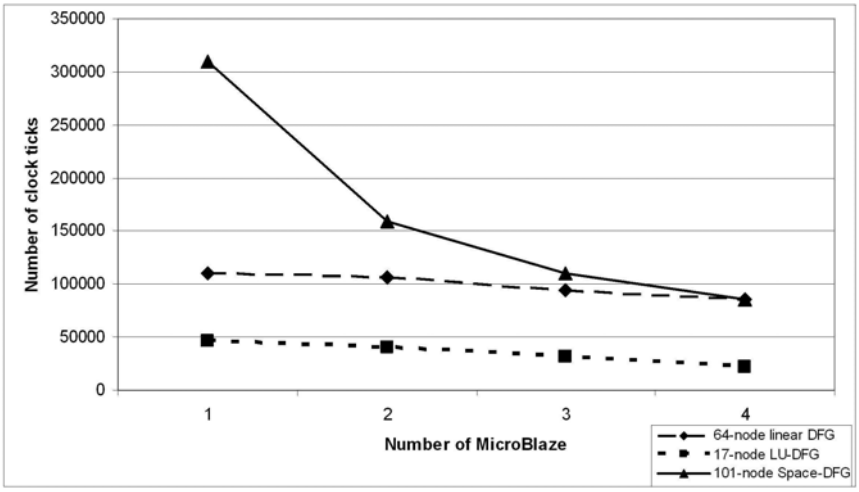

Fig. 6.5: Improvement in propagation time due to parallel processing.

6.1.2 Distribution

We now analyze the performance results of distribution with parallel processing in place. In fig. 6.4 the trial using one cluster in one MicroBlaze fails because of insufficient configuration stack memory. This because there are too many distribution steps, with each choice point containing several variables. This results in insufficient memory in the configuration stack to proceed to further distribution steps. Therefore, the performance measurements for converging to the first solution when distributing is compared against the second trial for two processors.

For distribution, it is observed that there is an anomaly in the number of distribution steps taken for a single processor as opposed to a partitioned graph on multiple processors. The number of distribution steps increases for the partitioned graph leading to increased time in finding the first solution for the multi-processor implementation.

The reason for this seemingly counter intuitive change in distribution steps lies in the implementation of the distribution algorithm. As mentioned in Chapter 4, distribution consists of variable ordering and value ordering. The variable ordering affects the number of distribution steps. For a first-fail distribution when there are multiple variables with equal interval, variable ordering resolves the tie. When there are many variables with the

same lowest interval, the left-most in the ordered list is picked as the distribution point.

For the implementation of the distribution algorithm in this research, an ordered list of variables is not maintained. Instead each processor keeps track of its lowest interval variable when accessing them during each propagation step. So the order in which the variables are encountered completely depends on the order in which the constraints are posted within each cluster.

The results in fig. 6.2 do not show any reduction in time taken to find the solution by partitioning the distribution step because the number of distribution steps increases upon partitioning. However, the results shown in fig. 6.4 for the trial involving four processors breaks this convention by showing that despite containing more distribution steps than the trial involving two processors, a solution can be reached earlier. This is because the performance benefits obtained from faster propagation outweighs that from more distribution steps resulting in an overall reduction in time for solution.

6.2 Memory Consumption

We now analyze the memory consumption of the configuration stack associated with distribution. A single element in the configuration stack essentially represents the size of the constraint store allocated to that processor. The available memory space and maximum configuration stack memory occupied are important parameters to consider because the process of distribution involves the saving of the state of all the FD variables present in the constraint store before branching to a different state. This state saving procedure is required in order to provide provision for backtracking to exploring the search tree. Shortage of available stack memory for distribution will cause distribution step to fail. The available memory therefore restricts the memory available for distribution, and hence the size of the problem that can be executed on this system.

The stack consumption is conveniently represented by means of bar graph in figs. 6.6-6.8 where the darker region represents the percentage of total available MB memory consumed for a configuration. The total memory allocated is $14.37KB$ per MicroBlaze. This graph is

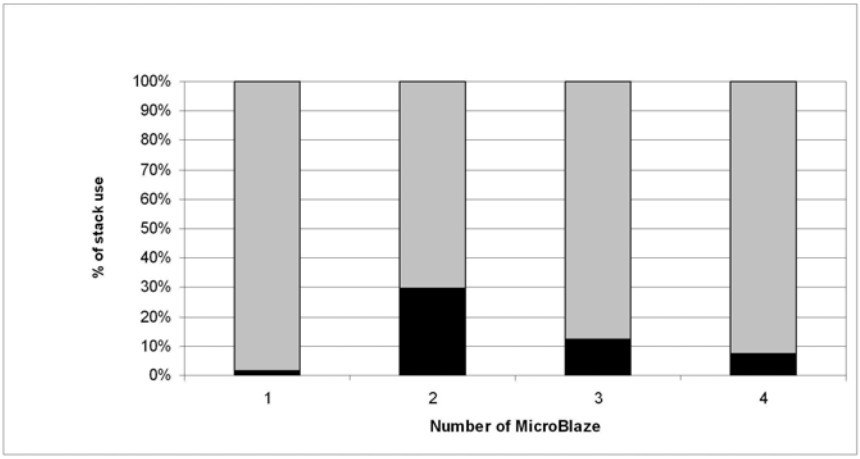

Fig. 6.6: Variation of memory for linear DFG.

a representation of the memory consumption tables shown in figs. 6.9-6.11 for the benchmarked DFGs.

The parameters in the table are self-explanatory. The memory size for one element in the configuration stack and the highest memory usage of the configuration stack are all stated in bytes. The size of an element in the configuration stack refers to the size of constraint store and the associated shadow copies in that processor. In this tabulation, only the largest size of a stack element is considered because the largest size determines the point of failure for a distribution. The fraction of the available memory is mentioned as a percentage. For a distribution step to be successful, memory should be available in all the participating processors. An analysis of the tabulated results follows.

As the number of partitions of the given DFG increases, the memory size of the constraint store per processor decreases and the size of one element in the configuration stack reflects this. It may be observed that the size of an element in the stack decreases as the number of partitions increases. Collectively however, the total amount of available memory for distribution increases with increase in the number of processors.

The benefits of partitioning in the distribution step is particularly noticed in fig. 6.11. For the first trial involving a single processor, it requires 60 distribution steps to reach the solution, but due to unavailability of stack memory the distribution fails. This is also illus-

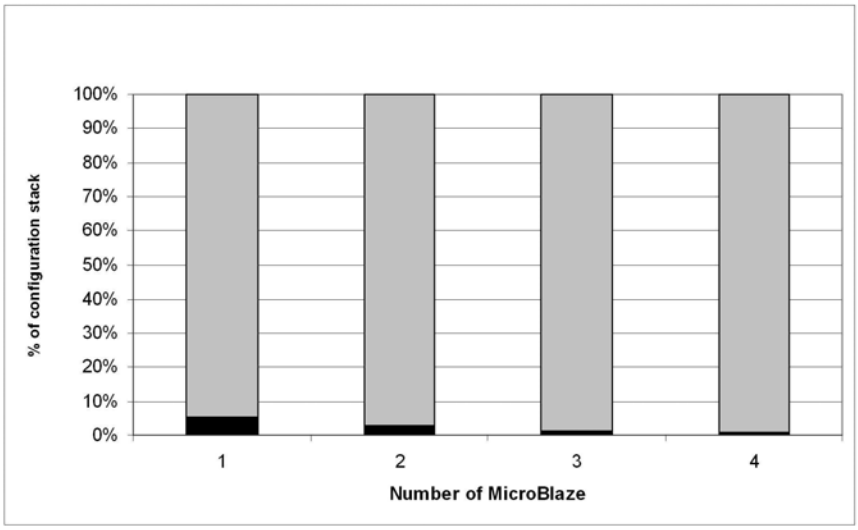

Fig. 6.7: Variation of memory for LU-decomposition DFG.

trated in the bar graph in fig. 6.8 showing, 100% utilization of the available stack memory of $14.3KB$. This problem is remedied when performing a distributed distribution on four processors as shown in the table for the fourth trial. Even though the solution requires 65 distribution steps, memory availability is no longer a constraint because the size of the constraint store per processor is now reduced in comparison to the single processor solution, thus reducing the net stack memory utilization per processor. In summary, partitioning the distribution step on multiple processors increases the available stack memory for the given problem by decreasing the average size of the constraint store per processor.

The dark region of the bar graphs illustrated in figs. 6.6-6.8 indicate the highest stack usage among all the processors. This high water-mark for stack usage is an indication of the maximum depth of the search tree that is explored to find a solution. But the solution to the problem may not necessarily be available in the deepest part of the search tree. It is quite possible to reach an unsatisfactory solution at the deepest level of tree in which case the exploration algorithm should backtrack and continue exploration in a different path. That implies that the number of distribution steps required to find a solution need not be equal to the maximum depth of the search tree traversed. However, in these results, the largest depth of the search tree traversed always corresponds to the number of distribution

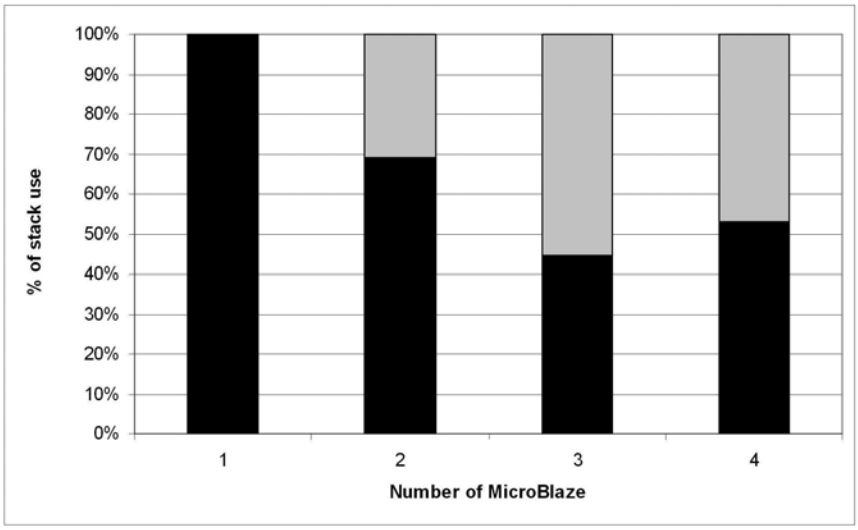

Fig. 6.8: Variation of memory for space application DFG.

Clusters	# distrb. steps	Size of 1 element	Highest stack usage	% of stack memory usage
1	1	264	264	1.79
2	32	136	4352	29.57
3	19	96	1824	12.39
4	15	72	1080	7.34

Fig. 6.9: Configuration stack memory usage for linear DFG.

steps required to reach a solution. Therefore the exploration path of the search tree is always a series of successful propagations for the DFGs explored in this example.

6.3 Scalability

The scalability of the hardware architecture is discussed in this section by analyzing the Post-Place and Route (PAR) results from Xilinx. This is tabulated in fig. 6.12. The slice utilization for the given device was 43%, BRAM being 47% and 8% for multiplier units. This utilization chart shows that the FPGA device can support more hardware units and can accommodate 8 MB units corresponding to the maximum number of MicroBlaze that the OPB bus can accept. The table further shows the break-up of the device utilization for each MB unit which consists of the MicroBlaze and its associated peripherals such as the

Clusters	# distrb. steps	Size of 1 element	Highest stack usage	% of stack memory usage
1	10	76	760	5.16
2	9	44	396	2.69
3	5	40	200	1.36
4	4	28	112	0.76

Fig. 6.10: Configuration stack memory usage for LU-decomposition DFG.

Clusters	# distrb. steps	Size of 1 element	Highest stack usage	% of stack memory usage
1	60	412	14832	167.98
2	47	216	10152	68.99
3	43	152	6536	44.41
4	65	120	7800	53.00

Fig. 6.11: Configuration stack memory usage for space application DFG.

interrupt controller, FSL links and memory.

The PAR report states that the complete design can be clocked at $100.2MHz$. However, the tool only provides a fixed clock rate of $100MHz$.

6.4 Comparison with Oz

The same sets of constraints were also solved using the first-fail heuristic in Oz. For the linear DFG, the number of distribution steps was either 1 or 63 and depended on the variable ordering and on the placement of the variable in the DFG. When the lowest interval variable appeared at the bottom of the DFG and was also the left-most in the ordered list, then a variable assignment to that node followed by propagation resulted in a valid assignment to all its predecessors, and thus a solution. In that case the solution was found in one distribution step. On the other hand, when the ordering was reversed, the lowest interval variable chosen for a value assignment occurred at the root of the linear DFG and subsequent distribution steps occurred on immediate successors. Therefore, a 63-node linear DFG required 63 distribution steps.

A solution was also found for the constraints formulated from the 17-node DFG shown in fig. 5.6. Every node that is labeled in this figure is represented by a labeled FD variable

Resource occupancy of sub-components in a processing unit			
	Slices	**BRAM**	**18x18 Multipliers**
MicroBlaze	1078	0	3
BRAM - 32KB	0	16	0
Interrupt Controller	86	0	0
Memory controller (Instruction & Data)	6	0	0
Memory Bus (Instruction & Data)	2	0	0
One FSL link	22	0	0
Other Components			
Timer	262	0	0
UART	51	0	0
MB OPB	256	0	0
Sub-total resource occupancy			
# of MB – 4	4312	0	12
# of FSL links - 4	264	0	0
# of INTC – 4	344	0	0
Total BRAM – 4	0	64	0
Memory controller + bus – 4	32	0	0
Other peripherals	439	0	0
Total resource occupancy	5960	64	12
Available resources on FGPA	13696	136	136
Device Utilization on XC2VP30	43%	47%	8%

Fig. 6.12: FPGA device utilization.

that is placed in a sorted list of variable labels. The variables in the list are indexed based on the node labeling. The list is essentially an array. The variable ordering depends on the order of sorting (ascending or descending) of the indices.

A descending ordering placed the variables with higher numbered labels to the left of those with lower numbered labels. In this case 10 distribution steps were required because many higher numbered variables occur as root nodes in fig. 5.6. Due to the nature of the DFG, even though distribution started with node-16 at the bottom of the graph, more distribution steps were required to reach the solution. Subsequent distribution steps after node-16 involved node 15, 14 and 13 in order. But these nodes occur at the root of the DFG and do not propagate their constraints to their successors.

When an ascending sort of the variable ordering was followed, variables with lower indices appeared to the left of the list. It may be observed in fig. 5.6, that many lower values variable labels occur in the bottom of the graph. Only five distribution steps were required to reach the solution. The sequence of the labels chosen for distribution was $1, 6, 3, 8$ and 16.

Due to the trivial nature of the problem requiring only propagation steps without any backtracking or constraint violations, performance of the Mozart-Oz solver could not be measured at lower level of granularity.

6.5 Conclusion

This research project achieved many objectives. Through means of a distributed memory architecture on a network of soft-core processors it was demonstrated that the steps of propagation and distribution for a finite-domain constraint solver can be executed in an embedded environment. Concepts for the solver were borrowed from established theories on constraint-satisfaction problems (CSP), but implementations were tailored for the platform and environment in which they were implemented. Even though only one inequality constraint was demonstrated, the implementation of the solver framework is generic enough to allow different propagators embodying various other constraints to be added. In summary the results demonstrate:

- Scalability of the architecture by having provision to extend the architecture to support 8 processors in a single FPGA;
- Propagation and distribution can be parallelized on a multi-processor network;
- Performance benefits in solving the given set of constraints can be obtained due to parallel processing;
- The net available memory for solving the problem also increases due to parallel processing.

6.6 Limitations

A number of steps may be taken to ensure consistency in the results obtained. First, the distribution algorithm can be enhanced such that irrespective of the number of partitions, the order of the variables that are distributed upon remains constant. Second, the clustering algorithm can be improved so that better load-balancing is achieved providing a further

increase in performance. Not only should the number of nodes and constraints allocated per cluster be taken into account, but also other factors such as the number and frequency of inter-cluster communications, also.

While the implemented design offers several contributions, it suffers from some drawbacks. First, the original goal was to implement a generic online constraint solver in an embedded environment for various classes of applications. Online event scheduling was particularly chosen due its popularity and broad scope of applications in embedded computing. The events to be scheduled are fed offline through a DFG to the clustering algorithm which partitions the graph and generates configuration information which is provided as input to the application code for each processor. This application code is compiled and then downloaded into the FPGA board as a bit stream.

It would meet the goals of online scheduling if new events or modifications to existing events can be detected, modeled and added to this embedded, multi-processor constraint solver without having to go through the aforesaid steps offline.

Secondly, the current implementation of the solver only contains a propagator for a simple, specific inequality constraint that models the precedence constraint for a task graph. It lacks in any form of constraint on the set of resources utilized at any point during execution. In that respect the constraint model of the task graph is incomplete and the schedule that is derived from it cannot be accepted as valid since it suffers from any notion of resources [2].

Third, the on-chip memory available to each processor in order to add large number of propagators, model different types of propagators and the depth of the distribution and search tree is limited by the available on-chip BRAM in an FPGA. This memory restriction in the design was set in place in order to reduce memory latency in accessing external RAM.

Lastly, inter-processor communication over the point-point communication links in the network of processors certainly acts as a barrier in performance. As the size of the constraint graph grows, the boundary introduced by the cluster cuts only increases the amount and frequency across cluster-boundaries. This communication overhead pulls the overall

performance down.

6.7 Future Work

A number of limitations to the current approach for parallel constraint solving were mentioned before. In line with those, we outline a number of enhancements and features that may be added to the current implementation.

- **Online constraint solver**: The current offline solving mechanism for multiple processors may be translated into an online constraint solver. This requires making many partitioning and mapping decisions for dynamic load-balancing many of which were previously done offline. Provision to accept or modify events and post or modify constraints also needs to exist. All these decisions need to be done along with the steps of propagation and distribution that are executed on the previously posted constraints [2].

- **Provision for resource constraints**: As explained in Kuchcinski [12], another propagator needs to be created in order to implement resource constraints. The existence of the precedence constraint alone does not suffice since the schedule derived using this approach is not useful.

- **Improved distribution algorithm**: The constraint solver can be designed to have the ability to add domain specific distribution algorithms.

- **Efficient clustering scheme**: The clustering scheme used in this offline constraint solver can be improved so that it takes the overhead of inter-processor communication occurring at cluster boundaries into account and compares that with the fraction of useful constraint solving time as opposed to the time spent in reading or writing information to a different cluster.

- **Feasibility of a memory hierarchy**: The Virtex-II Pro FPGA contained 306 KB of on-chip BRAM [24]. Due to the limited amount of fast, on-chip memory in the

form of BRAM that is available, the feasibility of providing a memory hierarchy in the form of a cache followed by larger RAM can be investigated.

- **Further evaluation**: The performance and capabilities of the implemented constraint solver needs to be evaluated further for various other applications. The improvement in performance also needs to observed with eight processors.

References

[1] M. Lin, J. Xie, H. Guo, and H. Wang, "Solving Qos-driven web service dynamic composition as fuzzy constraint satisfaction," in *Proceedings of the 2005 IEEE International Conference on e-Technology, e-Commerce and e-Service*, 2005.

[2] M. Fromherz, "Constraint-based scheduling," *American Control Conference*, vol. 4, pp. 3231–3244, 2001.

[3] J. J. Blanco and L. Khatib, "Enhancements to the ground processing scheduling system," in *TIME '97: Proceedings of the 4th International Workshop on Temporal Representation and Reasoning*, p. 72, 1997.

[4] M. Zweben and M. Fox, *Intelligent Scheduling*. New York: Morgan Kaufmann, 1994.

[5] M. Henz and T. Müller, "An overview of finite domain constraint programming," in *Proceedings of the Fifth Conference of the Association of Asia-Pacific Operational Research Societies*, 2000.

[6] R. Bartak, "Constraint programming: In pursuit of the holy grail," 1999 [Online]. Available: `citeseer.ist.psu.edu/bartak99constraint.html`.

[7] G. Smolka, "Problem solving with constraints and programming," *ACM Computing Surveys*, vol. 28, Dec. 1996.

[8] C. Schulte, "Oz Explorer: A visual constraint programming tool," in *Proceedings of the 14th International Conference on Logic Programming*, L. Naish, Ed., pp. 286–300. The MIT Press: Cambridge, MA, 1997.

[9] C. Schulte, "Comparing trailing and copying for constraint programming," in *Proceedings of the 1999 international conference on Logic programming*, pp. 275–289. Cambridge, MA: Massachusetts Institute of Technology, 1999.

[10] "Mozart Programming System," [http://www.mozart-oz.org/].

[11] J.-F. Puget, "A C++ implementation of CLP," in *Proceedings of the Second Singapore International Conference on Intelligent Systems*, Singapore, 1994.

[12] K. Kuchcinski, "Constraints-driven design space exploration for distributed embedded systems," *Journal of Systems Architecture*, vol. 47, no. 3-4, pp. 241–261, 2001.

[13] A. Fernández and P. Hill, "A comparative study of eight constraint programming languages over the Boolean and finite domains" [Online]. Available: `citeseer.ist.psu.edu/427353.html`.

[14] B. Williams, M. Ingham, S. Chung, and P. Elliott, "Model-based programming of intelligent embedded systems and robotic space explorers," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 212–237, Jan. 2003.

[15] "ILOG product website ," [www.ilog.com].

[16] "ECLiPSe website," [www.eclipse-clp.org].

[17] K. Kuchcinski, "Java Constraint Programming Library - JaCoP," [http://web.it.kth.se/ cschulte/events/SweConsNet/kuchcinski.pdf].

[18] C. Wolinski and K. Kuchcinski, "A constraints programming approach for fabric cell synthesis," *Digital System Design, 2005. Proceedings, 8th Euromicro Conference on*, pp. 356–363, 30 Aug.-3 Sept. 2005.

[19] K. Kuchcinski and C. Wolinski, "Global approach to assignment and scheduling of complex behaviors based on hcdg and constraint programming," *Journal of Systems Architecture*, vol. 49, no. 12-15, pp. 489–503, 2003.

[20] J. Jaffar, A. E. Santosa, R. H. Yap, and K. Q. Zhu, "Scalable distributed depth-first search with greedy work stealing," in *ICTAI '04: Proceedings of the 16th IEEE International Conference on Tools with Artificial Intelligence (ICTAI'04)*, pp. 98–103. Washington, DC: IEEE Computer Society, 2004.

[21] Dr. Alf Wachsmann, "Concepts of Parallel Computing : Clustering and Shared Memory," [http://www.slac.stanford.edu/ alfw/talks/HPCparallel.pdf].

[22] G. Sih and E. Lee, "Declustering: a new multiprocessor scheduling technique," *Transactions on Parallel and Distributed Systems*, vol. 4, no. 6, pp. 625–637, June 1993.

[23] A. Dasu and J. Phillips, "Deriving FPGA based custom soft-core microprocessors for mission planning algorithms," in *21st Annual AIAA/USU Conference on Small Satellites*, 2007.

[24] Xilinx, "Virtex-II Pro and Virtex-II Pro X Platform FPGAs DS083 (v4.7)," Nov. 5, 2007 [Online]. Available: `www.xilinx.com`.