Scalable FPGA fabric for parallelising 2D-surface trajectory cost calculations

Design and Evaluation of Application and Hardware

Dibyajyoti Jana

Embedded Computing Systems
Submission date: June 2014
Supervisor: Kjetil Svarstad, IET
Co-supervisor: Geir Åge Noven, Kongsberg Defence Systems

Norwegian University of Science and Technology
Department of Electronics and Telecommunications

Project Assignment

Candidate Name:

Dibyajyoti Jana

Assignment Title:

Scalable Fabric for parallelizing 2D Trajectory Cost Calculations – Design and Evaluation of Application and Hardware

Assignment Text:

Many practical applications need to evaluate the quality of many trajectories on a two dimensional map, based on some static or slowly changing cost functions, e.g. topographic elevation, or weather conditions. Such applications would benefit from a hardware accelerator, that can parallelize and perform these cost computations efficiently. The present work proposes a scalable hardware design for such an accelerator, that is amenable to FPGA implementation.

The present work is the continuation of the work done in the previous semester (autumn 2013), where a SystemC model was developed for the proposed design. The tasks in the present semester (spring 2014) are to aim at developing on that model in two directions – one towards a higher level of abstraction involving development of an application to utilize the proposed accelerator and evaluate the effectiveness of the design. The other direction of development was to use the SystemC model as a reference to define detailed hardware and software architecture of the design and implement them. However, the time available would not be sufficient to complete all of these tasks. So, the target would be to finish the higher-level system evaluation and demonstration including an application and incorporate the findings in the detailed design, and to finish a significant part of the implementation task and qualifying it, while drawing up plans and directions for the future work, necessary to finish them.

Assignment Proposer/Co-supervisor:

Geir Åge Noven, Kongsberg Gruppen
ggnoven@gmail.com

Supervisor:

Prof. Kjetil Svarstad, IET, NTNU

Abstract

One way of simplifying two dimensional trajectory cost computation is to partition the 2D domain (i.e. the “map”) into a grid of unit squares, and approximate the cost functions by constants within these sub-domains (called “map segments”), and similarly replace the trajectory by a piece-wise linear approximation, and accumulate the contribution of each map segment by using the constant cost functions of that segment and the length (and possibly direction) of the trajectory there, which are also easily computed because of the piece-wise linear approximation.

In hardware, the map segments can be naturally mapped onto a 2D array of processing nodes connected by a network-on-chip (NoC), where each node contains the cost data for the corresponding map segment, and can compute its local cost-contribution and add that into a data field of a packet, representing a trajectory, and pass it on to a neighbor, so that the packet traverses a path in the NoC that matches the trajectory, it represents. If the packet starts its journey through the network with a properly initialized data-field (unually 0), then after it finishes its journey and the final processing node adds its contribution to the field, it contains the cost of that trajectory. This architecture is scalable, and provides parallelization of computation, but has its draw-backs.

Because the communications between the nodes must occur in all possible directions (to model all possible directions of the trajectory), deadlocks are a real possibility. One way of detecting probable deadlocks is by detecting no progress within a timeout interval. They can then be resolved by dropping a waiting packet. But it is important to communicate the packet dropping to the external application. An auxiliary low band-width NoC, called the injection-ejection network (IENW), is planned to be used for this purpose, along with the main network, called computation network (CNW). IENW is also designed to be used to carry the packets into the correct start-point and carry out from the end-points in the processing array, reducing the CNW loading. Another problem is that the size of the hardware processing array is now connected to the map divisions, which makes reuse of hardware difficult. It may also be hard for applications to exploit the hardware optimally when it is too highly parallel, because then the application will have to produce packets at a high through-put. These problems are solved by letting more than one map segment be mapped onto the same processing node, using a structured approach introduced in Section 1.3.1.

In the previous semester a SystemC design was developed to model this hardware accelerator. In the present semester, a high level C model incorporating an external application and a high level model of the accelerator was developed to study its performance at the highest possible level in order to demonstrate the effectiveness of the design as well as to provide design guidelines for application development, e.g. how to ensure the best utilization of the hardware from the application perspective, how to accommodate the property of packet-dropping in the accelerator, etc. This activity successfully demonstrates the existence of practical applications that can benefit from this design, thereby demonstrating its utility.

In the present semester, a detailed micro-architecture of the communication infrastructure involving the CNW and IENW was developed and implemented in Verilog RTL. This was used for synthesis and timing, targeting a Xilinx Virtex7 FPGA. The results showed that a practical processing array of size 8x8 processing nodes can be comfortably accommodated, at a clock speed of about 245 MHz. These findings provide another level of confirmation of the feasibility of the design. The accelerator would also contain processors and the software running on these processors in order to implement the cost computation algorithm, packet routing, etc. These could not be implemented due to lack of time, but some guidelines for their development have been worked out. During the synthesis, the processors were replaced by a standard Microblaze micro-controller system for area estimation [15], assuming that they would have similar area. Thus the feasibility and utility of the design have been convincingly demonstrated, and its development has been placed on a clearly defined track.

Preface

This thesis is submitted to the Norwegian University of Science and Technology in partial fulfillment of the requirements for the European Masters in "Embedded Computer Systems" (EMECS) degree. This work has been executed during spring semester 2014 at the Department of Electronics and Telecommunication, NTNU, Trondheim, under the supervision of Prof. Kjetil Svarstad, and joint-supervision of Geir Åge Noven, Kongsberg Gruppen.

Acknowledgement

I would like to thank Geir Åge Noven for his problem proposal and suggestions about the basic system architecture. I would like to thank both Prof. Kjetil Svarstad and Geir Åge Noven for their help, reviews and feed-backs, provided to me continually during the entire period of this activity involving literature survey, design and implementation. I would also like to extend special thanks to Tore Barlindhaug for his kind help with the tool licenses and computation resources necessary for synthesis and simulation of the designed hardware.

Table of Contents

1. Introduction and Motivation......1
 1.1 Problem Description......1
 1.2 Proposed Algorithm......2
 1.3 Proposed System Design......3
  1.3.1 Map Folding......5
  1.3.2 Processing Nodes......6
 1.4 Methodology......6
 1.5 Present Contributions......7
2. Previous Work......9
 2.1 Communication Infrastructure......9
 2.2 Processing Node Hardware......11
 2.3 Communication Protocols......12
 2.4 Modeling and Simulation......13
 2.5 Conclusion......14
3. Application: Theory and Research......15
 3.1 Motion and Trajectory Planning......15
 3.2 Trajectory Optimization Problems......16
 3.3 Parallelization of Optimization Algorithms......18
  3.3.1 Genetic Algorithm......18
  3.3.2 Simulated Annealing......19
 3.4 Different Implementation Options......21
 3.5 Conclusion......21
4. Application: Design......22
 4.1 Hardware Modeling......22
 4.2 Application Implementing Simulated Annealing......23
 4.3 Application Implementing Genetic Algorithm......24
  4.3.1 Algorithm Description......24
  4.3.2 Application Design......26
  4.3.3 Potential Simulation Deadlock and Resolution......29
  4.3.4 Hardware Modeling – Packet Dropping Rate......30
  4.3.5 HW and SW Modeling – Dispatch Rate Adaptation......31
 4.4 Conclusion......32
5. Application: Simulation and Evaluation......33
 5.1 Comparison of SA and GA......33
 5.2 SystemC Simulation......33
 5.3 Influence of Packet-dropping......36
 5.4 Influence of Dispatch Rate Adaptation......39
 5.5 Conclusion......41
6. Hardware: Design – Micro-architecture......42
 6.1 CNW Node Architecture......42
  6.1.1 Memory Mapping......43
  6.1.2 CNW Bus Protocol......44
  6.1.3 Internal Bus Protocol......46
  6.1.4 FIFO-Controller......47
  6.1.5 Send-list......50
  6.1.6 Send-module......51
  6.1.7 Data Consistency of Shared Resources......54
 6.2 IENW Architecture......55

6.2.1 IENW Node Architecture........56
6.2.2 Deadlock Avoidance........57
6.2.3 IENW Protocol........59
6.2.4 IENW Control Logic........60
6.3 Processing Node Software........63
6.3.1 Virtual Map Records........64
6.4 Conclusion........64
7. Hardware: Verification........65
7.1 Design Requirements and Covering Test-Cases........65
7.2 Details of Test-Cases........67
7.3 Conclusion........72
8. Hardware: Synthesis and Evaluation........73
8.1 Performance Bottleneck and Resolution........73
8.2 Hardware Resource Usage (Single Node)........75
8.3 Hardware Evaluation: Multi-node........76
8.4 Conclusion........78
9. Discussion and Conclusion........79
9.1 Accelerator Model........79
9.2 Application........79
9.3 Implementation........80
9.3.1 Model-Implementation Differences........80
9.4 Conclusions........81
10. Further Work........83
Reference........86
Appendix A. Guide to Codes........88
A.1 High Level C Model........88
A.2 Verilog RTL Design........92
A.3 SystemC Model........93
Appendix B. Selected Simulation Data........94

List of Terminology

2D	Two Dimensional
CNW	Computation Network
FTR	Flight Trajectory Record – a packet that contains the details of a trajectory, and is passed around in the NoC fabric to calculate the trajectory cost.
GA	Genetic Algorithm
GPU	Graphics Processing Unit
HW	Hardware
IENW	Injection-extraction Network
Map	The 2D surface on which the trajectories are defined
Map Segment	Each element of the grid of rectangular subareas with approximately constant cost-function, in which a map is divided for the simplicity of calculation.
PN	Processing node, i.e. a node of the NoC fabric where all computations are performed – both for the communication and cost calculation
RADAR	RAdio Detection And Ranging
SA	Simulated Annealing
SAM	Surface-to-Air Missile
SW	Software
Terrain Data	The approximate cost function values associated with a map segment
Trajectory	A directed path on a 2D surface
UAV	Unmanned Aerial Vehicle
VMR	Virtual Map Record – a packet that is used to configure the terrain data in the system

Chapter 1

Introduction and Motivation

Many practical problems involve calculating costs of trajectories on a two-dimensional (2-D) map. The map may be a usual geographical map, or some other problem may be transformed into an equivalent form for easier visualization and solution. The cost may also be of different kinds. The simplest of them is the length of the trajectory. However, many more complicated costs may be defined, e.g. the distance traversed along a trajectory on a geographical terrain (with uneven topographic elevations), or to accumulate all positive differences in altitude in the direction of motion along the trajectory and to discard the negative ones. This is generally difficult to solve analytically, but to a cyclist, this would be a measurement of the effort she would require to spend to traverse this trajectory. Costs may also be a linear combination of different factors, e.g. weather situations at different locations on the map, or threat from enemy positions in case of a military application, etc. It is, however, often possible to reduce all such contributing factors into one scalar (e.g. weather condition) and one vector (e.g. terrain gradient) component, as long as the operations performed on them are linearly combinable. The present work proposes a way to perform such trajectory cost computations efficiently, and develops a hardware that implements this approach. This hardware then may be used as a hardware accelerator for any application that needs to perform such 2-D trajectory cost computations efficiently.

1.1 Problem Description

Trajectory costs, as described above, can normally be represented as line integrals (or sums of line integrals) of some cost function along piecewise differentiable trajectories. As mentioned above, one simple example would be to calculate the length, L, of a trajectory, T, on a 2D map. This can clearly be evaluated as the line-integral, $L=\int_T 1.dt$, where the cost-function is a constant scalar, 1. Another such scalar cost function, f(x,y) defined for Cartesian co-ordinates, could be the inverse of the velocity attained by an all-terrain vehicle (ATV) at different locations on a geographic plane, depending on the nature of the soil there, e.g. whether it is firm, loose, sandy, muddy, etc. In such a case, the line-integral, $\int_T f(x,y)dt$ evaluates the time taken by the vehicle to traverse the trajectory, and this will be its cost.

However, it is possible to also have vector cost functions. Let $\vec{\nabla}(x,y)$ be defined to be the gradient at the co-ordinates (x,y) on a topographical map of a geographical region. The line-integral, $\Delta_h=\int_T \vec{\nabla}(x,y).d\vec{t}$ in this case computes the difference in elevation of the start and end points of the trajectory, and may be defined as the cost of this trajectory, though there may exist easier ways of computing this quantity than by evaluating a line integral.

Some other trajectory costs may be harder to express as line integrals, e.g. the problem, mentioned at the beginning, of estimating a cyclist's effort to traverse a trajectory in a topographically uneven area. This can be evaluated only as a summation of line-integrals along segments of the trajectory. In order to solve this problem, the trajectory has to be divided into two sets of segments, say S_u, the uphill set and S_d, the downhill set, where $S_u = \{T_1, T_2, \ldots, T_n\}$, such that at every point on each of these segments, the condition $(\vec{\nabla}.\vec{e_T})>0$, where $\vec{e_T}$ is the unit tangent vector along the direction of the trajectory at that point (assuming a piecewise differentiable trajectory), is satisfied. All other segments of the trajectory make up the set S_d. Then, the estimate of the cyclist's effort may

be represented as $E_c=\sum_{p=1}^{n}\int_{T_p}\vec{\nabla}(x,y).d\vec{t}$. The gradient may still be regarded as the cost-function, though with constrained applicability.

Another cost, relatively difficult to formulate, is the distance traversed along a trajectory on a geographical terrain with uneven topographical elevations. The x-component of the gradient, say ∇_x is the rate of change of elevation at a location in the x direction. If we define $g_x=\sqrt{1+\nabla_x^2}$ it provides the local ratio of actual distance traversed on the topographical map to the displacement along the x-direction. Similarly, the y-component of gradient, ∇_y may be used to define $g_y=\sqrt{1+\nabla_y^2}$. Now, defining a vector function on the 2D map, $\vec{g}(x,y)=g_x.\boldsymbol{i}+g_y.\boldsymbol{j}$, where ***i*** and ***j*** are respectively the unit vectors along the x and y directions, the line-integral, $d=\int_T\vec{g}(x,y).d\vec{t}$ yields the actual distance traversed on the local geographical topography by following a trajectory, T, directed only towards positive x and y directions at every point (if it is differentiable at that point) on it. Similar expressions can be formulated for other orientations of the trajectory. Just like in the case of the cyclist's effort, a general trajectory can then be divided into segments, each of which lends itself to application of one of these formulae, and the results of these line-integrals can then be summed up to derive the total distance traversed on the geographical terrain following the trajectory.

It is possible to also have costs which do not involve line-integrals, but are defined by extrema of a cost-function along it. Using the previous examples of topographic maps, it may be the highest elevation of a point on the trajectory, or the steepest gradient along it. It is also possible to define the overall cost of a trajectory to be a (linear) combination of the basic costs of the types explained above. A road-builder, for example, would define the overall cost of the proposed lay-out of a road (represented as a trajectory for our purpose) depending on its length, its maximum steepness, maximum curvature, how long a vehicle would take to traverse it, etc.

As this discussion shows, the computation of trajectory costs may often be quite complicated, even though required in many applications. Hence it was felt that an efficient way to perform this task would be quite beneficial. Therefore, the aim was to define an efficient algorithm for estimating such trajectory costs, and develop a hardware system that can implement it efficiently at a high through-put, which would require parallelization. In order to enable rapid prototyping and deployment, an FPGA implementation is expected to be ideal. In order to exploit the full flexibility offered by FPGA's, the design is required to be scalable, so that it can be easily adapted to changing system requirements, whether involving change in complexity of the problem, or the throughput requirements.

1.2 Proposed Algorithm

As explained above, the cost of a trajectory can often be modeled as the line integral of a cost-function, or a linear combination of the line integrals of multiple cost-functions. In order to estimate the line-integral, the map is proposed to be divided up into an SxS unit square grid, and the cost-function(s) is/are assumed to be constant over each square, called a “map segment”. The trajectory is similarly represented by a piece-wise linear approximation defined by the end-points (termed “via-points”) of each of these linear segments in their proper order, as shown on Figure 1. The contribution of each map segment to the overall trajectory cost can then be computed as the product of the length of the part of the trajectory in that map segment and the value of the cost-function(s) (evaluated along the direction of the trajectory in case of a vector cost-function) in that map segment. As explained above, the cost-functions are assumed to be constant over the whole map segment, but in case of a vector cost-function, its value along the direction of the trajectory may

change if a via-point is encountered inside a map segment, and the trajectory direction changes at that point. This has also to be taken care of during the cost computation. The total cost of a trajectory can then be estimated by summing up the cost contributions of all map segments, that the trajectory passes through.

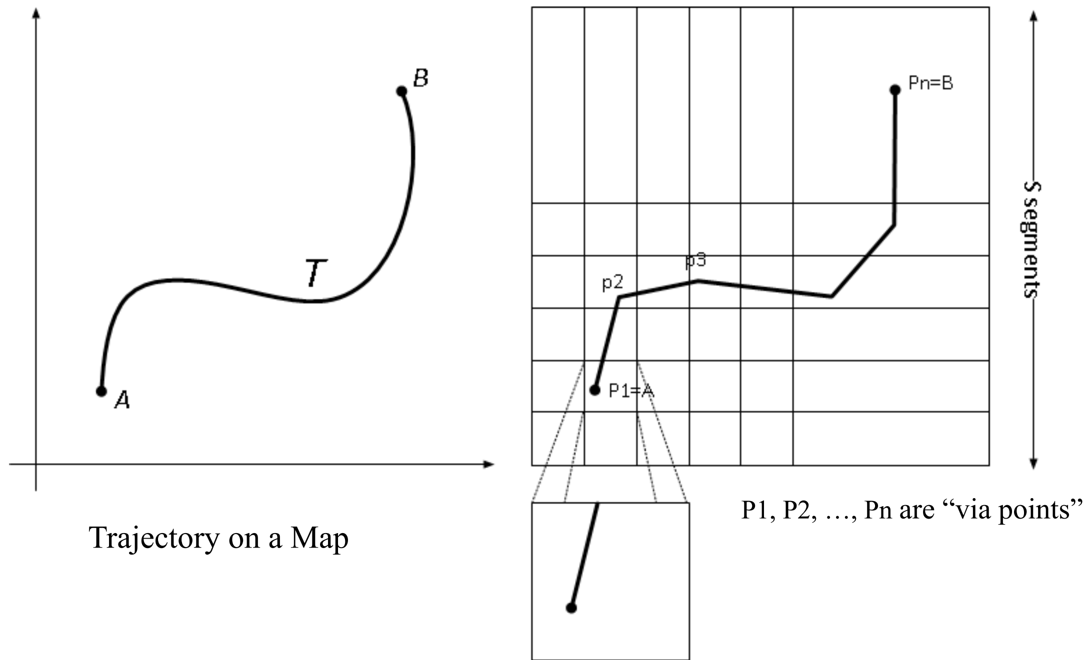


Figure 1: "Map" divided into segments and piecewise linear approximation of trajectory

An analogous approach can be utilized to evaluate the costs which are defined as extrema along the trajectory. In this case, instead of adding up the contributions from the various map segments that the trajectory passes through, they can be compared and the require minimum or maximum can be accepted as the cost. Because the extremum-type and the line-integral-type problems both have naturally analogous solutions, and because most of the useful costs are expected to be of the line-integral type, the discussion will henceforth primarily refer only to that. However, the analogous operations for the extremum-type cost can always be derived easily from it.

This algorithm is also effective at tackling the cost computations that involve segmentation of the trajectory, as was required for the last to examples in Section 1.1, because trajectory-segmentation is used as one of its basic steps.

1.3 Proposed System Design

In order to perform the above-mentioned computations efficiently, one natural solution envisioned was to use an SxS 2-D array of processing nodes connected with each other through a network-on-chip (NoC) of mesh topology, where the 2-D processing array maps directly onto the 2-D "map segment" grid described above with 1-to-1 mapping between the processing arrays and the map segments, each processing node containing the (locally constant approximate) values of the cost-function(s) in the corresponding map segment, and capable of computing the contribution of the local map segment to the total cost of a trajectory. When computing the cost of a trajectory, the trajectory can be defined as an ordered list of via-points, as described above, encased in a packet, which can be passed inside the NoC from processing node to processing node along a path that maps to the trajectory concerned, while each processing node encountered on the path adds its cost

contribution to a data field in the packet. The routing decision at each processing node can be taken based on the via-list. The data field in the packet is initialized to 0 before it starts its journey in the NoC, and it contains the computed cost of the trajectory when the packet reaches its destination processing node, and the destination has added its cost contribution to the said data field.

Being a NoC-centric design, the system is easily scalable (i.e. more nodes can be added easily). Also, it is easy to see that all communications are local between neighboring processing nodes, as the trajectories are continuous. Still it is a problem that the processing array size is determined by the size and the granularity of division of the target map, thus limiting the scope of usage of the hardware for widely varying maps. This design may also struggle to maintain good hardware loading. The loading of a processing node is defined as the proportion of time it is "active", i.e. it spends performing useful calculations on a packet, e.g. routing or cost computation, and the over-all hardware loading is defined as the average loading of all the processing nodes. Defining P_n to be the n-th processing node, and the total number of nodes in the processing array to be N, let us define:

$$\begin{aligned} A_n(t) &= 0 \quad \textit{if } P_n \textit{ is inactive at time } t \\ &= 1 \quad \textit{if } P_n \textit{ is active at time } t \end{aligned}$$

From this, the hardware loading (HL), as defined above, can be formulated as:

$$HL = \frac{1}{N}\sum_{n=0}^{N}\left(\frac{1}{T}\int_{0}^{T} A_n(t).\,dt\right)$$

where T = the period of use of the hardware. This expression may be alternatively written as:

$$HL = \frac{1}{T}\int_{0}^{T}\left(\frac{1}{N}\sum_{n=0}^{N} A_n(t)\right).\,dt$$

which can be interpreted as the average proportion of the "active" processing nodes in the processing array.

In order to make a rough estimation of the HL, let us assume that every packet under processing inside the processing array is always active, i.e. being processed by one node or another, and that each processing node can process only one packet at a time. Then, HL is also the ratio of the average number of active packets to total number of processing nodes in the array. If, the average latency of a packet inside the hardware (i.e. the time required to complete its cost computation) be T_l, and one new packet is assumed to be launched for computation at an interval of T_t, then in the first T_l interval of the execution, the hardware receives (T_l/T_t) packets, but likely does not produce any output packet. But, after that in every T_t time interval it receives a new packet and produces one new output on average, thus reaching a stochastic steady-state. In this steady-state, then, the number of active packets inside the hardware can be estimated as (T_l/T_t). If the map is divided into an SxS grid, and accordingly a processing array of size SxS is used, then $N=S^2$, and $HL = T_l/(S^2.T_t)$.

Assuming that the map area is selected near-optimally, i.e. the map-area chosen contains very little area beyond what is required to represent the trajectories, the smallest length of a trajectory would be close to S. On the other hand, the maximum length possible for a straight-line trajectory is √2S. So, if the trajectories under consideration are relatively well-behaved, i.e. not with too many bends, their length will also be in the order of S (say, below 3S or 4S). Thus, it is clear that the average trajectory length is of the order of S, and the packet computation latency is expected to be proportional to the trajectory length. Hence, it is reasonable to assume that $T_l = \alpha S$, where α is a constant. This may be used to rewrite $HL = \alpha/(S.T_t)$. This means that with increasingly finer division of the map (i.e. increasing S), in order to maintain constant HL, the T_t has to be reduced proportionately. It also means, of course, adding more processing nodes into the hardware, which is a problem in itself. Even the required reduction of T_t is expected to be a problem, as the application

that would use the proposed hardware for accelerating its trajectory cost calculation, would ideally prefer to not have to change its own throughput with the changes in map division granularity, which may be required for proper accuracy of the cost computation.

The afore-mentioned two problems (modifying hardware and modifying application throughput) associated with the proposed hardware structure can be solved by allowing multiple map segments to be mapped onto the same processing node. An intelligent and simple way of achieving this, while maintaining the desirable quality of keeping all communications local, is by introducing the concept of logically folding the map, as explained below.

1.3.1 Map Folding

The idea is to define a "logical folding" of the map along each axis as shown on Figure 2 when defining the mapping of map segments to the processing nodes. In this scheme, all the map segments that end up coinciding after the folding, are mapped on to the same processing node. For example, if there are 12 map segments (0...11) along the X-axis, and if only 2 folding layers are used along this axis, then the map segments (0...5) coincide on (11...6) respectively, and the processing array requires only 6 nodes along the X-axis, where node 0 processes map segments 0 and 11, node 1 processes map segments 1 and 10, etc. The same map may be folded along the X-axis using 3 folding layers, and then, map segments (0...3), (7...4) and (8...11) coincide respectively. It requires a processing array of 4 nodes along the X-axis, where node 0 processes map segments 0, 7 and 8; node 1 processes map segments 1, 6 and 9, and so on. The same idea is applicable also along the Y-axis. Thus, a 12x12 map after folding with 3x3 layers (i.e. 3 layers due to folding along X-axis and 3 layers due to folding along Y-axis, yielding total 9 layers) will have map segments (0,0), (0,7), (0,8), (7,0), (7,7), (7,8), (8,0), (8,7) and (8,8) coinciding and being processed by the processing node (0,0) in a processing array of size 4x4. The folding does not change the logical representation of the map, map data or the trajectories. It only modifies the mapping of the map segments onto the processing nodes, the mapping thus becoming many-to-one.

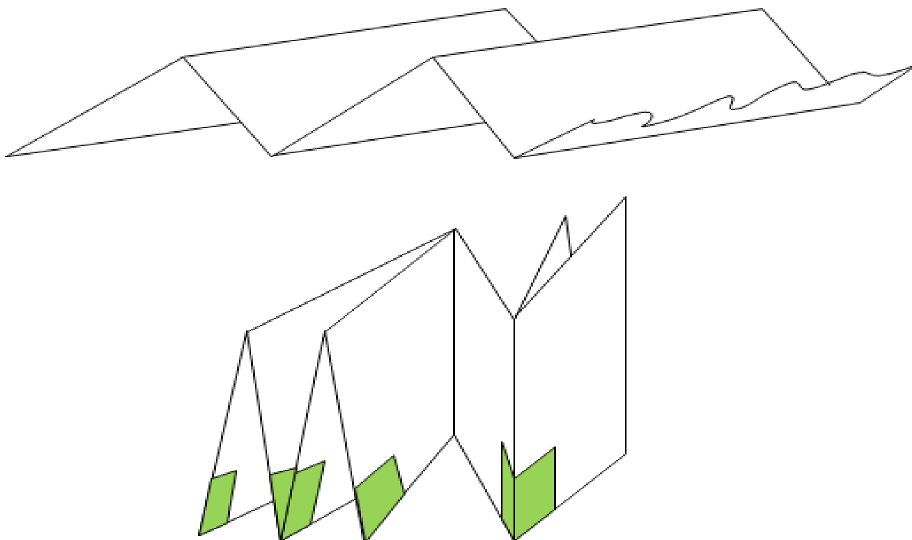


Figure 2: Folding the map. Top: Folding along one axis only. Bottom: Folding along both axes – the green map segments all map to the same processing node.

This mapping mechanism ensures that different map resolutions and map sizes may be computed using the same processing array. It also ensures that the map segments which are neighbors on the map, are mapped either onto the same or neighboring processing nodes, preserving the property that all communications in the hardware are local. This also ensures that increasing map resolution (i.e.

allowing finer division) would increase the packet latency in the hardware, allowing the application to increase the interval between the packets it launches to the hardware in order to maintain the same level of loading. This relaxes the timing requirements on the application and should be easier to implement. If the application needs to be made oblivious of the resolution change, that may also be achieved by using a hardware “packet launching” mechanism, that can take care of this, instead of the application. But this has to be decided on the basis of specific system requirements on a case by case basis, and is thus not considered further in this work.

1.3.2 Processing Nodes

As explained before, the NoC fabric connects the processing nodes, that perform at least two important tasks – the cost computation, as well as routing calculation, determining in which of the four possible directions each packet has to be routed from the present node. The routing calculation algorithm is expected to be fixed, allowing a packet to follow a path in the processing array, that traces the trajectory in the map specified by the packet's via-list. Thus it can possibly be implemented as a hardware component. However, the cost computation must be reconfigurable, with many different kinds of possible operations, including but not limited to the ones explained in the Section 1.1. In order to ensure the best reconfigurability, the cost computation algorithm is best implemented as a software running on a small microprocessor. These calculations also require the length, and possibly the direction of the trajectory in the present map segment. These data can be computed as byproducts of the routing calculation. Hence, the cost computation and the routing computation are expected to be quite closely related. So, it is decided, as an initial design decision to implement both of them in software running in the present processing node. Therefore, every processing node may be thought of consisting of a communication node, that handles the communication requirements of the NoC, and a microprocessor, which performs the routing and cost computations.

1.4 Methodology

The outline of the basic concepts described so far was available as a part of the project proposal, including the key algorithmic concept of map folding. The ultimate aim of the work was to study the proposed hardware solution in further detail, and implement it. The work would have a hardware design in RTL, and possibly a prototype in FPGA as its final result. Additionally, it would be useful to develop an example application, that can use this hardware as a hardware accelerator to accelerate its trajectory cost computations while exploiting the hardware optimally, in order to evaluate and demonstrate the efficacy of the design.

In order to smoothen the design of the “hardware accelerator”, i.e. the hardware processing node array connected through a NoC as described above, the hardware-software co-design approach was followed, in which the accelerator system was first modeled in SystemC including both the hardware structure as well as the software to be run on the microprocessor in each processing node. The data gathered from the simulations of this model would be used in two ways. Firstly, an even higher level design would be developed in C-language that models a system comprising an external application and the hardware accelerator. This design can be simulated to understand the mutual influences between the application and the accelerator designs, whereby constraints can be formulated for future application designs and requirements can be generated for the hardware design. Secondly, the micro-architecture of the hardware accelerator, consisting of the communication infrastructure (NoC and communication nodes) and the algorithm to compute both cost and routing, implemented as a software running on a processor, as described in Section 1.3.2, would be worked out, and both hardware and the software components of it would be designed and implemented. Each of these tasks again involve further partitioning and micro-architecture

definition, implementation, verification and performance evaluation. The designs are different levels of abstraction are also mutually interdependent. Simulations of the higher level models help evaluate architecture level decisions, while data received from the evaluation of the lower level models help to fine-tune the higher-level models. Thus, the whole design involves intimated feed-back between the three levels of abstraction used (high-level C model, SystemC model, HW/SW implementation), and may potentially involve some iterations because of that. The whole scope of the work can be described in a graphical format as on Figure 3.

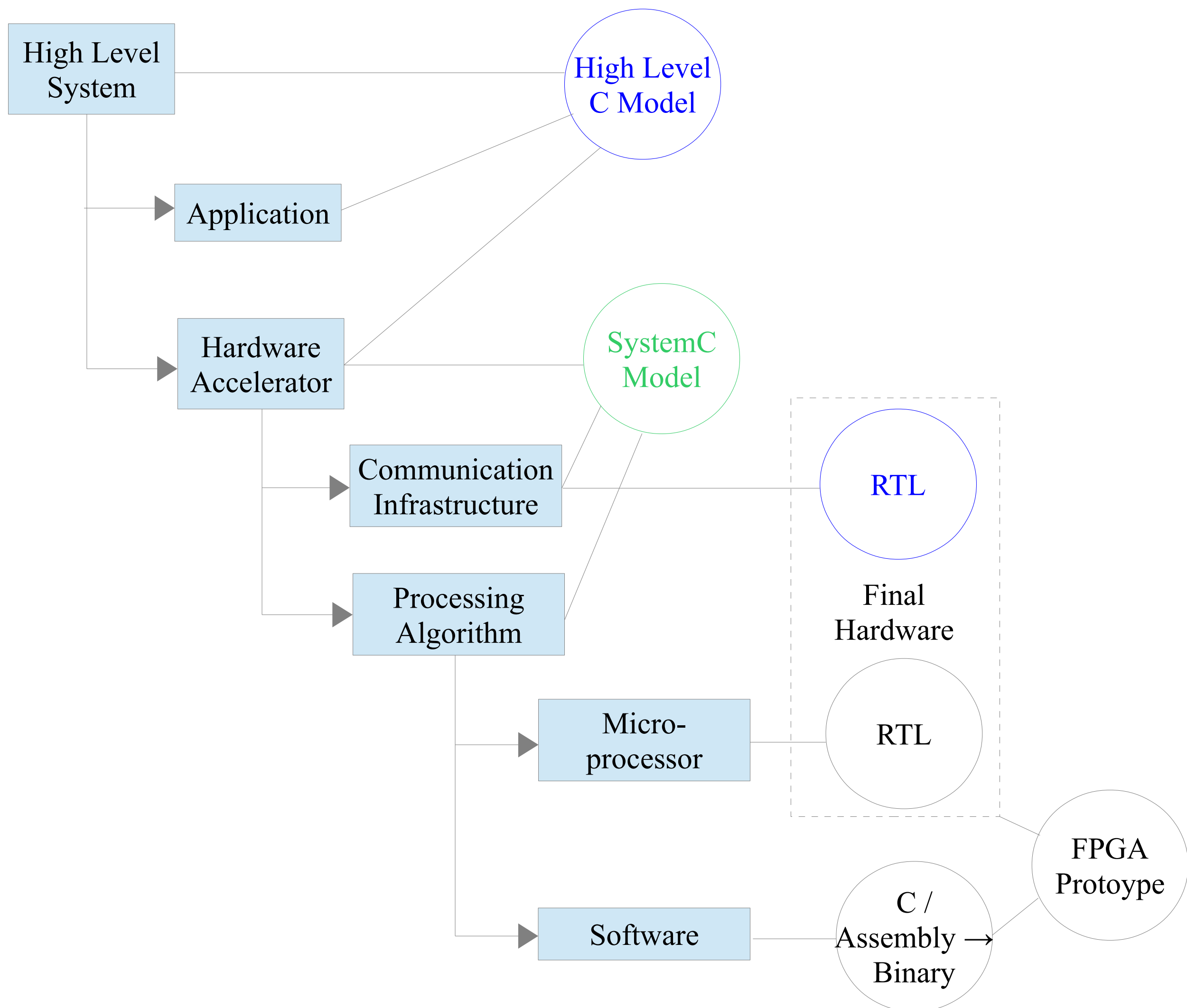


Figure 3: Scope of work in the whole project from system definition to HW/SW. The fields in blue mark the principal contributions in the present semester

1.5 Present Contributions

The previous section describes the different stages of the whole project. In the previous semester (Autumn 2013), the main activity was to define and simulate the SystemC model (marked in green on Figure 3) to evaluate the hardware accelerator. The relevant findings from this work will be summarized in Chapter 2. The main activities in the present semester (Spring 2014) have been:

1. the development of the high level C-model consisting of an example application and a model of the hardware accelerator and simulating it to evaluate the system, and

2. the development and implementation of the communication infrastructure in RTL.

These have been marked in blue on Figure 3. The details of the contributions made in the present semester will be described in the Chapters 3-8, as listed below:

- Chapter 3 describes the motivation and preparation behind the choice of the high level application, supported by extensive literature survey.
- Chapter 4 describes the development of the high level C model, including both an application and a model of the hardware accelerator.
- Chapter 5 describes the evaluation of the system, both through simulating the high level C model, as well as the SystemC model developed last semester and described in Chapter 2.
- Chapter 6 describes the hardware architecture of the accelerator, clearly distinguishing the communication infrastructure from the processor, and then describes the micro-architecture of the communication infrastructure in detail. This has been implemented in Verilog RTL.
- Chapter 7 outlines the verification strategy of the hardware. All the described test-cases have been implemented in Verilog testbenches and used to unit-test the design. However, only completely directed testing was used, and thus the coverage is expected to be low. But it provides a comprehensive description of the various functional requirements to be satisfied, and guidelines for testcases to test them. This can be easily developed to execute fully fledged verification.
- Chapter 8 describes the performance analysis of the design through synthesis and timing.

Thus, in this semester, on one hand, the high level C model has developed and simulated, and on the other the complete hardware for the communication infrastructure has been implemented in Verilog and evaluated for performance, and unit-tested to ensure that it is free of obvious bugs. But it cannot yet be certified as completely bug-free as the verification was not comprehensive.

The next chapters summarize and discuss the findings from the previous chapters (Chapter 9) and discuss the possible direction of future work (Chapter 10) to finish the project by completing the verification of the hardware implemented this semester, by designing and implementing the processor (if not using a standard processor off-the-shelf), by implementing the software, and finally by prototyping the whole HW/SW of the accelerator on an FPGA. These could not be attempted in this semester due to paucity of time, though Section 6.3 does provide a brief guideline for the software development, and synthesizable RTL for the communication infrastructure has been developed, and is ready to be used in prototyping.

Chapter 2

Previous Work

As preparation for implementing the hardware accelerator proposed in Chapter 1, a top level system architecture was first worked out and modeled in SystemC and simulated in order to evaluate its feasibility and quality in terms of various system performance parameters. This activity was executed during the autumn semester 2013, and [11] is the project report for it. Some of the work in the present semester (spring 2014), to be described in the present document, uses the findings reported in [11] as its starting point and design constraints. Hence, the relevant findings and design decisions described in [11] are being summarized and catalogued with proper references in this chapter, so as to place the present activity in its proper context.

2.1 Communication Infrastructure

As described in Chapter 1, the hardware accelerator is made up of a 2-dimensional grid of "processing nodes", connected to each other through a network-on-chip, where each processing node is associated to one or a set of "map segments" and performs the trajectory cost calculation for that/those specific segment(s). Refer to Figure 4. Each trajectory cost calculation problem is represented by a packet (called flight trajectory record, FTR) that is passed along the processing array in the required order, while each node incrementally adds its contribution of the cost to the packet, and the calculation finishes when the packet reaches its destination.

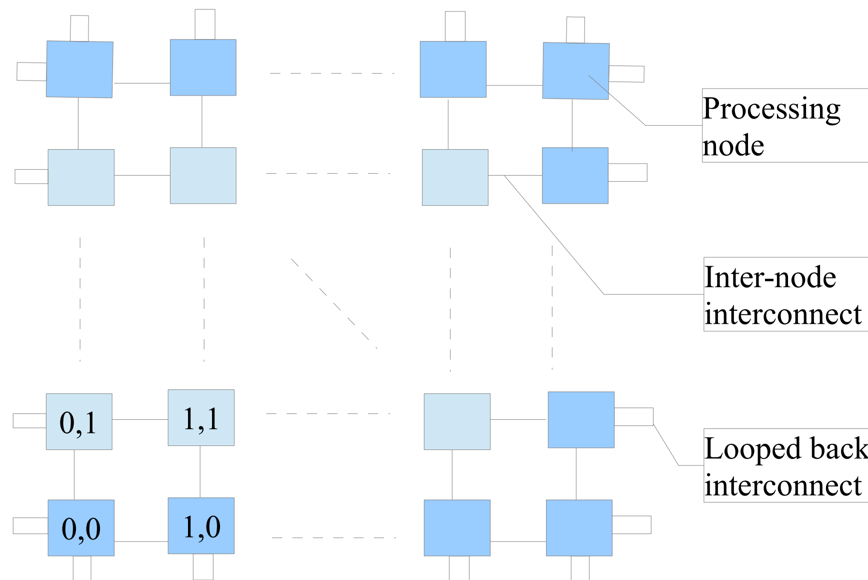


Figure 4: Mesh NoC Topology proposed for CNW

The map folding strategy described in Section 1.3.1 implies that a processing node at the edge of the processing array may be associated to multiple neighboring map segments. When a trajectory passes between such map segments, the "looped back interconnects" as shown above may be utilized to route the corresponding computation packet from the corresponding processing node back to itself.

CNW Routing

The network-on-chip involved in thus transporting the FTR packets has been termed "computation network" (CNW). As is clear from Section 1.3.1, the mapping of the "terrain data map" onto the processing array is performed in such a fashion that any pair of neighboring map segments either map onto the same processing node, or onto a pair of neighboring processing nodes. Also, because the trajectories are assumed to be continuous, it follows that all packet exchanges are limited between neighboring processing nodes. However, when a trajectory enters as well as exits a map segment very close to one of its vertices (or when the trajectory passes from one segment to one of its diagonally neighboring segments exactly through their common vertex), the calculation may be somewhat sped up by routing the corresponding packet diagonally and accepting a cruder cost approximation of the short omitted segment using the cost function of one of its computing neighbors. The support of "diagonal routing" introduces some cases where the inter-node communication is no more local, rather goes through the intermediary of exactly one node, which is a neighbor to both the source node and the diagonally located destination node. To make things easier, the switching mechanism in this case has been taken to be "store-and-forward" (SAF).

CNW Deadlock Resolution

It is to be notes, however, that the trajectories may involve any kind of turns and twists, thus making it imperative that the CNW be able to route packets in all possible directions. As explained in Sections 2.5 and 4.1.3 in [11], this is expected to lead to unavoidable deadlock situations in the network. Hence, a deadlock-resolution strategy was developed to solve this problem. The strategy is based on a deadlock-timeout counting. Each time a processing node tries to pass on a packet to its next neighbor, it initiates a timeout counter, and if the packet cannot be successfully sent out within this period, a deadlock is assumed to have been detected, and the corresponding packet is "dropped" from processing. If the external application requires, it can relaunch the computation of those "dropped" packets later on. But in order for this to happen, the external application needs to be informed of the identity of the dropped packets. Therefore, a mechanism is required for a processing node to communicate this information outside the hardware accelerator, when it is already facing a deadlock in the CNW. In order for this to happen reliably, a secondary low-bandwidth deadlock-free network-on-chip, called the "injection-ejection network" (IENW) was envisioned to connect all the nodes in addition to CNW.

IENW

IENW doubles in for transporting the FTR packets from the external application to the first processing node inside the hardware accelerator, and also from the last node out of the hardware back to the application. Different possible topologies for IENW were discussed and compared in Section 4.2 in [11]. During the course of the work in the present semester, however, a quite different topology was developed and implemented as described in Section 6.2/Figure 35 of the present document. All in all, however, the existence of the IENW ensures that all packets launched by the external application reaches the respective first processing node, and all packets ejected by the hardware, whether because of a detected deadlock or because the computation was finished for it, are transported back to the application. As explained above, the IENW was, however, expected to be simple and low-bandwidth, and thus occupied a lower priority in the design task. Thus the SystemC model developed in last semester did not try to model it accurately, and instead used a very simple replacement as described in Section 6.2/Figure 14 of [11], concentrating instead on modeling the CNW and the processing nodes.

2.2 Processing Node Hardware

The basic structure of the processing node was taken to be made up of the logic required to communicate over CNW and IENW, as well as the logic required to execute the cost calculation. Thus the basic logical structure of the node was envisioned as below on Figure 5, where FIFO's were instantiated on the CNW output interfaces. FIFO's could alternatively be instantiated on the input interfaces.

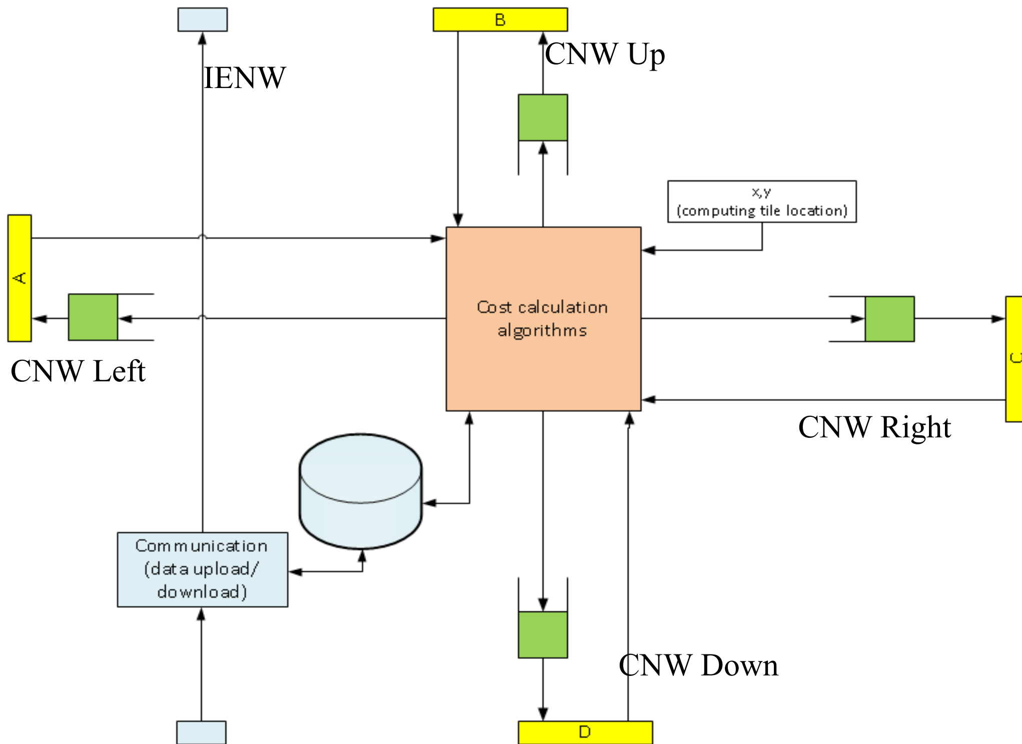


Figure 5: Logical structure of processing node with FIFO on the output

RAM Optimization

A direct hardware implementation of this architecture is, however, deemed to be inefficient in terms of memory usage, especially if targeting an FPGA implementation, as each FIFO needs its own associated memory, but FPGA's only have a limited number of block RAM's to efficiently implement that. Thus, it was decided to merge the RAM's of the FIFO's as well as the data memory of the processor into one physical dual port RAM. This way, the required number of RAM's per processing node is minimized. At the same time, this also eliminated any requirement of copying the FTR data from the FIFO's into the processor data memory. However, it was noticed that this is much easier to implement if the FIFO's are instantiated on the input CNW/IENW interface rather than the output interface, because the knowledge of the correct output interface for a packet presupposes the completion of the local routing calculation, which may actually be performed by the processor, and thus requiring the storage of the packet in the data memory. If the FIFO's are placed on the output, then the packet will need to be copied into the correct FIFO (which may share the same memory, but will likely have its own separate address space) after the completion of the routing calculations, thus introducing extra delay. If the FIFO's are introduced on the inputs, on the other hand, then there is no such copying involved. The sharing of memory among different FIFO's, however, can be achieved only at the expense of a reduced data transfer rate, as only one FIFO can be read/written in any clock-cycle, thus giving rise to a TDMA-like scheduling of the RAM access

for the different FIFO's. This was, however, expected to be a reasonable limitation, because the data availability rate is expected to be determined by the much lower throughput of the processor, which is required to perform fairly complex tasks of arbitration, routing, cost computation, etc. The resultant micro-architecture of the processing node (ignoring the difference between IENW and CNW) is shown on Figure 6 below.

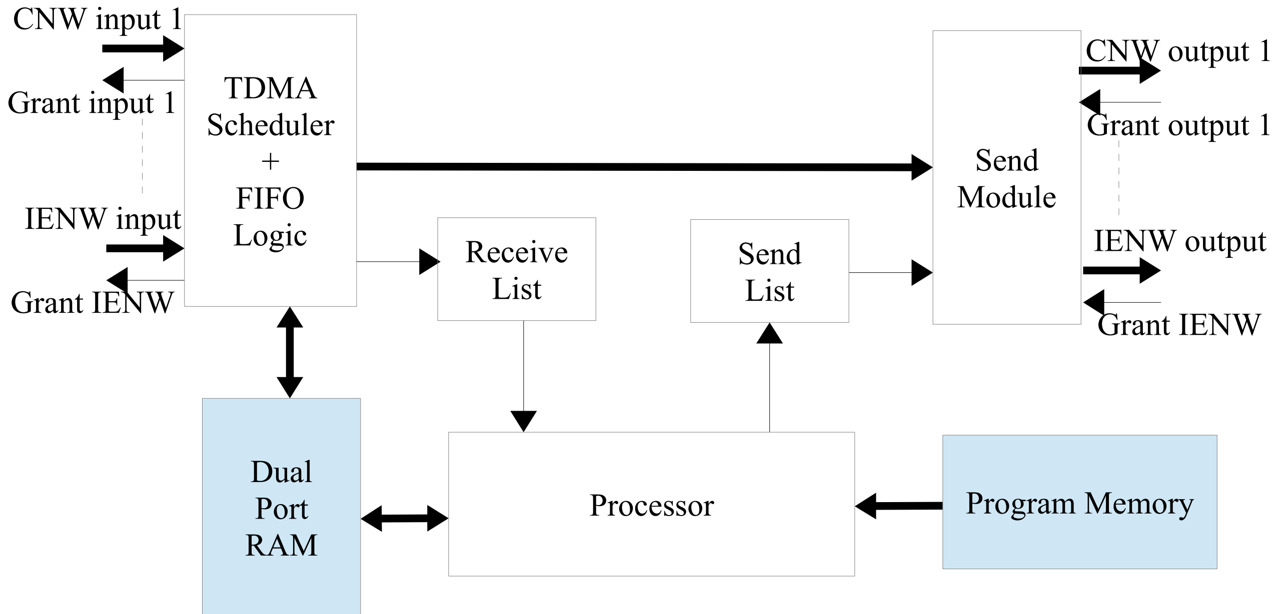


Figure 6: Micro-architecture of Processing Node (with FIFO on input)

The “TDMA scheduler+FIFO logic” block interfaces with the input CNW/IENW ports and writes the incoming packets into the dual port RAM, implementing the FIFO's. It also reads out the packets and sends them to the send-module on request, in order for the packets to be sent out to the next downstream node or to be ejected through the IENW. The processor can manipulate the packets in situ inside the dual port RAM, to be implemented as a block RAM, in case of an FPGA implementation. The “Receive List” data-structure is used by the FIFO-Logic to communicate availability of packets in the FIFO to the processor. Similarly, the “Send List” data-structure is used by the processor to communicate the completion of computation of a packet to the Send-Module, which can then try to send out the completed packet. This design, after some minor modifications, has been used as a starting point for further development and implementation of the hardware in the present semester, as has been explained in Chapter 6.

2.3 Communication Protocols

Packet level protocols for the communication were also worked out in the work described in [11]. As described in Section 5.2 of [11], two kinds of packets were planned for – Virtual Map Records (VMR) and Flight Trajectory Records (FTR). VMR is used to load the map data into the processing nodes, whereas FTR is used to represent a trajectory in the map, and is manipulated by the processing array to compute the associated cost. A list of the required fields in these packets has been presented in the afore-mentioned section. These structures have been found to be adequate during the work performed in this semester, but a couple of restrictions have been imposed on them:

1. The least significant 2 bytes of the 2nd (4 byte-)word of an FTR has to be the num_via (i.e. via-list size) field.

2. The hardware treats VMR packets the same way as FTR's. Hence, the least significant 2 bytes of the 2nd (4 byte-)word of a VMR packet has to equal the size of the packet in words minus the size of FTR header in words, allowing the hardware to calculate the packet size correctly.

Apart from the packet structure, a routing calculation algorithm was also developed, and presented in Section 4.1.4 of [11] for CNW.

2.4 Modeling and Simulation

The design described above was modeled in SystemC, as described in Chapter 6 of [11]. Some simplifications were made for the ease of implementation, when they were not expected to influence the system behavior significantly, e.g. the IENW was implemented as two rooted trees of height 1, and the FIFO's were instantiated between the processing nodes, instead of inside as the connecting elements. The resultant structure is shown on Figure 7.

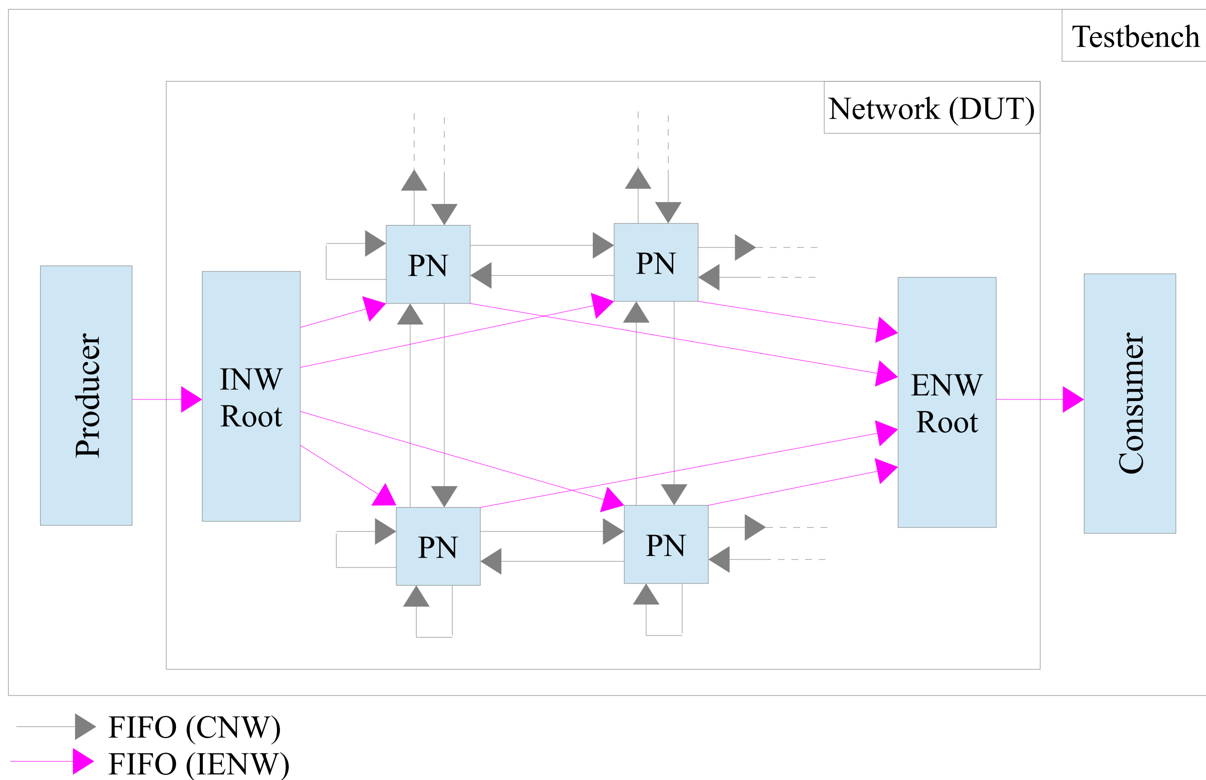


Figure 7: SystemC model and simulation environment

This model was used to thoroughly evaluate many performance parameters of the system, and the results have been presented in Chapter 7 of [11]. The results were consistent with the design expectations, including the fact that the folding concept introduced in Section 1.3.1 does ensure good utilization of the hardware resources when computing costs of random trajectories. Some of the important characteristics of the system as designed and validated by simulation are as follows:

1. When the incoming packet characteristics (e.g. trajectory length, number of via points, etc.) remain statistically stable (e.g. when the packets are completely random), the system seems to have a maximum supported throughput. When packets are fed in at a higher rate than that, many packets are ejected to approximately maintain this saturation throughput of successfully computed output packets (Figure 18 of [11]). So the hardware may be thought of as behaving statistically like a pipeline with initiation interval given by this saturation

throughput.

2. It is hard to predict the computation latency of an individual packet inside the hardware, as it depends on factors like presence and number of other packets in the different nodes it visits, and whether they have arbitration and routing conflicts with this packet, etc. However, the latency also includes a trajectory-length dependent component, which is easier to estimate.
3. Because of the unpredictable latency, and also because the time interval between the incoming packets is usually much smaller than the average latency (in order to exploit the "pipelining" effect described above), the order of completion of packet computation is unpredictable.
4. Not all packets that are launched for computation may finish successfully. Some may be "dropped"/"ejected" in order to solve possible CNW deadlocks, as described above in Section 2.1. As expected, the packet dropping becomes significant when the input throughput exceeds the maximum supported throughput, and statistically increases linearly after that.

Any application that would use the proposed hardware as an accelerator for its trajectory cost calculations must be aware of and accommodate this system behavior.

2.5 Conclusion

It can be concluded that the work from the last semester (autumn 2013) led to a good high level understanding of the expected behavior of the design under development, and it also provided guidelines to further work. While a significant part of the design work has been further developed leading to the hardware implementation in RTL in this semester (Chapter 6), it has also been used to develop a higher level C-model, with help from additional SystemC simulations (Section 5.2) to help in developing and evaluating an application (Chapter 4). The following Chapters 3-5 provide an overview of what kinds of applications may possibly use the proposed hardware as an accelerator, describe a simple application, explain the motivation behind it, and demonstrate that it is indeed possible to use the proposed hardware efficiently with a carefully designed application. This exercise provides useful data about factors to be considered in both application development as well as hardware development. Chapters 6-8, on the other hand, describe a hardware implementation based on the findings of last semester as explained in this chapter, as well as with guidance from the application development. Hence the present work may legitimately be viewed as the continuation of the work done last semester and described in [11].

Chapter 3

Application: Theory and Research

As explained in Chapter 1, the system under consideration is envisioned to be used as a hardware accelerator for computing trajectory costs over a 2-dimensional domain. Evidently, such a hardware will be useful to speed up applications which require computation of many 2D trajectory costs. One obvious application would be the determination of the optimal trajectory between two points on a map under different constraints and cost functions using heuristics that need to evaluate the costs of many different trajectories. The literature was surveyed to find out the state-of-the-art techniques in the relevant field of motion and trajectory planning, Cf. [1] – [10].

3.1 Motion and Trajectory Planning

Motion and trajectory planning, as discussed in [1], is a vast field of investigation with numerous practical applications, e.g. finding the best or a feasible trajectory between two points on a geographical 2D map under different constraints. The quality of the trajectory may be decided, e.g. by its length, or the time consumed to traverse it by a given vehicle, or the energy expended in doing so, etc. The constraints may be environmental, e.g. the terrain may have obstacles, or it may derive from the degrees of freedom of movement, the body under motion possesses, e.g. human beings can control their speed and direction of motion in all possible directions on a 2-D surface, i.e. they have 2 degrees of freedom of translational motion – along the two perpendicular Cartesian axes on the 2-D surface, and 1 degree of freedom of rotational motion around the axis perpendicular to the surface. A car, on the other hand, can only accelerate or decelerate along the front-rear axis, and turn left-right around the vertical axis, and thus has 2 degrees of freedom. These two kinds of planning problems are described respectively as holonomic – having full control over motion in all directions – and nonholonomic problems, which concern less controllable bodies or vehicles. In either case, the motion (i.e. speed or direction) of a moving body can be altered in a specific way by applying a specific "control input" (e.g. turning the steering wheel or stepping on the gas or brake pedal of a car) for a specified amount of time. The solution of a motion planning problem is thus given by an ordered set of {control input, duration} pairs. In order to optimize a trajectory, it is often required to first solve a feasible motion planning problem, and then use trajectory planning techniques to optimize the resulting trajectory.

[2] proposes an efficient algorithm for finding a feasible solution to a nonholonomic motion planning problem. Explained in terms of a geographical terrain, the problem is to find a feasible trajectory for a nonholonomic body to move from one point to another on a 2-D surface under environmental constraints like obstacles. The heuristics first divides up the terrain coarsely into a rectangular grid of subsections, and searches for a feasible trajectory from the given start point, assuming that the control input changes at most one time in every map segment. It builds a search tree using constrained depth-first search (i.e. the depth difference between the search tree leaves is below a specified constraint at all points) while satisfying all nonholonomic constraints. The terrain divisions are made finer in stages, i.e. control inputs are allowed to change more frequently, if it is necesary to find a solution. This approach finds feasible, but not optimal, trajectories as can be seen on Figure 8.

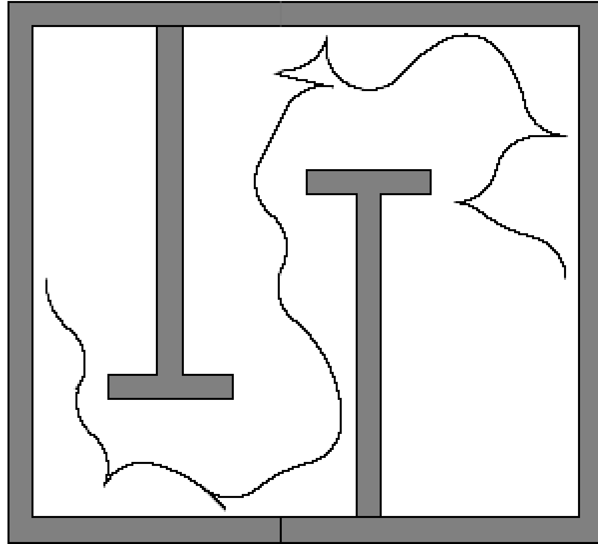


Figure 8: Example solution trajectory for a feasible nonholonomic motion planning problem in 2-D under presence of obstacles. Ref [2]

3.2 Trajectory Optimization Problems

Some optimal trajectory planning problems have been discussed in [3]–[6]. [3] presents a detailed modeling of radar cross sections of unmanned aerial vehicles (UAV), hostile air defense systems involving RADARs and surface-to-air missiles (SAM), and aircraft controls. It then goes on to plan the motion and the trajectory of such a UAV that has to fly into an area under air defense coverage with known RADAR locations and go from one given point to another, while minimizing its probability of getting shot down. The UAV may have other "mission tasks" specified as a list of fly-by points along with the times of arrival at those points. The algorithm characterizes the location and movement of the UAV on the map as either unthreatened or threatened, and tries to minimize the time spent under high threat.

[6] describes another military application where the task is to plan the motion of a UAV to follow a target while there are threats from known RADAR locations on the map. The proposed algorithm tries to keep the UAV within a defined proximity radius of the target as long as possible. Under some circumstances, e.g. in order to avoid some threat on the map, it may have to move so far away from the target that its sensors cannot track the target any more. In such cases it tries to estimate the trajectory of the target and tries to meet it at a point on this estimated trajectory in future. If the target remains undetected for a specified time-out, the UAV returns to base avoiding the threats. This involves an online alogirthm with dynamic inputs about the target's motion, as well as changes in the threat map.

[4] presents another UAV motion planning problem, but now for reconnaissance missions without threat. It has to photograph some given targets in a terrain. Assuming that the UAV always flies on a plain at constant altitude, the areas on this plain from where various targets are photographable are polygons, as the terrain may hide the objects from specific locations. The algorithm proposes a way to find out the optimal trajectory to visit all the polygons and then come back to the initial point. This is a variant of the traveling salesman problem, called Polygon-Visiting Dubin's Traveling Salesman Problem, where the UAV has been modeled as Dubin's vehicle, i.e. a vehicle that only goes forward and turns left/right in an arc. An illustrative solution for such a problem, as computed by a genetic algorithm is shown below:

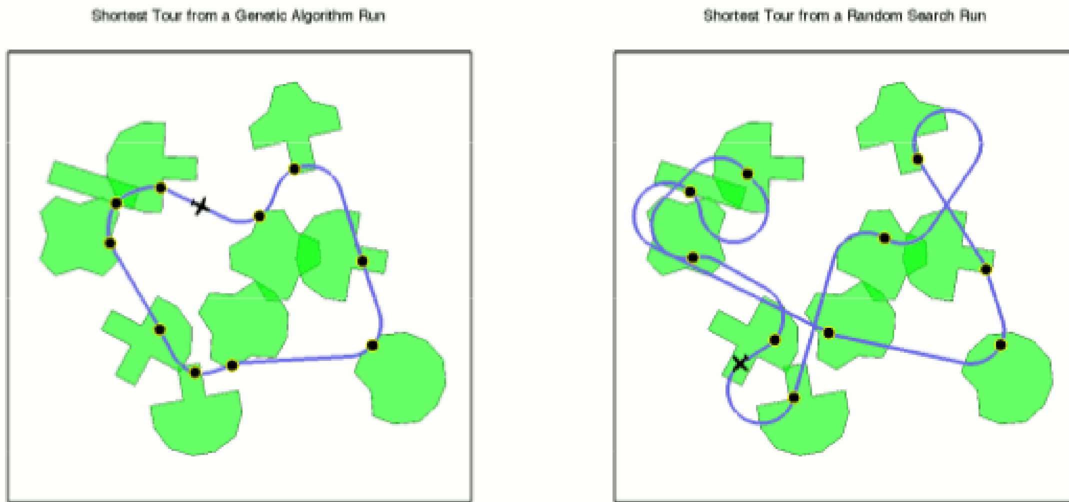


Figure 9: Example solution for finding the shortest trajectory of a Dubin's vehicle visiting all the green polygons and coming back to its source – as computed by a genetic algorithm. Ref [4].

[5] discusses a general aircraft trajectory optimization problem, relevant to both military and civilian applications. It provides a detailed 6 degrees of freedom (DOF) aircraft motion model, viz. translation along the 3 Cartesian axes and rotations around them, with a 4-DOF control model: throttle (1-axis translation), and elevator, aileron and rudder providing control on the 3 rotation axes. It then determines a point to point trajectory on a given terrain to optimize one of the 3 parameters (or a linear combination of them): minimize time of flight, minimize energy expended in flight or maximize time within a given elevation over the terrain – the last one is motivated by requirements to avoid detection by RADAR. There may or may not be threats present on the map. Figure 10 presents two example solutions found for a purely time-optimization problem using this algorithm but two different optimization heuristics:

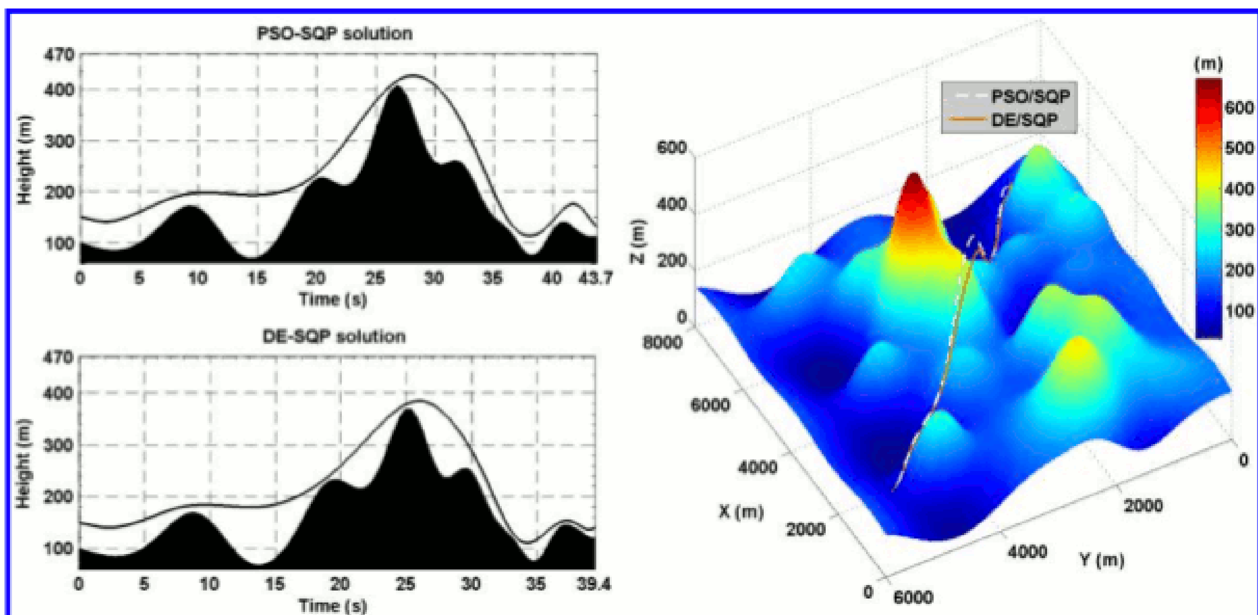


Figure 10: Example solution trajectories for a flight time optimization problem for an airplane flying over a 3-D terrain – using two different optimization heuristics – PSO-SQP and DE-SQP. Ref [5]

The algorithms in [3]–[5] can be regarded generally as composed of a system model, a formulation of the problem as a static motion optimization problem on a known map, and then executing the optimization. Different papers employ different optimization algorithms, e.g. [3] uses Matlab fminimax() function, [4] uses a genetic algorithm, and [5] uses a number of advanced optimization algorithms, e.g. Particle Swarm Optimization – Sequential Quadratic Programming (PSO-SQP) and Differential Evolution – Sequential Quadratic Programming (DE-SQP) combinations. [6] also involves the similar setup of modeling, stating the optimization problem and applying optimization. While modeling and problem statement is a design decision for all of these algorithms, the optimization has to be run for each problem configuration, and in case of [6] it has to be done on-line in real time in a possibly changing threat map. This means, it is crucial to accelerate the optimization process to make the implementation of these algorithms feasible. The optimization problems generally involve evaluating the cost or fitness of many different trajectories. Thus it is expected that it is useful to design a hardware accelerator that can parallelize and speed up this cost calculation, and the proposed hardware in the present work may indeed fill in this requirement.

3.3 Parallelization of Optimization Algorithms

One key factor that influences the performance of such a hardware accelerator is the degree of parallelizability of the implemented optimization algorithm itself. Two well-known and widely used optimization algorithms are simulated annealing (SA) and genetic algorithms (GA), the latter one has in fact been used in [4]. In order to find out a practical application suitable for the hardware under development, the literature was surveyed to find out ways to parallelize these algorithms.

3.3.1 Genetic Algorithm

Genetic algorithm (GA) is an iterative optimization heuristic based on the principles of natural selection and biological evolution. In case of biological organisms, the principle of natural selection states that in each generation, the fitter individuals have higher reproductive success and pass on their genes, and thus tend to produce a fitter next generation, where the fitness is evaluated with respect to the environment the population lives in. Thus, it is expected that after the elapse of an appropriate number of generations, the population contains individuals very close to the fittest possible organisms for that particular ecological environment. In terms of the general optimization problem, the fitness to the environment is emulated by a fitness function of a candidate solution, which acts as the analogue of an individual organism. Just like in the case of the biological population, the evolution progresses by deriving new generations of a population of candidate solutions. Each solution is encoded as a “chromosome” containing multiple “genes”. The reproductive success of the organisms is paralleled by a “parent selection strategy”, and sexual reproduction is imitated by a cross-over operation, akin to the same phenomenon in biological meiosis cell division, that generates the gametes, the basic cells involved in sexual reproduction. In more concrete terms, in genetic algorithm, the cross-over is applied on two “parent” chromosomes to derive two “child” chromosomes by exchanging some genes between the two parent chromosomes. Biological mutation may also be emulated in the GA by inducing small perturbations in the chromosome structure. The algorithm can also incorporate other optional operations.

[7] provides a general survey of various parallel GA implementations. It divides them into 3 main groups:

1. **Single population master-slave:** A “master” process maintains the population of solutions globally and perform the genetic operations on them, e.g. cross-over and mutation. However, the evaluation of the fitness of different solutions is distributed over multiple parallel processes.

2. **Single population fine grained:** There is a global population, but each process treats only one (or a small number of) individual solutions, and there is no global master. The individuals are assumed to be connected with each other through a topology in a connected graph – usually same as the connection topology of the parallel computer, that the computation is running on. Parent selection and cross-over are localized, i.e. each process can only interact with its neighbours.

3. **Multiple population coarse grained:** In this approach, each process treats a separate population, and applies GA on it independently to evolve it. Only occasionally, some individuals "migrate" between the processes allowing exchange of information among them.

It can be observed that if an application is developed that uses GA to optimize 2-D trajectories, then the hardware structure proposed in Chapters 1 and 2 can be used to accelerate any of these parallelization approaches, as long as the hardware is used only to accelerate the computation of the fitness of the candidate solutions.

3.3.2 Simulated Annealing

Simulated annealing is another popular iterative optimization heuristics, based on the physical phenomenon of annealing in metals. This involves heating the material to a temperature above a critical temperature and then cooling it slowly so that the crystal structure inside always remains at thermal equilibrium. When the metal is thus brought to room temperature, its crystal structure assumes the lowest possible energy state, being the most regular possible, avoiding internal stress and lattice defects. This process is paralleled in the domain of general optimization algorithms by drawing an analogue of the general solution quality function to the physical energy function, and by defining a gradually diminishing temperature at each iteration which determines the degree of acceptable "disorder" in the solution. In each iteration, the present solution is slightly perturbed and its energy-function is evaluated. This is accepted as the new solution if it has a lower energy than the previous solution. Even if its energy is higher, it may still be accepted probabilistically, depending on the exact amount of energy increase and the iteration-temperature (i.e. acceptable degree of disorder). Smaller increases have higher probability of being accepted if the iteration temperature remains fixed, and higher iteration temperature allows higher probability of acceptance when the energy-increase is same. The basic algorithm is completely specified by defining an energy function, a cooling schedule and an acceptance criterion. The last is often defined as an exponential function producing the probability of acceptance (PA) of a new solution as

$$
\begin{aligned}
PA &= 1 \quad \text{if } M > 1 \\
&= M \quad \text{otherwise} \\
&\text{where } M = e^{(-\Delta E / T)} \\
&\Delta E = \text{Increment of energy} \\
&T = \text{Iteration temperature}
\end{aligned}
$$

[8] discusses some strategies to parallelize simulated annealing (SA). This paper proposes the following approaches:

1. **Asynchronous (AS):** This is parallelization of the multi-start approach, i.e. independent runs of the SA algorithm with different random initial solutions are executed in parallel by multiple processes, and the best solution is chosen at the end.

2. **Synchronous with occasional solution exchange (SOS):** Similar to AS, but after some specified number of iterations, all the solutions are gathered by a master process, and some genetic operations (e.g. cross-over) and selection are applied on them to form a new solution population, and SA is again applied on them in a distributed manner for the specified

number of iterations, before gathering the population in the master again and repeating the process. The annealing temperature is globally managed at each synchronization at the master. This approach is a combination of SA and GA.

3. **Synchronous with occasional enforcement of best solution – fixed interval (SOEB-F):** In this approach, all processes start with the same random initial solution and temperature. Like in SOS, all the results are gathered by a master after a specified number of iterations in each process, and then the best solution from the population is chosen for further SA iterations to be done parallelly on the distributed processes. The temperature is controlled globally at each synchronization step, like in SOS.
4. **Synchronous with occasional enforcement of best solution – varying interval (SOEB-V):** Like SOEB-F, all processes start with the same initial condition and temperature, and apply the SA algorithm independently. However, each process controls its own annealing temperature, and communicates its result to the master when a specified temperature reduction has been achieved. When the master has received results from all parallel processes, it chooses the best one from them and broadcasts it to all processes for further processing till all of them reach the next temperature target. The process is repeated. In this process, all the processes are synchronized when they are at the same temperature.
5. **Highly coupled synchronous approach (HCS):** All processes start with the same random initial solution and the same temperature. The results are reported back to the master after each iteration in each process. Additionally, each process is allowed to perturb only one axis/variable of the solution in each iteration. The master then chooses the best solution after perturbation, and integrates other less performing perturbations into it as long as that improves the solution. Then, each process receives this composite solution and gets an axis/variable specified, on which to perform the perturbation in the next iteration. This approach has a very high communication overhead on a traditional parallel computer.
6. **Modified highly coupled synchronous approach (MHCS):** In order to reduce the communication overhead in HCS, the synchronization/merging of solutions is performed only at an interval of a specified number of iterations, yielding the MHCS algorithm.

These methods are then benchmarked in [8] with respect to a standard suit of test functions. MHCS was found to be the best performing in terms of speed and quality of result for most kinds of problems. AS was found to be the usually the worst performing.

[9] proposed two parallel SA implementations. The first of them, viz. “Clustering algorithm” is in essence same as the SOEB-F from [8], except that the initial solutions for the different processes are different. The second algorithm is called “Genetic clustering algorithm”, and it uses a genetic algorithm to find a population of good solutions, which are then used as the initial solution for the “clustering algorithm” mentioned above.

[10] is a highly theoretical paper that explores the Markov process based optimization heuristics, e.g. SA and GA, and explores their relationships. It proposes a “Parallel simulated annealing” algorithm by performing the perturbations in parallel and then choosing the best among them using an acceptance probability. This seems similar to SOEB algorithms from [8], assuming that the synchronization is performed after every iteration. The paper then introduces the “Massively parallel simulated annealing”. This is exactly same as the AS algorithm from [8]. This algorithm is then extended by defining a “parallel neighbourhood model”, where the processes are assumed to be connected to each other in a defined topology in a connected graph like in the case of “single population fine grained” parallel GA described in [7]. A “parallel neighborhood algorithm” has been defined on this topology that allows interaction between the neighboring processes, just like in the corresponding GA schemes, allowing improvement of the solution population as a whole.

3.4 Different Implementation Options

The parallel algorithms described in [7] – [10] have been discussed for shared or distributed memory processors. When these approaches are applied to solve the trajectory planning problems, as described in [3] – [6], some challenges are faced. Because a realistic map of, e.g. terrain elevation, threat location, etc. is difficult to store in a compact representation (e.g. as analytical functions), these data are in general memory intensive. This leads to the following challenges in different parallelization approaches:

1. **Shared memory:** In case of parallel shared memory processors (e.g. GPU using global memory), though the map data may be shared, because of the parallel execution of different processes dealing with different trajectories, a lot of memory access conflicts are generated. This leads to memory bandwidth bottleneck.
2. **Distributed memory:** In this case, two different approaches can be taken:

(a) **Data duplication:** In this approach the map data is duplicated in the locally accessible memory of each processor. However, this is expected to be quite expensive in terms of memory requirements.

(b) **Data fragmentation:** In this approach, the map data is divided up into smaller fragments, each of which is stored in a small memory, local to a processor. Then the cost calculation for each trajectory can be accomplished by communicating between these processors, each computing the cost of its local part of the trajectory. This, however, assumes that the overall cost of a trajectory can be determined by (linear or some other analytical) combination of the costs of the individual trajectory segments. This way, only one copy of the map data is needed to be stored, and because it is distributed in many memories, there is no memory bandwidth limitations either. However, this improvement comes at an increased cost of communication between the processors, which is expected to be acceptable as all communications take place between neighbors.

It is apparent that the data duplication approach may be suitable for distributed systems with lot of memory but relatively slow communication, e.g. computing clusters. The data fragmentation approach would be more suitable for systems with low amount of memory, but relatively faster communication, e.g. a network-on-chip connecting an array of on-chip processor cores.

As explained above, all three approaches have their own advantages and disadvantages. For the off-line optimization problems described in [3] – [5], probably any of these three parallelization techniques can be applied. The on-line optimization problem, discussed in [6], however, probably needs an energy-efficient low-memory on-chip solution, e.g. a GPU-based (shared memory) or a NoC-based (fragmented distributed memory) solution. The present work aims to realize this last solution: a NoC-based "fragmented" distributed memory system for parallelization of these trajectory optimization problems.

3.5 Conclusion

It is apparent from the discussion in this chapter that trajectory optimization problems are one of the most important group of problems with practical applications, and the NoC-based distributed hardware proposed in Chapters 1 and 2 is perfectly placed to accelerate such computations. However, the architecture has some limitations and specific characteristics which need to be accommodated by the application to optimize the hardware usage. The next two chapters describe the design and evaluation of some such applications, and demonstrates the usefulness of the hardware to accelerate the solution of some interesting practical problems by these applications.

Chapter 4

Application: Design

One of the key steps in system design is to estimate its performance early in the design cycle through a high level model, and demonstrate its effectiveness. The same was attempted for the present problem by developing a high level model incorporating both a model of the hardware accelerator and an application, that would exploit the capabilities of this hardware. The aim of this exercise is to demonstrate the feasibility and the performance of the proposed system at a high level, as well as to provide guidance to future design of its two separate components – the application and the hardware. As explained in the Chapters 1 and 2, and as justified in Section 3.4, the targeted hardware is a NoC based homogeneous distributed computing system, each node of which will run its own low level software/firmware. But at the global level, an application software will run on an external master processor, which is envisioned to offload the trajectory cost calculations to the NoC-based hardware accelerator. The master processor itself may be distributed too. The connection fabric between the accelerator and the processor has to take care of the necessary arbitration and routing in that case. So, the rough high-level view of the application-hardware interaction is as shown on Figure 11:

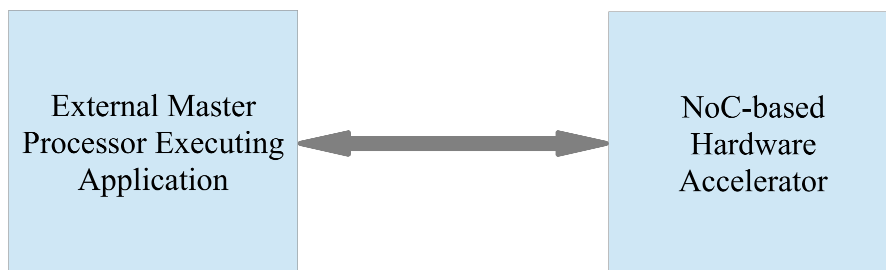


Figure 11: Proposed high-level design partitioning between application and hardware accelerator

4.1 Hardware Modeling

The hardware accelerator, described in Chapters 1 and 2, possesses the following properties as explained in Section 2.4. These require to be implemented in the high-level hardware model:

1. **Pipelined execution:** This means that in order to use the hardware optimally the application should be able to feed trajectory cost calculation problems into the accelerator at a near constant rate close to the accelerator's saturation through-put, which may, however, drift slowly during the execution. Some variations are acceptable due to buffering of packets inside the hardware, but stopping and starting repeatedly after relatively short bursts of busy time is inefficient. It also implies that the transaction through-put, generated by a well-designed application, should adapt itself to the changing saturation through-put of the hardware.

2. **Weakly predictable latency:** The main component of the computation latency of a packet, when the hardware has low load, is the product of the number of processing nodes it visits and the average processing time at each node. The former can be estimated by noting that most of the times, if a trajectory has two consecutive via-points in segments (x_1, y_1) and (x_2, y_2), the packet has to traverse ($|x_1-x_2|+|y_1-y_2|+1$) nodes to cover these points. This is also an exact upper limit on the quantity. The processing time at each node is somewhat harder to

know as the processing software may involve many asymmetric branches, but it can be estimated within some range. On top of these quantities, there also exists latency, stemming from network congestion inside the accelerator, when the load on the hardware is nontrivial. This is hard to understand and estimate from outside. As a result, the latency of each packet is only weakly predictable. Thus, the packet latency may be estimated based on the list of via-points by deriving from it the approximate number of processing nodes visited, with a hardware load-dependent random overhead added to it.

3. **Out of order execution:** The execution latency of each packet strongly depends on its content (e.g. the list of via points) as well as other factors (e.g. congestion). As a result, packets often finish execution in a different order than they were launched in.
4. **Packet dropping:** It is possible that some packets are dropped by the hardware because of network congestion, in order to resolve potential deadlocks.

The parallel global optimization heuristics described in Section 3.3 can be implemented in an application, adapted to the hardware under consideration. Each individual trajectory cost or fitness calculation can be performed through an FTR packet. The results can then be used by the application running the optimization algorithm on the master processor. For simplicity, it is assumed in the sample application that the master processor is sequential, though the accelerator hardware does not depend on this assumption, as explained before. Two different applications have been explored, as explained below.

4.2 Application Implementing Simulated Annealing

The first algorithm explored is based on the "Asynchronous Parallel Simulated Annealing" as explained in Section 3.3.2. The sample problem to solve is to find the lowest cost trajectory from one given point to another on a 2-D map with local costs. In the beginning, the map is divided into rectangular grids, each grid with uniform cost in it, and this information is loaded into the hardware accelerator. Then the application runs two parallel threads – dispatcher and recipient, and maintains a "packet queue" of N number of FTR packets, and an "accounting table" containing N entries, each comprising the fields – present cost, temperature and associated trajectory.

Recipient thread:

1. Initially the costs and temperatures in the "accounting table" are initialized at the maximum possible values, and N random initial paths from point A to B – each path described by an FTR packet with a unique packet ID between 0 and N-1 – are generated, and they are stored in the "packet queue". Each packet with ID = k is associated with the entry in the k-th location of the accounting table, and the corresponding trajectory is also stored there. (Dispatcher thread dispatches these packets to the hardware.)
2. Then the thread continually receives packets from the hardware accelerator, and if the packet was ejected due to congestion, puts it back into the packet queue, or else calculates the acceptance probability of the trajectory using the newly calculated cost as well as the temperature and past cost associated with this packet retrieved from the accounting table. Then a random number in [0,1) is generated and compared against the acceptance probability to determine whether this trajectory is acceptable. If accepted, the table entry is updated, and a perturbation is applied on the trajectory to produce a new FTR, which is then put into the "packet queue". If not accepted, then the corresponding trajectory stored in the accounting table is retrieved and a perturbation is applied on that to produce a new FTR, which is then put into the packet queue. In either case, the packet ID is kept intact. This way, the algorithm is completely independent of unpredictability of the packet latency and

execution order, but packet dropping will slow down the convergence rate.

3. At the end, the cost entries in the table are compared, and the trajectory with the best cost is chosen as the solution to the problem of finding the best trajectory.

Dispatcher thread:

1. The dispatcher thread takes a packet from the head of the packet queue after each interval of a specified time, as long as there is an available packet, and sends it into the hardware accelerator for cost calculation. This way a smooth controllable throughput into the accelerator is ensured.

These threads can be scheduled with a well-defined period, making them attractive for real time applications. It is possible to reduce the size of the FTR queue to a small value (~3-4) by careful scheduling of the recipient and dispatcher threads.

4.3 Application Implementing Genetic Algorithm

The next algorithm considered for implementation is based on the "Multiple population coarse grained parallel GA" approach as explained in Section 3.3.1. In this approach, M populations of N individual candidate trajectories are created in the master processor. The trajectories, represented by FTR's, are, as usual, sent in to the accelerator to calculate their cost, which then gives their fitness inside the respective population. As before, we need a queue of FTR's from which a dispatcher thread extracts packets to send into the accelerator. We need another thread that receives the packets from the accelerator and fills in an MxN array. Whenever one population is complete, it is evolved into the next generation through crossover and mutation, and occasional migration is also applied.

4.3.1 Algorithm Description

[4] describes a GA implementation to solve the optimization problem of a closed loop trajectory. This algorithm has been adapted here to solve a point to point trajectory optimization problem. Following are the different design details of the implemented algorithm:

Genetic encoding:

Each chromosome corresponds to a trajectory. Each via-point, including the start and end points, is a gene. The start and the end points, or equivalently the first and the last genes, are identical for all chromosomes. It is assumed that each chromosome (trajectory) has at least 3 and at most a user defined constant number of genes (via points).

Fitness value:

The fitness value of each chromosome has been defined to be the reciprocal of the corresponding trajectory's cost. Thus, the lower the cost, the higher the fitness.

Elitism:

Like in [4], an elitist version of GA has been used, i.e. a small number of the fittest chromosomes are passed to the next generation unaltered. This ensures that the fitness of the fittest chromosome in the population can never go down from one generation to the next. The proportion of the chromosomes thus passed on is user-defined.

Parent selection strategy:

The selection scheme is "fitness proportionate", also known as the "roulette wheel" strategy, as described in [4]. In other words, the normalized fitness of a chromosome is equal to its probability of getting selected in any draw.

Cross-over:

For crossover, two parents are chosen from the population of chromosomes by random draws using the selection strategy as explained above. It is permitted for the same chromosome to be chosen as both parents. Then, a cut-point is chosen for each trajectory. The cut-points in the two parents are aligned and a cross-over is then performed. Because the cut-points are chosen (more or less) independently for the two chromosomes, they allow for length variation in the chromosomes, or equivalently, variation in the number of via points in the trajectories. When choosing the via points, it is however ensured that no chromosome with less than 3 or more than a user-specified constant number of genes is generated. This strategy is very different from the one described in [4] owing to the fundamental difference in the chosen problem to solve.

Mutation:

Mutation is a very important part of the algorithm as it has been implemented. Two different kinds of mutations have been implemented:

- **Gene insertion:** It has been observed that the cross-over scheme used seems to lead to production of many children with very asymmetric chromosome lengths, the longer of which tend to have lower fitness and get eliminated quite early, leading to an over-all diminished average chromosome length in the population, as it evolves. Many a times, the population would thus get dominated by shorter chromosomes quite early on, even though they may not be globally the fittest. In order to counter this bias towards shorter chromosomes, a gene insertion mutation scheme has been developed. After crossover, the shorter child-chromosome is chosen, or if both the child chromosomes have the same length, then both are chosen. With a user-specified probability, it is decided whether to add a gene into them. If it is so decided, a point is selected on the corresponding trajectory uniform-randomly, and a new via-point, and equivalently a gene, is created at that point. This mutation does not alter the underlying physical trajectory, but creates a new gene in its encoding, which is now available for gene mutation as well as for the placement of a crossover cut-point in the next generation.
- **Gene mutation:** After the crossover and possible gene insertion, the child chromosomes are chosen for mutation with a user-specified probability. If a child is so chosen, then one of the intermediate via-points (i.e. excluding the start and end points) of the corresponding trajectory is chosen at random. This via-point is then set to a random point within a unit-square around it. In case, a point thus selected falls outside the terrain under treatment, the nearest point on the edge of the terrain is chosen instead. This mutation scheme is loosely based on the "position shift" mutation strategy of [4].

Refer to Figure 12 for a pictorial explanation of the cross-over and mutation operations.

Migration:

The algorithm implements a fixed ring topology of migration. Thus, population 0 passes to population 1, population 1 to 2, ... , population (N-1) to N and population N to 0, where (N+1) is the number of populations. User can, however, choose to enable or disable migration, and can control the exact behavior of migration by the following two parameters:

MIGR_GEN_GAP = number of generations passed between two successive migrations
MIGR_PERC = %-age of population size that is passed in each migration

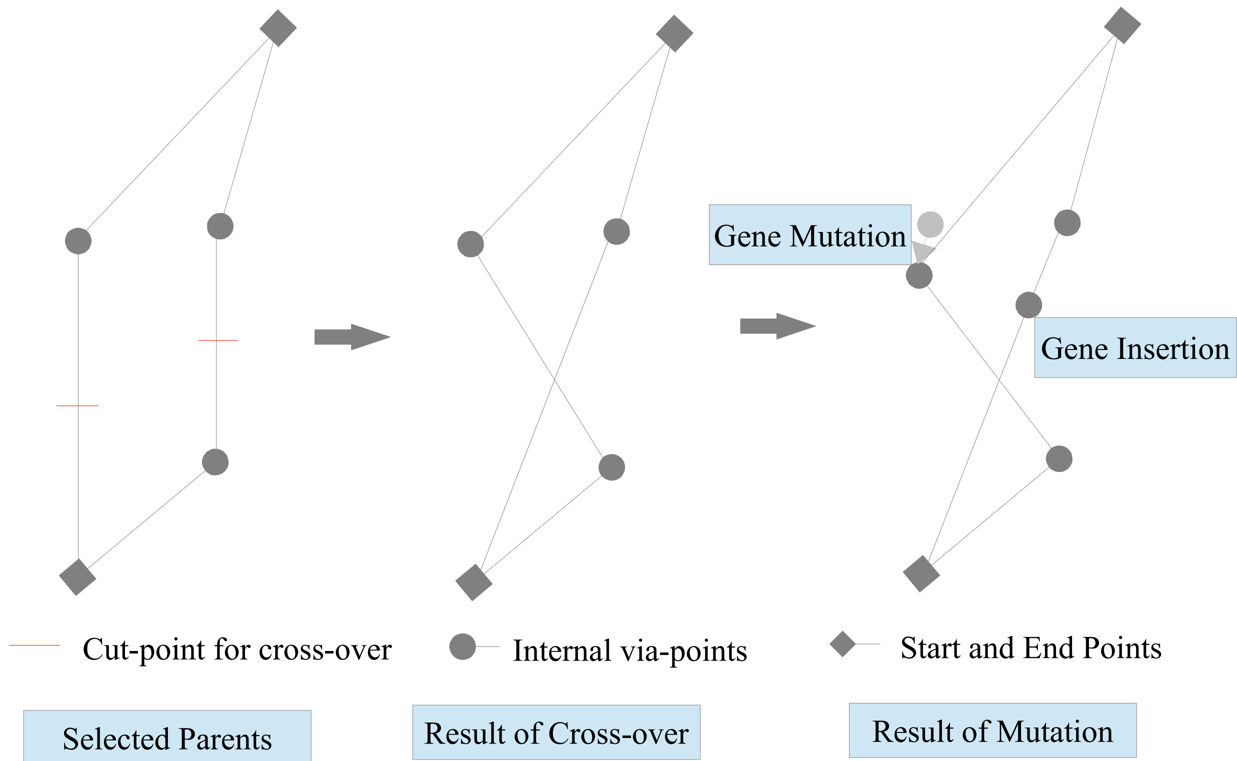


Figure 12: Application of genetic operations

Because of re-computation of packets, which may be necessary when too many packets are dropped by the hardware, different populations may be in different generations, even within the same application iteration. The migration strategy ignores this difference in generation, and simply uses the latest available generation of the migration source. The uniform generation gap of MIGR_GEN_GAP is maintained at the destination population. Under the unlikely event that not enough completed solutions are available in the present iteration in the source population (due to excessive packet ejection because of deadlocks), the destination population uses only those solutions which are actually available for migration.

4.3.2 Application Design

The sample problem chosen to be solved is same as in Section 4.2, viz. to find the lowest cost trajectory from one given point to another on a 2-D map with local costs. The application implementing the above-mentioned algorithm is envisaged as consisting of 3 parallel threads:

- Dispatcher
- Gatherer
- Genetic operator

They communicate amongst themselves through two shared memory data structures:

- Population table (Array of arrays of chromosomes/trajectories + their costs)
- Dispatch queue (Queue of chromosomes/trajectories, whose costs have yet to be calculated)

At the beginning of the execution, Dispatcher and Gatherer threads are initiated, and the data structures are initialized to empty. Then the “Initializer” routine initializes M populations and stores

them into the Dispatch queue as well as population table. After that, all the threads are initiated.

Dispatcher thread:

The Dispatcher thread polls for available chromosome/trajectory/FTR in the dispatch queue periodically, and if an FTR is available, it is read out of the queue and sent into the hardware accelerator to calculate its cost. Ideally, at the steady state, the frequency of polling should be matched to the saturation throughput of the hardware for the best performance.

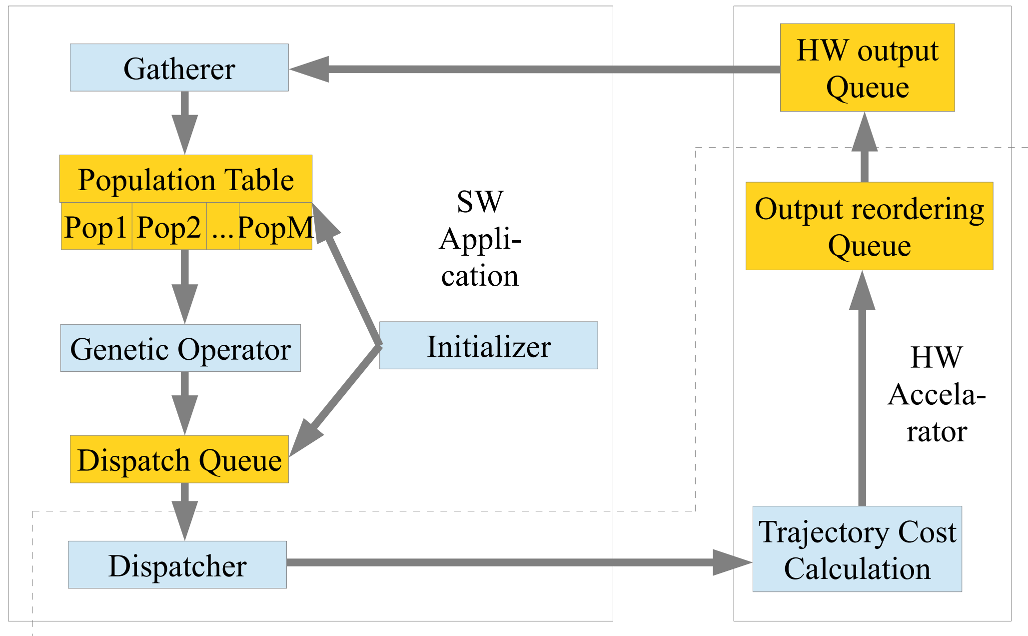


Figure 13: Basic Simulation Architecture of Application. The modules enclosed by the dotted line are all part of the "Dispatcher" thread for the simulation purpose.

Gatherer thread:

The Gatherer thread receives all FTR's coming from the hardware accelerator, and identifies which population each of them belongs to, by their packet ID. It then puts each packet into the right slot of the population table. Each population is assigned a specific memory area in the table, so that they can be accessed at random as normal arrays.

Genetic Operator thread:

1. When the genetic operator thread is idle, it tries to read from the "Population Table" the population, which has the current turn. The read gets blocked until the population completes execution in the hardware, and gets stored into the proper population table by the Gatherer thread. The turns are given in a predefined order (0, 1, 2, …, N, 0, 1, 2 …). After having read a population, the Genetic Operator thread applies the genetic operators on it to derive the next generation for that population and writes the resultant FTR's back into the population table, as well as in the dispatch queue, except for the "elite" solutions. The costs of the elite solutions are already known from the previous generation, and hence these packets are not needed to be recomputed, and hence not placed into the dispatch-queue. As explained above, the genetic operations include sorting the chromosomes with valid cost (i.e. packets that finished execution in the HW, and were not ejected due to deadlock) in order of their fitness, applying cross-over using a roulette-wheel selection strategy on them, applying elitism and mutation, and thereby deriving the new generation.

2. The application is packet-dropping aware, so that whenever a new generation of a population is derived, it is derived based solely on the completed packets from the parent generation. Under the most drastic packet dropping circumstances, the algorithm can produce new generations if there is at least one completed packet in the first generation, and none later on, based solely on the first completed packet which becomes the sole elite solution, acting as both parents for all subsequent generations. Of course, this is not useful, as the algorithm does not progress. But this means, the application is quite robust. However, this mode of operation will severely reduce the diversity of the population, and even if the HW becomes less congested and able to finish packet computations later on, the subsequent evolution will show a population-bottleneck effect, likely resulting in lower quality of result.
3. In order to solve the above-mentioned problem, i.e. to avoid population bottlenecks, it is possible to enforce through a user-defined parameter (SUCC_PERC) a minimum size of the parent population required to derive the next generation from. If this size is not reached because of excessive packet-dropping due to deadlocks in the hardware, then the ejected packets are placed into the dispatch-queue once again without advancing the generation of the present population.
4. It should also be noted that the algorithm does nothing special to take care of any packets with unusually long latency. As long as the populations have enough number of packets, the last packets to finish execution in every population do so in sequence of their start time. Thus, the new generation of a population is derived only after all the packets of the same population for the previous generation have finished executing (either successfully or by being ejected due to deadlock).

One key feature for ensuring optimal use of the hardware is to be able to adapt the polling frequency of the Dispatcher thread, and thereby the hardware input throughput, to the hardware load conditions. As a rule, the loading is very difficult to evaluate accurately, but the rate of packet dropping/ejection from the hardware provides a rough guideline. The aim is to maintain a user specified low rate of packet dropping. This may entail in varying the throughput over the execution time of the algorithm, because it is directly related to the average length of the trajectories in the populations, as well as hardware congestion inside the accelerator. Depending on the map data and the optimization problem, this average may increase or decrease gradually.

Implementation of Timing

Only the Dispatcher thread has been made directly timing-sensitive. The other two threads co-ordinate themselves with this thread through the availability of their respective input data. The hardware accelerator model is also called from inside the Dispatcher thread in order to synchronize its timing, thus effectively incorporating the hardware in simulation into the Dispatcher thread.

1) Whenever an FTR packet is dispatched to the hardware, it is assigned a dispatch-time. This is done using the present value of the dispatch-rate and the dispatch-time of the previous packet. For example, if we assume a uniform dispatch-rate, where one packet is dispatched every C hardware clock periods, then the first packet has a dispatch time = 0, the second packet = C, the third packet = 2C, etc. In general, however, the dispatch-rate may vary over the period of execution as described in Section 4.3.5.
2) The HW module calculates the trajectory cost for the packet and adds a processing time, depending on the number of map segments visited and number of via-points, to the dispatch time to estimate a finish-time. The packet is then put into an output reordering queue, which is ordered by the finish time of the packets. The HW processing time is modelled as:

```
BASE_PROC_TIME*number_of_segments_visited +
VIA_PROC_TIME*(number_of_vias-2)
```

3) Similar to the dispatch-time above, this thread also keeps track of an eject-time, based on the EJECT_PER parameter providing the period as described in Section 4.3.4. At the end of each period of this length, some packets are transferred from the output reordering queue to the hardware output queue, depending on their finish time-stamp. Therefore the granularity of the output packet timing is given by EJECT_PER. See Section 4.3.4 for more details.

4) Towards the end of the simulation, when no new packets are dispatched any more, the remaining finished packets in the reordering queue are simply flushed out.

5) The dispatch-time of the first packet of each population and generation is dumped in a file (timing_summary.log) during the simulation. Let us call it the dispatch time of the corresponding population and generation. The latest finish-time among all the packets in each population and generation is also dumped in the same file, and let us call it the finish-time of the corresponding population and generation.

6) For each population, the gap between the finish-time of one generation and the dispatch time of the next generation represents the time available to the application from the hardware output queue to the dispatcher module, to compute the chromosomes in the new generation. This time is expected to be dominated by the "genetic operator" module. If the computation cannot be finished within this time, it leads to an underutilization of the hardware, but no catastrophic failure. So, it can be visualized as a soft real time scheduling problem. The scheduling, at its most basic, according to the scheme described on Figure 13, is a non-preemptive scheme, where one of the populations ready for the genetic operations can be chosen when the "genetic operator" thread is idle, and when a population is chosen, its computation of the next generation is not preempted. However, more sophisticated scheduling schemes may be used if necessary.

4.3.3 Potential Simulation Deadlock and Resolution

The hardware model – having a notion of timing and emulating a parallel system – needs to synchronize its inputs and outputs in correct temporal sequence, thus the production of output packets has a temporal dependency on the availability of **unrelated** input packets. It means, that the availability of the last output packet of any population in the output queue depends on the availability of subsequent input packets till the former has been put into the hardware output queue. At the population level, the finish-time of computation usually maintains the original ordering. However, it is not guaranteed to do that, e.g. when there are small populations with widely varying trajectory lengths. The "gatherer" thread maintains input order in its output. The "genetic operator" thread, however, reads its inputs in a specified order and produces the outputs in the same order. Hence, it does not process any population, even if it is ready, unless it is its turn. This restriction makes it simple to implement the routine, but introduces a small probability of deadlock in *simulation* (but never in actual application). This happens when the "genetic operator" thread is waiting for a specific population to end executing in the hardware, while the hardware model consumes all other available packets in the dispatch queue but cannot produce the output for all the packets, that the "genetic operator" is waiting for. Then the simulation fails to proceed in the HW model because it needs more inputs to take the simulation forward, and the "genetic operator" is waiting for the HW to finish computation for a specific population. The "genetic operator" being an untimed sequential program, fails to notify the HW-thread that this is the case. In such cases, the simulation runs into a deadlock.

The deadlock is guaranteed to occur when there is only 1 population, which could be handled by specifically identifying that case. However, it may happen with multiple populations too, when the population size is small, there are few populations and there are a few very long trajectories in one

population, which take too long to finish execution before the HW consumes all available input packets (also those for other populations). This is a rare scenario, but cannot be ruled out. In order to solve this problem, a deadlock detection has been implemented in the HW model, based on a time-out. Whenever a simulation deadlock is detected, the simulation time is put forward till the time when the next output is due. This produces, at least one output without waiting for an input, and with any luck it breaks the deadlock. If not, the deadlock detection and resolution is performed again. But this is a purely simulation problem. In real HW, the time progresses with the clock, not depending on the input. So, there will be no such deadlocks.

It should be noted that adding too many software I/O operations (e.g. calling printf()) in the Genetic Operator thread slows it down, in turn starving the hardware model. This can also lead to deadlock timeout. This does not affect the quality of the results achieved, but it makes the simulation timings diverge from what would be expected in the actual system behavior, where such deadlocks can never occur. Thus when running such timing sensitive simulations, an eye needs to be kept on whether too many unexpected deadlock timeouts are occurring (messages are printed on the terminal to notify, if they do). If they do, unnecessary I/O's should be disabled.

4.3.4 Hardware Modeling – Packet Dropping Rate

One of the defining characteristics of the hardware being developed is that it may eject some packets before completing computation on them, if a hardware deadlock is detected. Based on the hardware simulation results, described in Section 5.2, the ejection decisions have been implemented in the following way in the high-level hardware model. It is based on a user-defined temporal granularity (EJECT_PER), e.g. equal to the initial packet dispatch period. The probability of a packet being ejected over this period of time is based on the number of packets active in the hardware during this time – which is taken to be approximately constant. The probability is modeled as a linear approximation to an S-curve as shown in Figure 14. The idea is to check the number of active packets in the hardware after each interval of an EJECT_PER, and find out the ejection probability for that. Then it is decided whether a packet is to be ejected or not based on a uniform random number generated. If yes, then an active packet is chosen at random from the

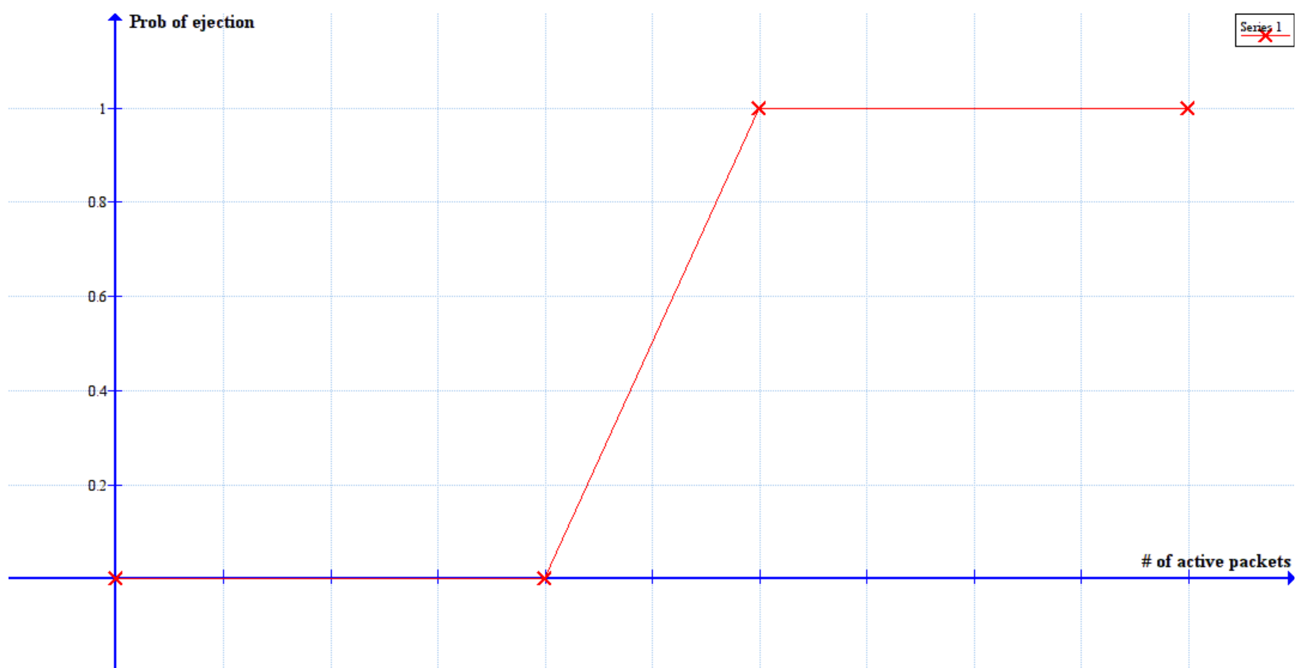


Figure 14: Packet ejection probability over a user-defined time interval

"output reordering queue" and ejected. Then the process is repeated – without changing the simulation time-step – with the number of active packets reduced by 1, until one of the trials decides in favor of not ejecting any packet. This way, there is a chance of ejecting multiple packets, and the number of active packets is never allowed to go above EJECT_THR1. It also ensures that there is no ejection if the number of active packets is below EJECT_THR0. This, however, does not model the "bunch ejection" phenomenon (Ref. Section 5.2) observed in the hardware under high congestion very closely, and thus is accurate only under low and moderate congestion conditions. This is, however, reasonable when the application implements a "Dispatch rate adaptation" strategy as explained below in Section 4.3.5.

4.3.5 HW and SW Modeling – Dispatch Rate Adaptation

As has been described in Section 5.2, the behavior of the hardware towards the end of the application run, when the population is converging, is very much dependent on the exact trajectory, that the solutions are converging towards. This may be modeled as changing EJECT_THR0 and EJECT_THR1 accordingly over the life of the application run. The exact nature of this change is, however, dependent on a few mutually opposing trends:

1. When the solutions converge to an optimal trajectory, they are often more regular on an average than at the beginning of the simulation, leading to fewer via-points and consequently smaller packet sizes. Thus the hardware FIFO's may tend to be able to accommodate more active packets. This translates to higher EJECT_THR0 and EJECT_THR1.
2. Due to the increasing localization of the paths, that the packets follow inside the HW, congestion in the relevant parts of the hardware may actually increase, correspondingly reducing the maximum packet throughput. However, for a highly folded map, the change is expected to be small, because the folding is expected to spread the load over to the whole processing array.
3. In some cases, however, depending on the exact nature of the trajectory, that the solutions are converging towards, some of the processing nodes may experience elevated congestion (essentially creating performance bottle-necks), leading to decrease in EJECT_THR0 and EJECT_THR1.

It should be additionally noted, that even if the thresholds do not change appreciably, the packet through-put supported by the hardware may change due to the change in the average length of the trajectories, leading to a change in their latency inside the hardware. Predicting these changes, however, would require a detailed model of the hardware, like the SystemC model described in Chapter 2. But that would be computationally rather inefficient when it comes to evaluating the application. This observation implies that it is important for the application to be able to change the dispatch rate to dynamically adapt to this changing congestion scenario, even if it cannot be predicted. If the congestion increases – e.g. because of the development of some performance bottle-neck – the dispatch rate should be relaxed accordingly, in order to avoid too many packet ejections leading to worse quality of result. On the other hand, if the congestion decreases – e.g. because of average length of the trajectories in the population becoming shorter – the dispatch rate may be increased in order to fully exploit the HW resources and decrease the application run-time – of course, assuming that the SW implementation of the application, i.e. mainly the execution of the genetic operators to derive new generations – can keep pace. Thus, the application is guaranteed to optimize the HW usage, irrespective of whether EJECT_THR0 and EJECT_THR1 change during the execution or not. Hence, the hard task of modeling this change can be foregone.

The adaptation algorithm is based on the following parameters:

`DISP_ADP_PER` = number of packets, in interval of which adaptation is made – time granularity
`MIN_EJ_RATIO, MAX_EJ_RATIO` = min and max of the range of proportion of packets dropped
`RELAX_RATE, TIGHT_RATE` = define how quickly dispatch rate is relaxed or tightened
Defining, `targ_no_eject` = mean of the max and min number of ejections allowed within `DISP_ADP_PER` packets, `no_eject` = actual no of ejection during one DISP_ADP_PER interval, the adaptation equations used are:

```
dispatch_per = dispatch_per * (1.0 + (RELAX_RATE * (no_eject-targ_no_eject) / (float) (DISP_ADP_PER - targ_no_eject))) // If too high an ejection rate is observed
```

```
dispatch_per = dispatch_per * (1.0 - (TIGHT_RATE * (targ_no_eject-no_eject) / (float) targ_no_eject)) // If too low an ejection rate is observed
```

It is important to choose a `MIN_EJ_RATE > 0` in order to be able to load the hardware optimally. If it is taken to be 0 or less, then the application can never detect an under-loading of the hardware.

4.4 Conclusion

Based on the findings described in the previous chapter, it was understood that trajectory optimization algorithms have many important applications. Thus, two such algorithms, employing the popular heuristics of SA and GA, were designed for evaluation. In order to exploit the hardware under development optimally, they were needed to be parallelized. As it is explained in the subsequent Section 5.1 however, the parallel GA implementation using multiple populations and migration strategy was found to be the better choice, and was optimized to perfectly match the hardware capabilities, as explained in this chapter. This is designed to accommodate the packet ejections in the hardware, as well as to optimally load the hardware for any point-to-point 2-D trajectory optimization problem. In the next chapter these capabilities of the application are tested by simulating their performance on some sample test-cases.

Chapter 5

Application: Simulation and Evaluation

5.1 Comparison of SA and GA

The two application models tried out were the one using simulated annealing as described in Section 4.2 and the one using genetic algorithm as described in Section 4.3. To start with, on a simple serial implementation of both the algorithms was made and tested on a simple problem, where the map is divided into two regions with different but constant local costs, and the trajectory with the smallest cost has to be determined between two points lying in these two regions:

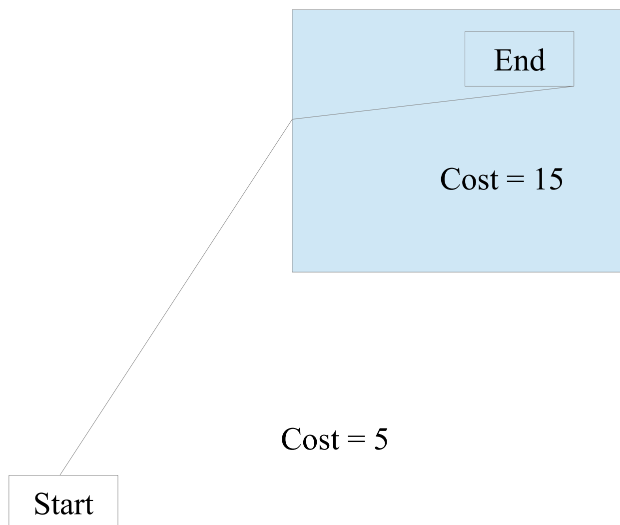


Figure 15: The problem and solution of determining the trajectory with lowest cost between two sections with different but constant costs.

It was observed that both SA and GA using applications could easily solve such simple problems. However, as described in Section 3.3.2, the asynchronous parallel implementation of SA is not expected to perform very well. On the other hand, a proven GA implementation from [4] was found to be reasonably close to the present problem, and hence adapted. It was also possible to parallelize it using the advanced “Multiple population coarse grained” as explained in Section 3.3.1, and exploit the hardware under development quite efficiently, as described in Section 4.3. For any implementation of SA, it is also important that all the packets finish evaluation, which is not guaranteed by the present hardware proposal. By the very nature of the algorithm itself, GA can, however, tolerate some amount of packet dropping. Therefore, it was only the application using genetic algorithm, that was developed in all its details, and will be considered for all subsequent evaluations.

5.2 SystemC Simulation

In order to develop the high-level hardware model described in Chapter 4, it was necessary to understand the hardware behavior throughout the execution of the application for different optimization problems, especially to understand the loading profile as well as packet ejection probability from the hardware. These data were particularly needed to take the modeling decisions described in Sections 4.3.4 and 4.3.5.

The SystemC model as described in Chapter 2 was used to find a suitable abstraction of the deadlock probability and contribution of congestion to the packet delay and to investigate how well-suited the simulation results with random trajectories as described in [11] are to the GA optimization application, developed in the present work. Here are the key findings:

1. Once the GA starts to converge, the behavior of the hardware becomes pretty erratic – in the sense that it strongly depends on the nature of the trajectory, the solution is converging towards. Assuming that the final trajectories won't be too irregular, only 4-via-point trajectories were investigated. It was observed that, everything else remaining same (including the start and end-points), the HW behavior starts to depend on the exact trajectory a lot, in a way that is not very easy to understand or model. Consider the following data for trajectories from (8.5, 5.5) → (181.5, 188.2) on a 190x190 map using a 10x10 processing array injecting one packet every 750 clock cycles. The two middle via points are each chosen randomly from a map segment, that remains unchanged for the whole population – this emulates a converging population.

   1. Trajectories (8.5, 5.5) → (71.x, 134.x) → (102.x, 125.x) → (181.5, 188.2):
      - 42% packets are ejected due to deadlock
      - Trajectories cross ~375 map segments (=> latency contribution = 26 Kclocks)
      - Avg latency of finished packets = 55 Kclocks => avg latency due to congestion = 29 Kclocks
   2. Trajectories (8.5, 5.5) → (180.x, 97.x) → (114.x, 69.x) → (181.5, 188.2):
      - 85% packets are ejected
      - ~545 segments crossed (=> latency contribution = 38 Kclocks)
      - Avg latency of finished packets = 59 Kclocks => avg latency due to congestion = 21 Kclocks.
   3. Trajectories (8.5, 5.5) → (126.x, 24.x) → (71.x, 165.x) → (181.5, 188.2):
      - ~0% packets ejected
      - ~465 segments crossed (=> latency contribution = 33 Kclocks)
      - Avg latency of finished packets = 56 Kclocks => avg latency due to congestion = 23 Kclocks

   This means that the congestion (expected to show up as ejection and delay in packet processing) is not monotonically related to how many map segments the trajectories pass through on an average, as was originally believed it would. It was expected that the high degree of folding (19x19) would take care of the load distribution. But it seems it could not. In fact, the load distribution is quite uneven. The mean absolute difference of loading among all processing nodes is 70%, 27% and 23% of the average processing node load respectively for the 3 cases above (as opposed to <2% for fully random trajectories). Also interestingly, the most unevenly distributed load did not lead to the highest instance of deadlocks! Thus, the deadlock probability could not be correlated to the average length of the trajectories.

2. Simulation results in [11] use random 3-via point paths. These results are too optimistic even for the starting random population for the present problem, even if only 3-via point paths are assumed, because, unlike in [11], the start and end points are not random. This problem will likely be less pronounced if there are more via points, though – which is very likely. So, it is not a dramatic failure, but the data for [11] cannot be used directly.

3. The system seemed to be leading to serial ejections, i.e. the ejections seemed to be bunched together, bringing down the system loading quite a bit intermittently. Observe the plot in Figure 16 with 4-via point trajectories, simulating the converging GA case. Following are the details of the test-case:

   Trajectories: (8.5, 5.5) → (91.x, 170.x) → (124.x, 136.x) → (181.5, 188.2)

Packet through-put = 750 clocks/packet
Rate of packet ejection = 41% of the injected packets.
It should be noted that while around 60 packets are normally active inside the HW at any time, sometimes it comes down to as low as 20 packets, clearly because of the bunched ejections.

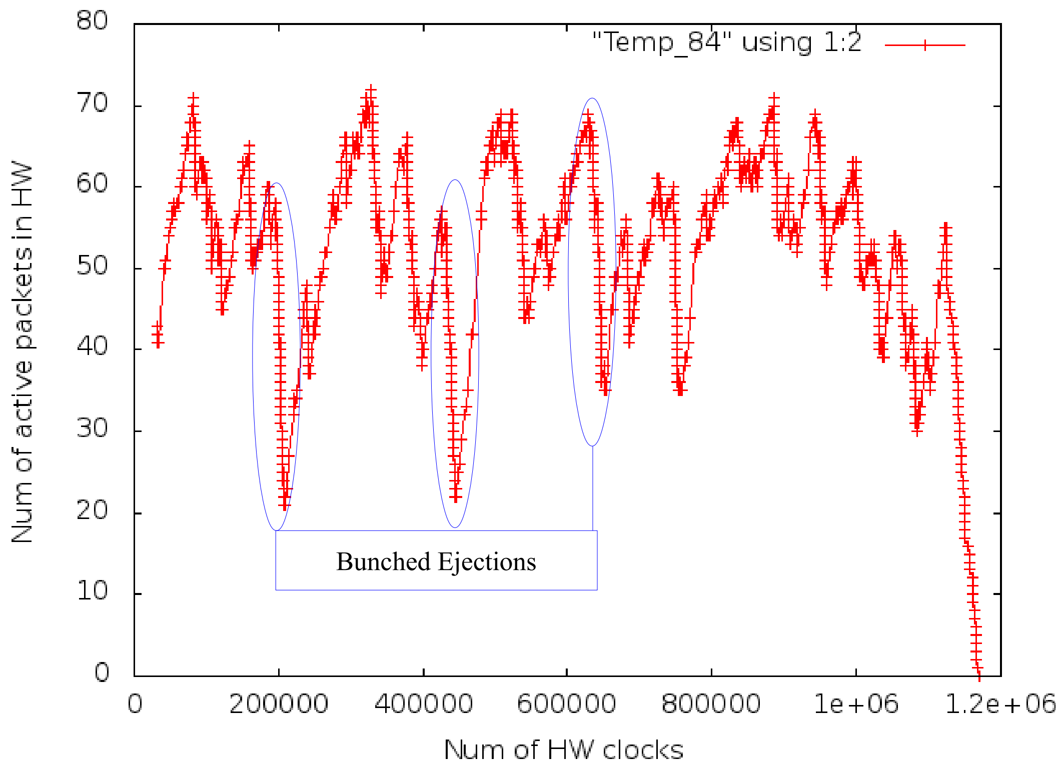


Figure 16: Example of instantaneous number of active packets in HW as derived from SystemC model simulation – high congestion case

4. One promising metric found for estimating the ejection probability is the number of active packets in the HW. As would be apparent from Figure 16, the number of active packets hardly ever goes beyond 70 packets (the HW contains total of 400 packets' slots in the connecting FIFO's between the processing nodes. So, it is 17.5% of the total). With a 3-via-point case, peak FIFO-occupancy was observed slightly above 25%. It is hard to explain why it is more for the 3-via point case, but normally with 4 via-point cases, the FIFO occupancy seemed to be more or less maxed at 20%. This upper-limit has been modeled in the high-level hardware model as the threshold EJECT_THR1 in Section 4.3.4. It is also apparent that when the number of active packets is below a threshold, EJECT_THR0 from Section 4.3.4, packet ejects are very rare or non-existent, as can be observed on Figure 17 for the following simulation run, where the number of active packets remains around 50 during the simulation leading to 0 packet ejection. Here are the details of the test-case:

   Trajectories: (8.5, 5.5) → (150.x, 189.x) → (170.x, 184.x) → (181.5, 188.2)
   Packet through-put = 750 clocks/packet
   Rate of packet ejection = 0.
   Thus these simulation results provide empirical support for the packet ejection probability, as modeled at a high-level and described in Section 4.3.4.

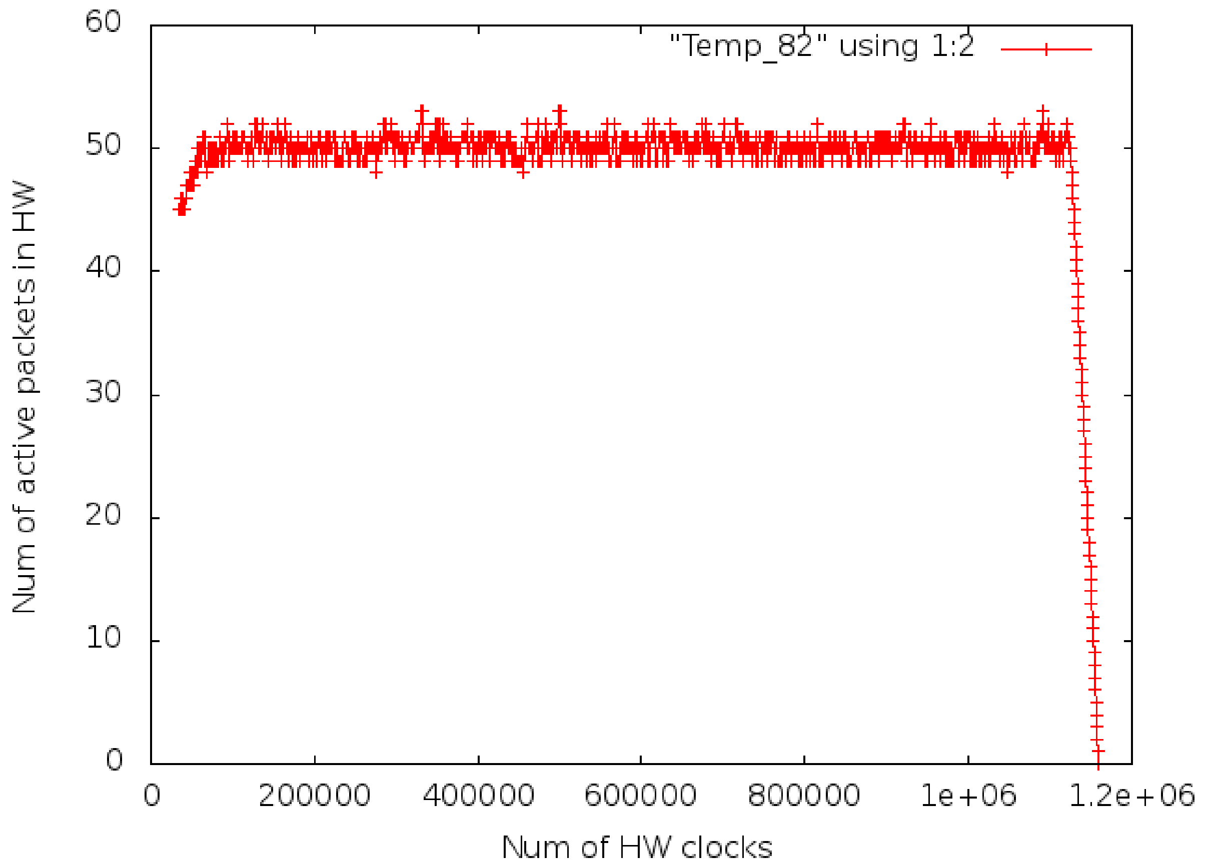


Figure 17: Example of instantaneous number of active packets in HW as derived from SystemC model simulation – Low congestion case

5.3 Influence of Packet-dropping

Simulations were run on the high level C model to understand the effect of packet dropping on the performance of the GA algorithm. The test-case used was as shown in Figure 18, where the trajectory with the lowest cost between the start and end points needs to be determined. The optimal trajectory has to take quite a long detour, resulting in a relatively complicated shape. The problem

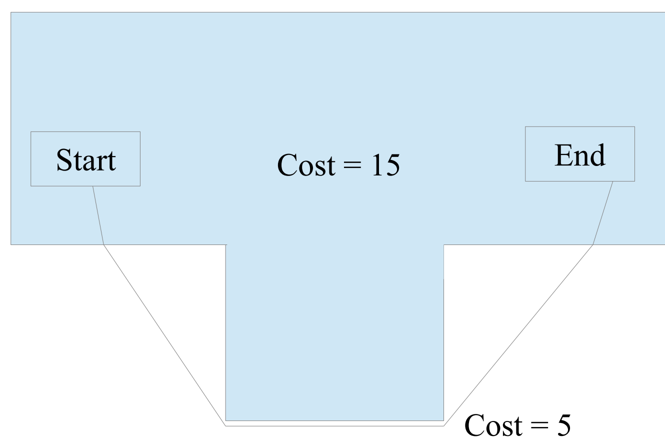


Figure 18: Sample problem for evaluation of GA application and its solution.

also has an alternative local minimum, given by the direct straight-line path between the start and end-points. Also, in order to study the effects of packet dropping, the hardware model was taken to implement a uniform rate of packet-dropping, i.e. it ejected packets with a user-defined constant probability, rather than using the strategy described in Section 4.3.4. So, it was a slightly non-standard hardware model. Also, the dispatch-rate adaptation (Section 4.3.5) was not enabled, because it would have no effect on the quality of results, but could slow down simulation to account for high packet ejection rate, that was used for some of the simulation runs. Also, re-computation of ejected packets was not enabled, i.e. the parameter SUCC_PERC, explained in Section 4.3.2, was effectively set to 0. A selection of the simulation results have been tabulated in Appendix B, and used here to plot the next three graphs, which compare under different conditions the convergence behavior of the algorithm, given by how close the best solution found by it after any specific number of iterations (or equivalently, generations) is to the known best solution.

In general, the algorithm appeared to be quite robust. The convergence hardly breaks down, even under very severe packet dropping – tested till a packet dropping probability of 0.9, i.e. when only about 10% of the packets finish execution successfully on an average. The following graphs show the performance of the algorithm, solving the problem in Figure 18. The trajectory with the globally minimum cost has a cost of 168.5, while the other local minimum, given by the straight-line trajectory is 180. The error bars in the graphs represent ±σ (i.e. standard deviation of the solution costs). The graphs show the results of only those runs which converged towards the global minimum (168.5), instead of the other local minimum (180).

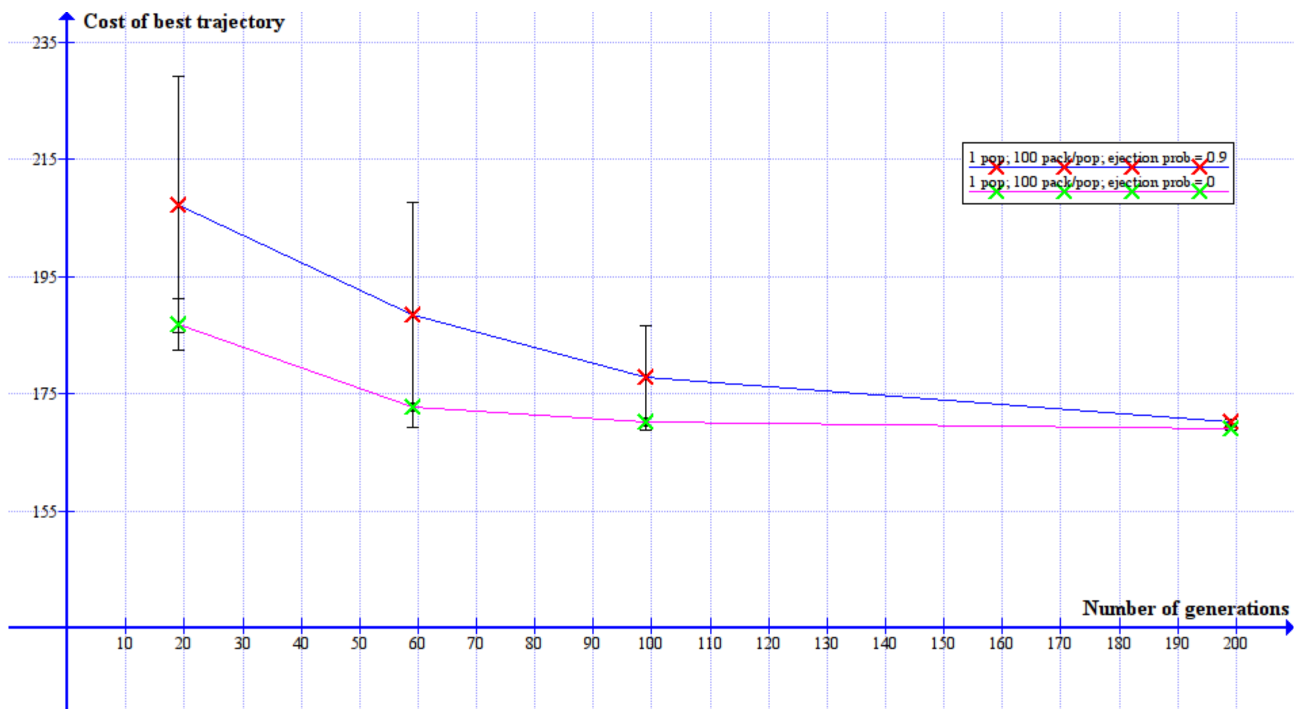


Figure 19: Cases with only 1 population of 100 packets, and ejection probabilities of 0.0 and 0.9

The results from this simulation suggests that the effect of packet dropping does not affect the result of the algorithm in the long run, but it deteriorates its consistency, as well as the rate of convergence. The next graph (Figure 20) also upholds this view, but it also suggests that using 2 populations of smaller sizes rather than one population of double the size makes the consistency of the results less dependent of packet-dropping rate. Figure 21 shows, however, that the quality of result and the rate of convergence do not seem to be strongly dependent on whether one or multiple propulations are used.

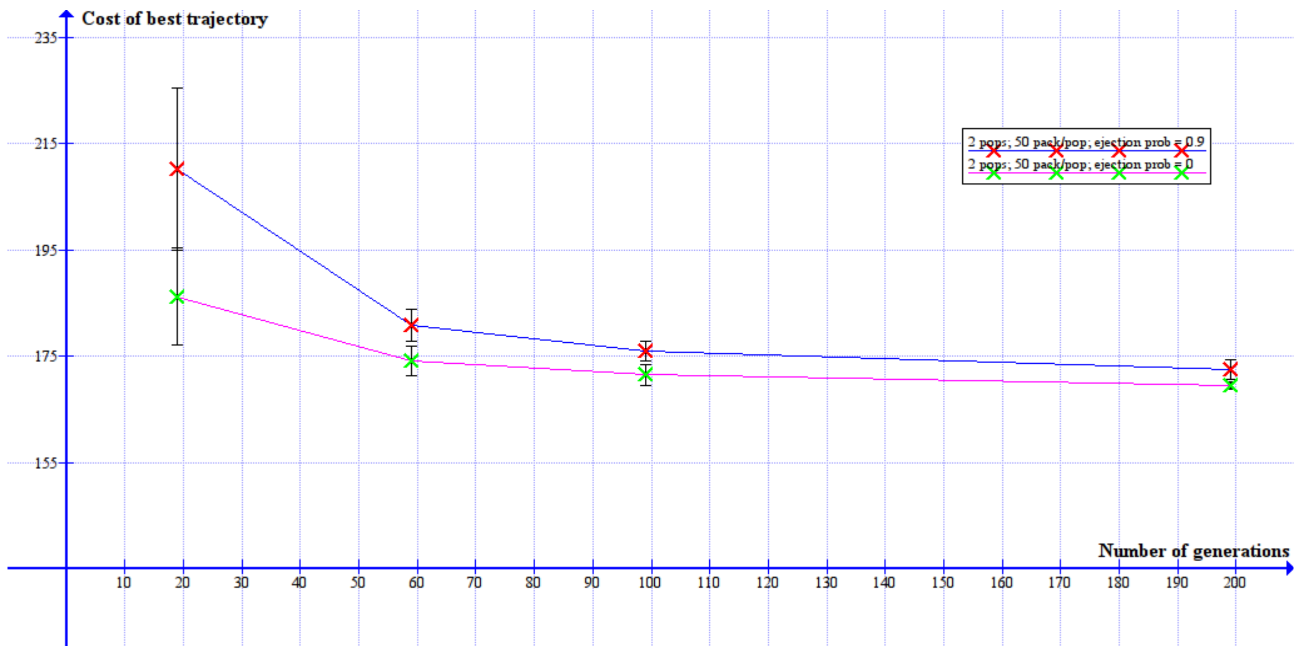


Figure 20: Cases with 2 populations of 50 packets each, and ejection probabilities of 0.0 and 0.9

Using the same set of data as in the previous two curves, but now showing the performance comparison of using 1 big population of 100 packets versus 2 smaller populations of 50 packets for an ejection probability of 0.9:

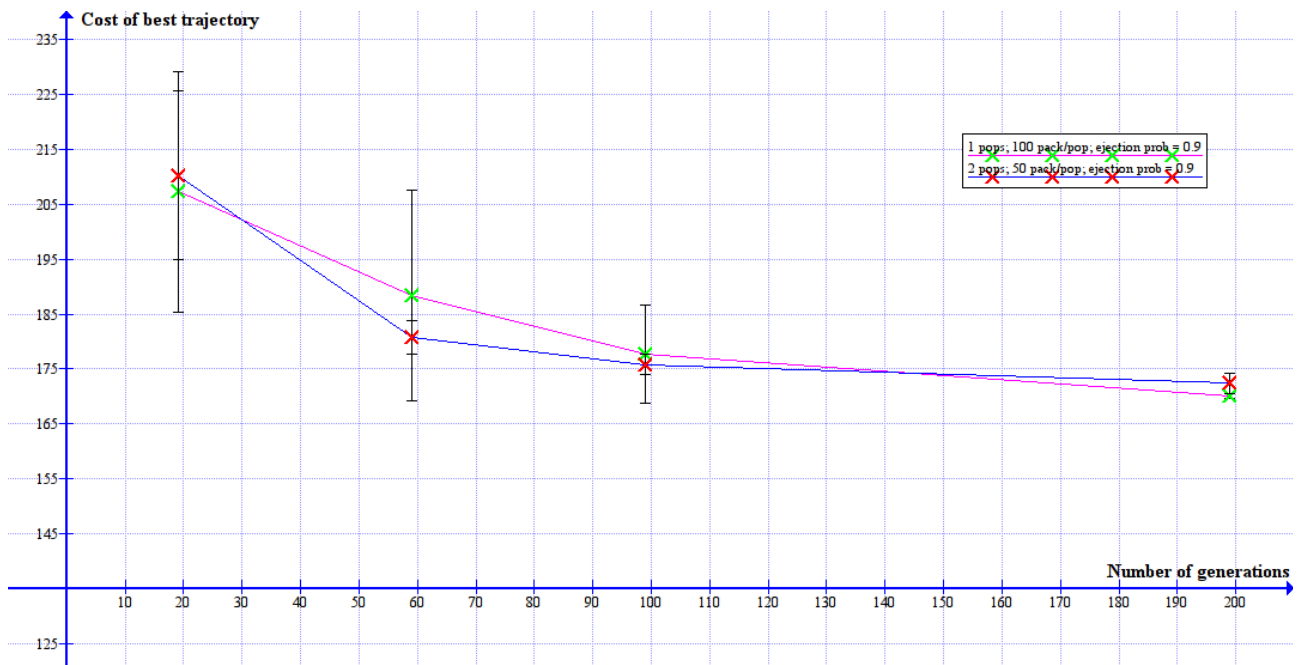


Figure 21: Comparison between 1 population of 100 packets and 2 populations of 50 packets each.

It can be concluded from these simulation runs that:

1. The application always converges, even under very severe packet dropping. This is expected because of the elitist nature of the specific GA implementation. It ensures that the solutions can never become worse from one generation to the next. However, it is possible that the convergence leads to a local minimum.
2. However, the higher the rate of packet dropping, the slower the rate of convergence.

3. Under severe packet dropping, distributed solution with multiple populations may have better convergence in the medium term than a solution where all the packets are lumped in one population. This is, however, not prominent when more packets are used in every generation.
4. One point not shown on these plots is the probability of converging at a local minimum. It was observed that smaller population sizes lead to higher probability of converging to a local minimum.
5. Thus, it can be surmised from points 3 and 4 that it is better to use multiple medium-sized populations (~100 packets each) than to use a very big single population. This is also ideal, in terms of HW usage, as big single population translates into idle time of the HW when the application is calculating the new generation for that sole population.

5.4 Influence of Dispatch Rate Adaptation

In order to evaluate the efficacy of the dispatch-rate adaptation strategy described in Section 4.3.5, the same problem in Figure 18 was solved again using the standard S-curve based packet ejection probability computation, as described in Section 4.3.4, enabled in the hardware model. When the dispatch rate adaptation is disabled, the resultant loading profile of the hardware was as follows:

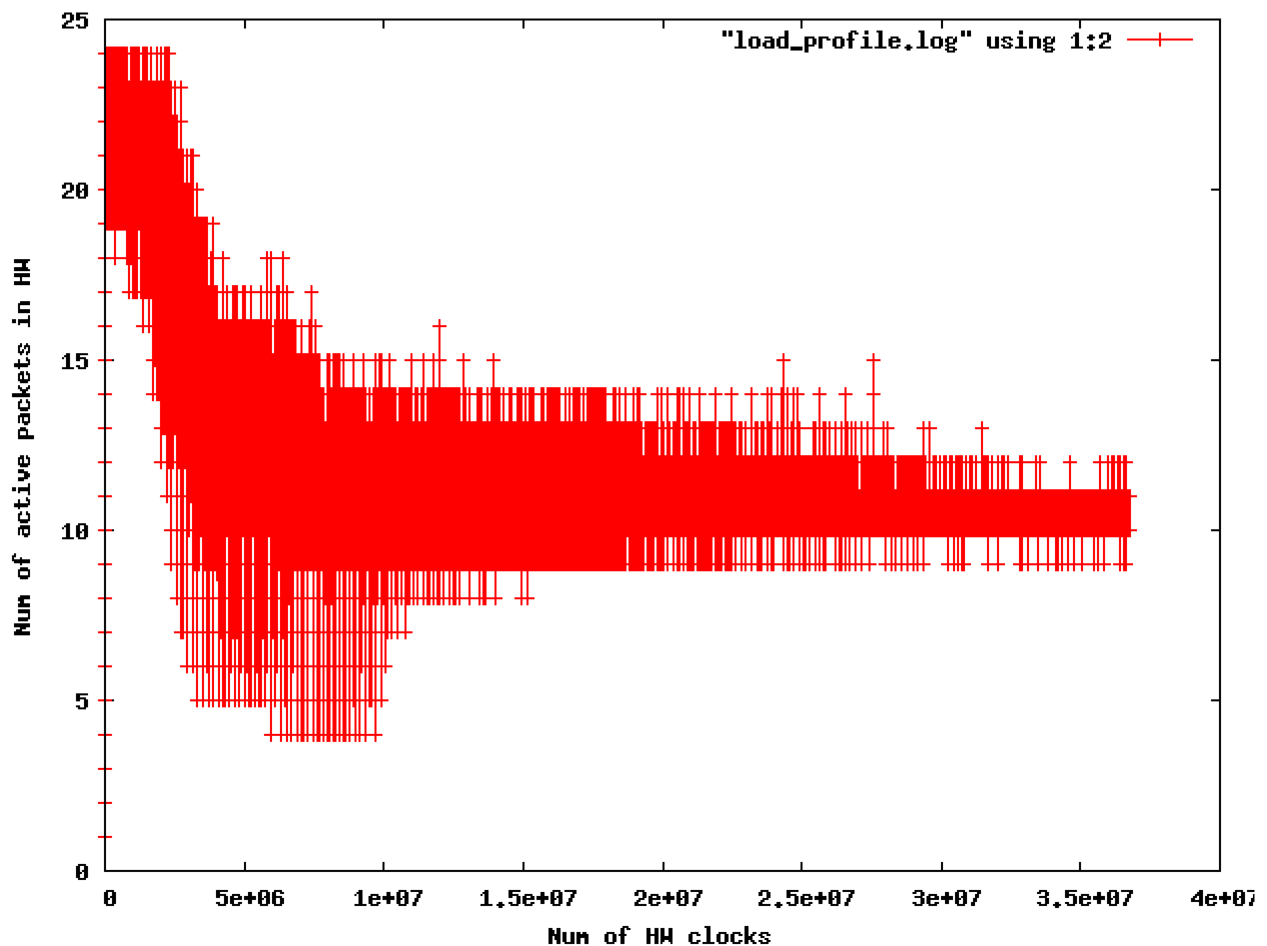


Figure 22: Hardware load profile without dispatch-rate adaptation

The HW is quite under-loaded towards the end of the execution, and it takes about 37 million clocks to finish the application run (200 iterations of GA). On the other hand, when the dispatch rate

adaptation is enabled, the resultant profiles is as follows:

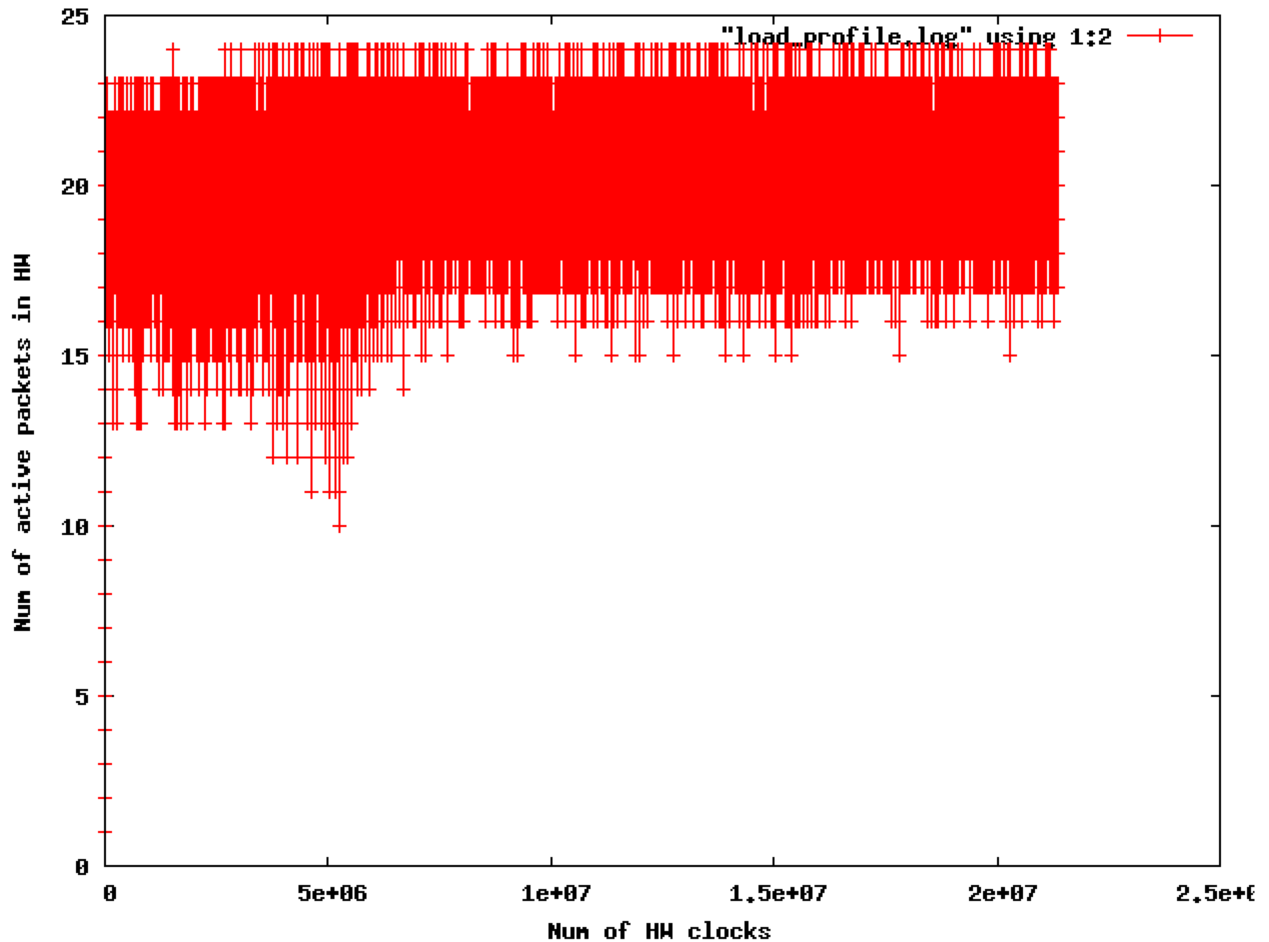


Figure 23: Hardware load profile with dispatch-rate adaptation

In this case, the hardware is quite uniformly loaded for the most part, and the application finishes is about 22 million clocks. That is 40% improvement in execution time – of course, assuming that the software can cope with it. The adaptation may even improve the quality of the results by reducing number of cases when a new generation of a population is computed from a small part of the older generation, triggered by too many dropped packets, resulting from too high a dispatch rate. It is observed to a limited extent in the simulation runs which used an initial dispatch period of 200 clocks/packet, which is same as the fixed dispatch rate in the nonadaptive runs. As the adaptive run shows, an initial dispatch period of around 340 clocks is more suitable, and the algorithm adapts quickly to that level. Initially this leads to a lot of packet ejections in the non-adaptive run, leading to somewhat lower quality, but the adaptive run minimizes such ill effects.

Figure 24 shows the adaptation of the dispatch rate over the period of simulation run. As this figure shows, the later iterations of the simulation could support a higher packet rate. The simulation assumed constant EJECT_THR0 and EJECT_THR1; yet this happened because the average length of the trajectories in the later iterations was less than that in the initial runs with completely random trajectories. This led to a lowering of the execution time for each packet in the hardware, leading to lower congestion. This is also the reason why the hardware becomes underloaded in the non-adaptive run towards the later iterations of the application, as shown by Figure 22.

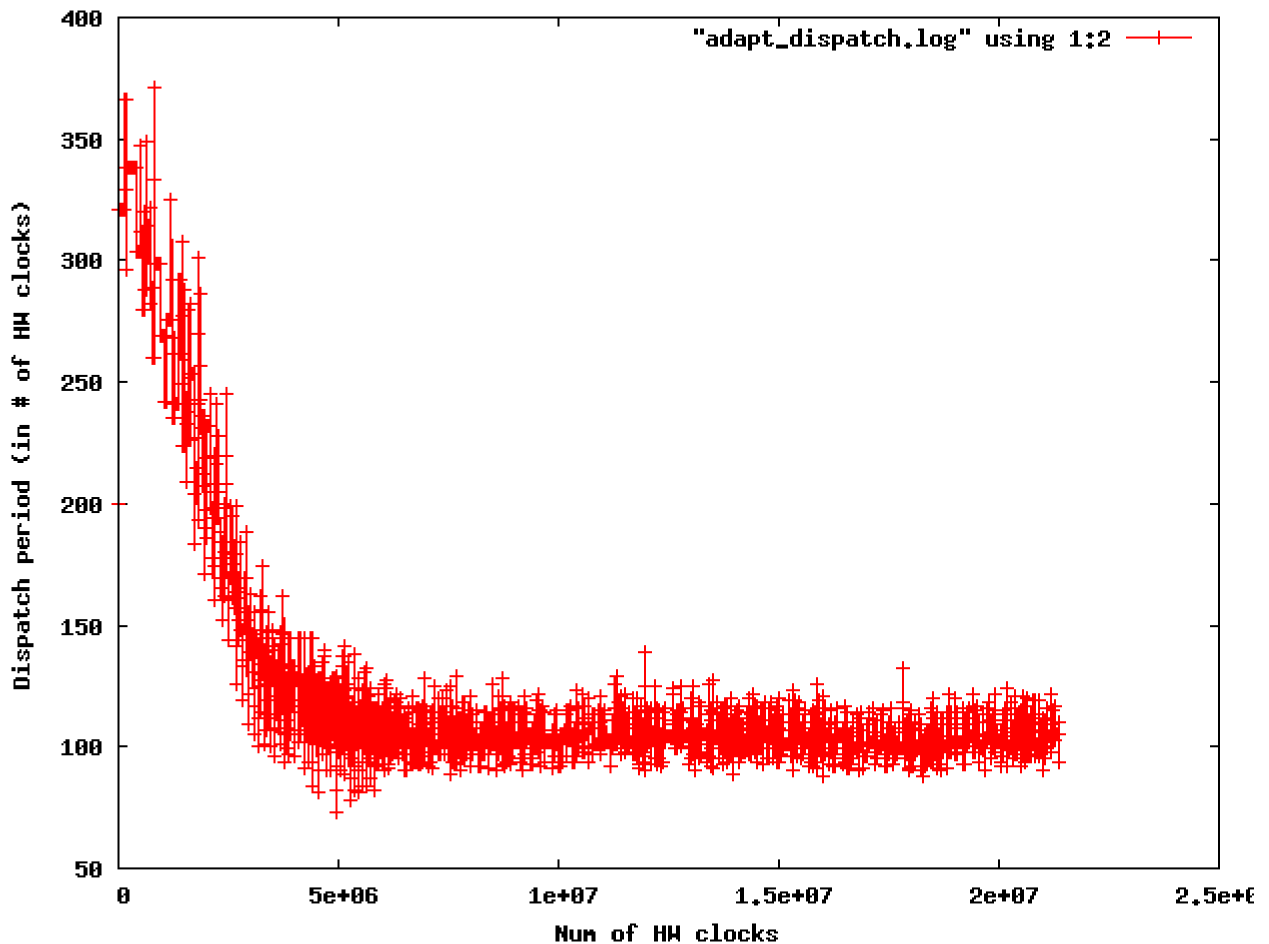


Figure 24: Adaptation of dispatch-rate

5.5 Conclusion

It can be concluded from the above-mentioned results that the hardware envisioned in Chapters 1 and 2 can indeed be used to perform multiple trajectory cost calculations in parallel as long as the application can ensure that it can feed packets into the hardware at regular intervals, and can adapt itself to the changing maximum throughput support of the hardware. It should also be able to cope with ejection of incomplete packets due to hardware deadlocks. The simulation results in Sections 5.3 and 5.4 clearly show that the parallel GA-based application designed in Chapter 4 is one perfect example application that solves a practical problem, that of optimizing trajectories between two points on a 2-D map, by efficiently exploiting the capabilities of the proposed hardware. As is clear from the discussion on motion and trajectory planning Chapter 3, this is a very important problem with many applications in military as well as civilian domains.

Chapter 6

Hardware: Design – Micro-architecture

As has been demonstrated in the previous chapters, when coupled with careful application design, the hardware proposed in Chapters 1 and 2 can indeed be an excellent tool to function as a hardware accelerator. In addition, as with any NoC-based homogeneous multiprocessor system, this hardware is highly repetitive, and is essentially an array of processing nodes connected to each other through computation network (CNW) and insertion-ejection network (IENW). The basic architecture for such a processing node was already developed in [11] and mentioned in Chapter 2. This serves as the starting point for all subsequent hardware design. It should be noted that the processor(s) shown in the architecture of the processing node is a part of the processing node, and thus a part of the hardware accelerator, and must not be confused with the processor executing application, which is external to it, as has been shown on Figure 11.

The proposed processing node architecture in [11] assumed that the CNW and IENW interfaces and protocols were identical. This is, however, unlikely because of the different requirements on these two networks, and it will be clear from the subsequent discussion in this chapter. Therefore, an additional logic, called IENW node, is necessary to interface this module with the IENW. Thus, the processing node architecture that would satisfy the requirements for IENW as well as CNW, would look like below:

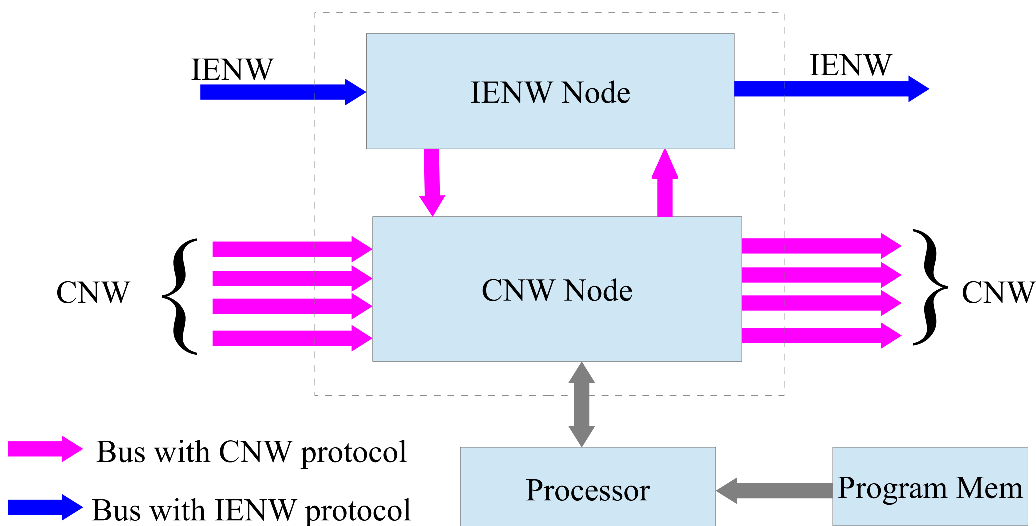


Figure 25: Micro-architecture of Processing Node including IENW node. The section inside the dotted line defines the "NoC communication infrastructure"

The following sections describe in detail the micro-architecture of the IENW node and the CNW node, as well as various relevant design considerations, including the bus protocols, and constraints on the software to be run on the processor.

6.1 CNW Node Architecture

As mentioned before, the architecture, shown on Figure 6 in Section 2.2, has been used as the base line for developing the processing node architecture minus the IENW node. The following relatively minor changes have, however, been made to this design in light of more detailed planning, yielding the CNW node architecture shown on Figure 26:

1. The receive-list has been put inside the dual port RAM, thus removing the need of extra storage elements (e.g. Flip-flop bank), as well as multiple I/O access ports of the processor.
2. In the same spirit, the send-list, which has now been included inside the send-module acknowledging the close connection between these two modules, has been made to share the same I/O interface of the processor as the dual port RAM.

This way, the processor needs only one I/O interface to interact with the rest of the processing node, as has been assumed in Figure 25.

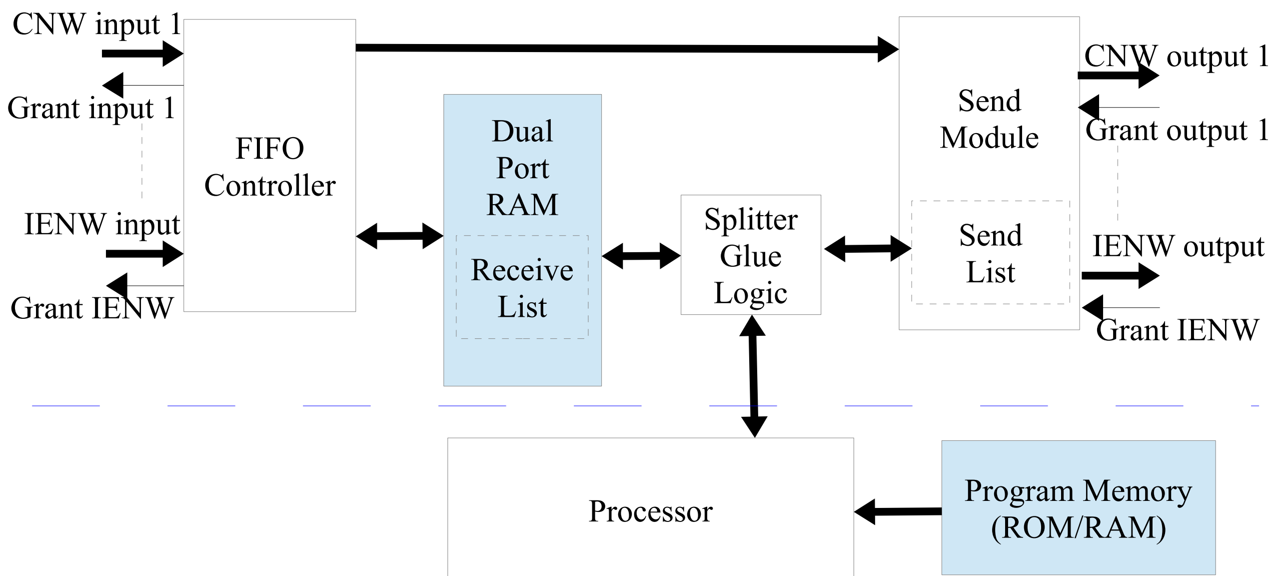


Figure 26: Micro-architecture of Processing Node except IENW. The part above blue dashed line is the CNW node.

The part of the design above the dashed blue line represents the CNW node. It is assumed that – seen from the CPU – the "send-list" is available at an address SEND_LIST_BASE, whose three LSB are 0. These 3 LSB's are mapped to the physical send-list address. SEND_LIST_BASE should be larger than the address of the last word in the DPRAM. The processor itself has not been implemented, but for some of the area and performance estimations, a Microblaze micro-controller has been instantiated as a stand-in for the processor and its program memory [15].

6.1.1 Memory Mapping

The dual port RAM has been split into 7 separate areas, as shown in Figure 27. The first 5 areas all have the same depth, fifo_depth = 2^(FIFO_DEPTH_W) words. Each of these areas is maintained by the FIFO-controller as the storage for an input FIFO, which is implemented as a circular buffer. The next area is the "receive list" which is just 5 words deep. Each of its words contains the current head-pointer of one of the input FIFO's (i.e. the memory offset from the base of that FIFO area), and a rec_flag which – when set to 1 – indicates that there is a valid packet at the location indicated by the head pointer for the processor to read. The last area in the memory serves as the main data memory of the processor.

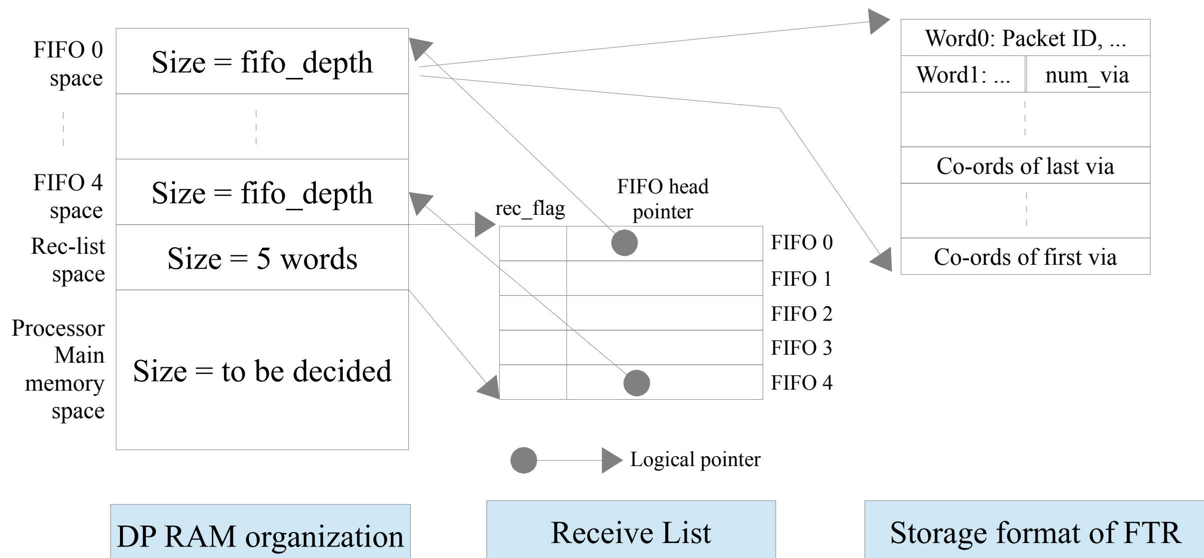


Figure 27: RAM organization and utilization

The following assumptions have been made in this design:

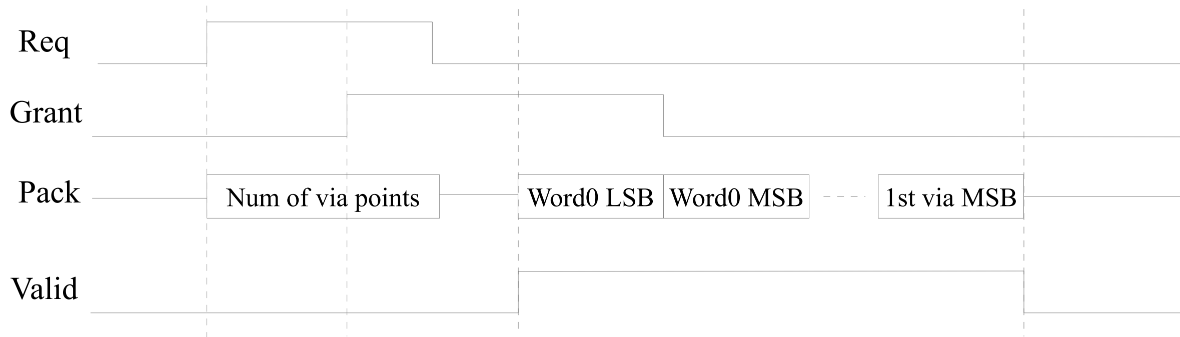
- Word size of 32 bits
- **The lower 16 bits of the 2nd word in FTR contain the number of via-points in it.** The FIFO controller uses this information to update the FIFO pointers. Hence, it is important to adhere to this requirement.
- The head pointers in receive-list are offsets within the range of that FIFO (0 to fifo_depth-1)
- The via list of an FTR packet is stored and transmitted in reverse order, i.e. the first via last, and the last via first, as shown in the figure.
- The FIFO controller has R/W access of the FIFO0 ... FIFO4 space and the Receive-list space in the memory, but no access into the processor main memory space.
- The processor has full R/W access on the processor main memory space, and it updates the intermediate data record and forward bits in the FTR's inside the FIFO space. Otherwise, it uses only read-only access to the Receive-list, as well as most words in the FIFO space. Most notably, it does **not** modify the num_via field in the packets. If it needs to do that, it is effected through the send-list as described below in Section 6.1.5.
- It is also assumed that the fifo_depth is a power of 2, and one word in the FIFO is inaccessible, i.e. the accessible depth of the FIFO's is (fifo_depth-1).

6.1.2 CNW Bus Protocol

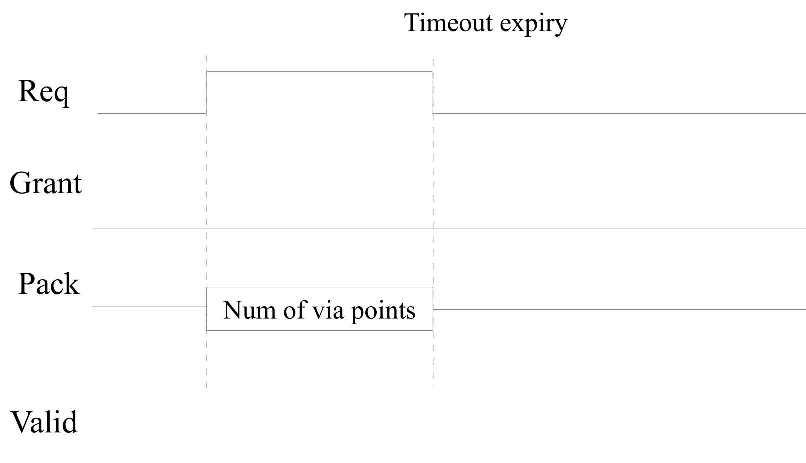
The bus protocol used by the external CNW/IENW interface on the FIFO controller is as shown below on Figure 28. This is also the over-all CNW bus protocol. Over-all IENW bus protocol is somewhat different, and thus the FIFO-controller IENW interface is connected to the IENW through a converter, as is shown on Figure 25.

The external bus protocol is synchronous, but the clock signal has been omitted from Figure 28 for the sake of simplicity. Req, Pack and Valid are signals going from upstream to downstream node. Grant goes from downstream to upstream node. The Pack nets are used to send packet words, but taken to have half the width of a word (i.e. 16 bits), as will be explained below in Section 6.1.4. This is done to ensure smooth interleaving of read and write accesses on the DPRAM, which has the same width as a word. During the transfer of a word the lower half word is transferred first,

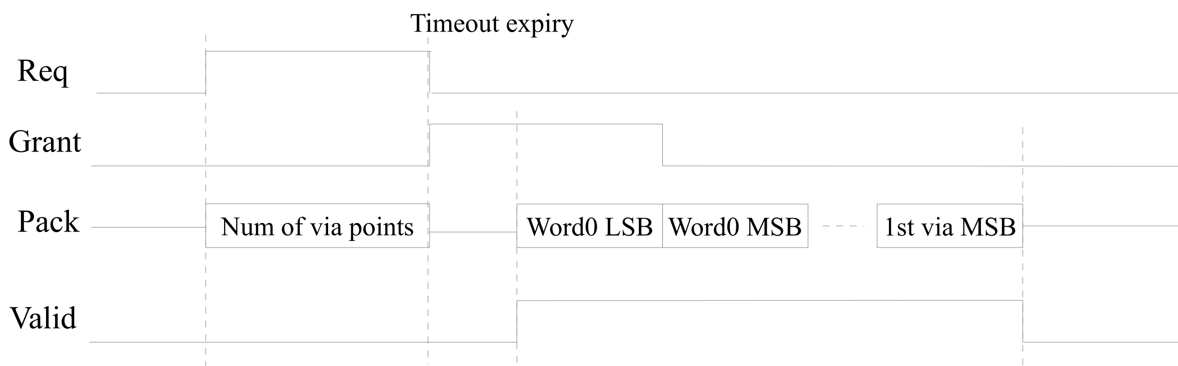
followed by the corresponding higher half word. Pack is also used to specify the number of via points in the packet about to be sent, when the upstream node asserts its Req signal. Theoretically, this imposes a restriction on the maximum number of via-points in the packet at 2^16 = 64K, but this is expected to be far beyond the expect range of this parameter.



Waveform A: Normal successful data transmission



Waveform B: Failed transmission due to timeout



Waveform C: Successful transmission: grant received just after timeout expiry (rare situation)

Figure 28: CNW bus protocol – different scenarios

The vertical dashed lines on Figure 28 represent the points of synchronizations between the different bus signals, but the number of clocks between any two of them is freely variable but >0. The duration of each packet data (marked "Word0 LSB", "Word0 MSB", ... "1st via MSB" in the

figure) is 1 clock cycle.

When a processing node (i.e. its send-module) wants to send a packet to the downstream processing node or IENW, it asserts the req signal and also sets the “pack” signals to the number of via-points in the packet it wants to send. When the destination receives this send-request, it may assert the grant if it can service it (i.e. has enough space in the FIFO to accommodate this packet in case of a node, and the bus is free in case of IENW). There is no timing requirement of how long the destination may take to assert the grant. The upstream node may, at its own discretion, however, de-assert the req, thereby terminate the send-request before receiving a grant (e.g. when the req has been high for too long, thus leading to the time-out of a deadlock detection counter). It must, however, monitor for a grant in the clock cycle just after the req de-assertion, and if it receives a grant in that cycle, it should behave as if it received it before the de-assertion of the req. The destination is not allowed to assert grant any later than this. Some time after a grant has been received, the upstream node should start transmitting the packet in the “pack” signals with the associated valid signal asserted. There is no timing requirement about the delay between the reception of grant and the assertion of valid. Also, the transfer is done in burst mode, i.e. once the transfer starts the “valid” signal may be de-asserted only after the whole packet has been transferred. At the completion of the transfer the “valid” is de-asserted. In case of a successful packet transfer, the grant and req signals may be de-asserted any time before the end of the transfer of the corresponding packet.

6.1.3 Internal Bus Protocol

The Protocol used on the processing node-internal interface between the send-module and the FIFO-controller is as shown below on Figure 29 (omitting the clock for simplicity). On this interface, the signals Req, Num_via and FIFO_no are sent from send-module to FIFO controller,

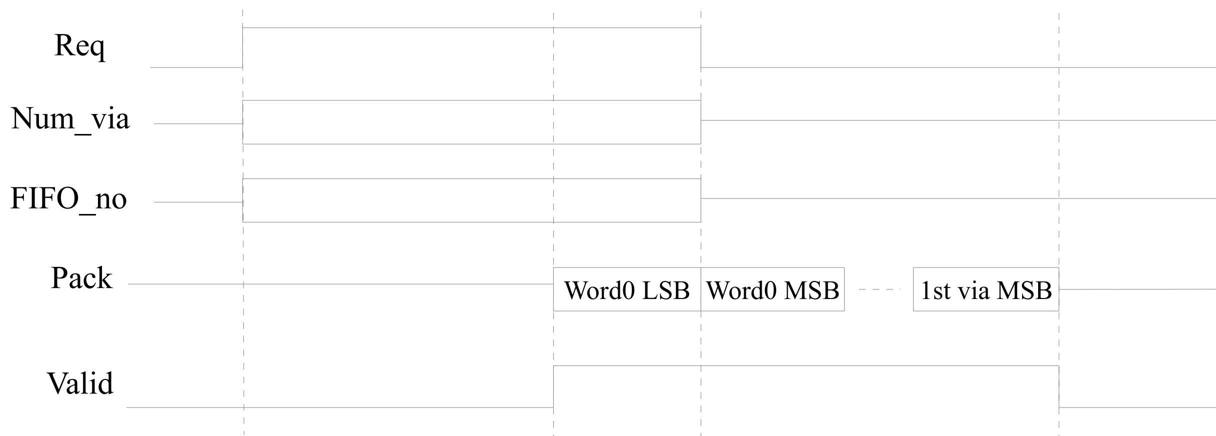


Figure 29: Bus protocol for internal bus between FIFO controller and send-module

and Pack and valid are sent from FIFO controller to send-module. When the send-module wants to request to read data from a FIFO, it asserts the req signal on this internal interface to the FIFO-controller along with specifying the FIFO number on FIFO_no signals, and the number of via-points to be read for this packet on the Num_via signal. The FIFO-controller starts sending out the data to the send-module 2 or 3 cycles later, qualified by the “valid” signal. The data itself is sent over the “pack” signals, which are half the word-width wide (i.e. 16 bits), as in the case of the external bus. Just like the external bus, the data transfer is effected in burst mode, i.e. the valid is de-asserted only after the whole packet transfer is completed. The “req” signal must be de-asserted before the data transfer is finished but after the data transfer has started. Each read-request is

guaranteed to be satisfied by the FIFO-controller within 3 clock cycles and no request is allowed to be de-asserted before the FIFO-controller has served it. It is important to note that the FIFO-controller does not only read as many via-points of a packet as requested by the send-module, it also modifies the corresponding “number of via-points” entry in the FTR it sends out to the send-module, so as to maintain the data integrity of the packet.

6.1.4 FIFO-Controller

The FIFO controller is basically composed of a “Write logic” and a “Read logic” block, with a MUX acting as an arbiter between the two, trying to access the DPRAM.

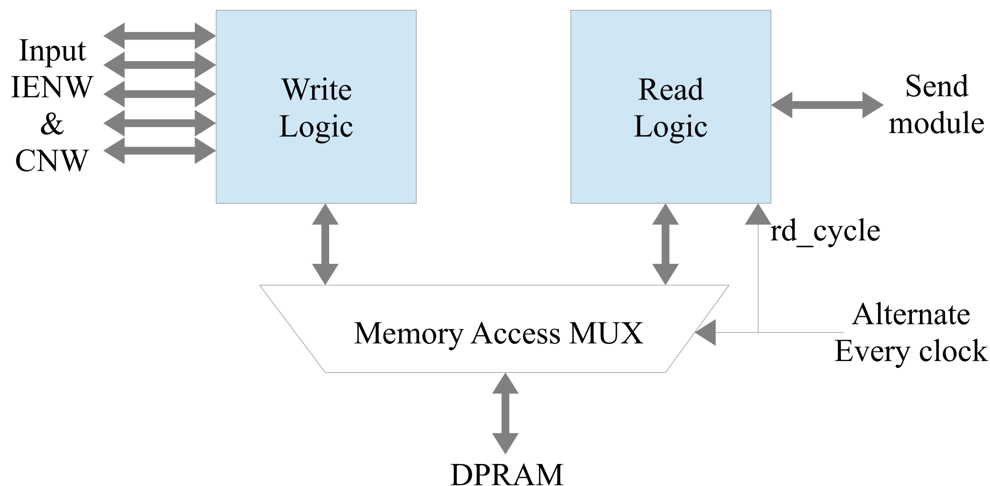


Figure 30: FIFO controller micro-architecture

This micro-architecture makes sure that the read and write accesses to the DPRAM are not blocking to each other. Otherwise, if the write logic goes into a blocking wait in order to receive new data from an upstream node, the read may also get blocked. This leads to the blocking of a shared resource (memory) and could easily lead to deadlock through circular wait, e.g. when a processing node is trying to write a packet into itself through the loop-back CNW segments at the boundary of the processing array (Ref: Section 2.1 for “loop-back”). However, in order to match the bandwidth of the CNW interfaces and the memory interface in this architecture, the **DPRAM interface has twice the data-width** (i.e. equal to the word width = 32 bits) than the IENW/CNW interfaces, which are thus half the word-width wide (i.e. 16 bits).

At the heart of the write and read logics of the FIFO-controller, there are two finite state machines (FSM's). The FSM description of the write-logic is as shown below on Figure 31. It is important to note that, this logic cannot write directly into the RAM because it is not synchronized to the memory access arbitration cycles. It writes into an intermediate parallel-in-parellel-out buffer (PIPO), called write-buffer, and this updation is visible to the memory interface MUX in the next clock cycle.

FIFO-Controller Write-Logic

On reset, the FSM is put into state 000 with i=0. In this state it checks if there is a write-request from the i-th external interface, and if so if there is enough space in the corresponding FIFO to write that packet. If either of this is false, then i is incremented modulo 5 and the state changes to 100. The next state is again 000, with a new i. In case, however, the checks at 000 are satisfied, then the next state is 001, and the corresponding grant signal is asserted to the external interface. The state

remains same till the valid is received from the same interface, in which case the next state is made 011 and the first half word of the packet received with the first valid is put into a temporary buffer. In 011 state, the grant is now de-asserted as the data transfer has already started and thus maintain the external interface protocol. At the same time, the MSB half word of the same packet word is received, and the whole word as well as the memory address where it is to be written, derived from the relevant tail pointer, is put into the write-buffer ready to be written into the memory. Then the tail pointer is advanced (modulo fifo_depth) and the state is moved always to 010, in which the valid signal is checked again to determine whether the data transfer has finished. If the valid is still high, the lower half-word of the next word is received and buffered internally, and the FSM goes back to the 011 state in order to receive the upper half-word of the same word. If valid were 0, however, signifying end of transfer, the next state is 110. In this state, the write-buffer is configured to update the receive-list entry no. i with the rec_flag set to 1 and the head-pointer set to the current head-pointer for the FIFO i. Then i is incremented modulo 5 and the next state is 100. The write-buffer is however still maintained unaltered. The next state is back to 000 signifying the end of the transaction cycle for the FSM, and the write-buffer is update to disable any memory write.

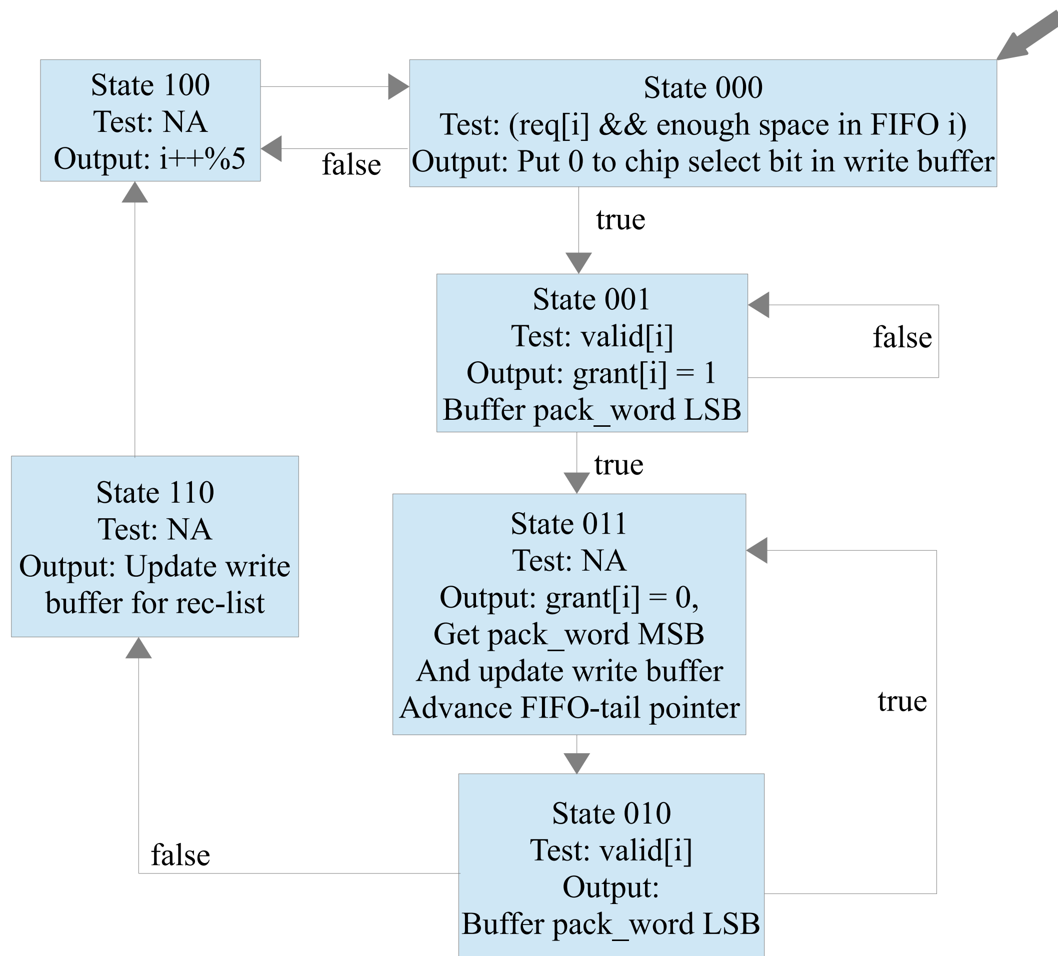


Figure 31: FIFO-controller Write-logic FSM

Note that the write-buffer remains fixed for exactly 2 consecutive clock cycles: as seen by the memory-write MUX, in states 010-011 (normal writing sequence) or 010-110 (for the last word of the packet) or 100-000 (for updation of receive list), letting the memory access MUX (Figure 30) the necessary time to actually write the relevant data into the DPRAM. This is necessary because the write-logic gets to update the memory in every alternate clock-cycle.

It should also be noted that, after each packet is written into the FIFO, the rec-flag in the receive-list

for the corresponding FIFO is made 1, and the corresponding head pointer entry is updated to the current header pointer, which may point to the middle of a packet if the same FIFO is currently being read by the read-logic in parallel. However, this is not a problem, as long as the processor does not read a packet from this FIFO before the read-logic hardware finishes its reading and updates the receive list accordingly with the correct head pointer, pointing to the beginning of a new packet. This is ensured by:

1. **Forcing the software to check the send-list to detect that the FIFO, it wants to read, is indeed available for reading by comparing "sent_toggle" and "ready_toggle" fields. (Cf. Section 6.3, Figure 42)**
2. **Designing the FIFO-controller write-logic and the send-module in a way that the updation of "sent_toggle" in send-list at the end of a FIFO read operation happens no earlier than the updation of the corresponding receive-list. (Cf. Section 6.1.6)**

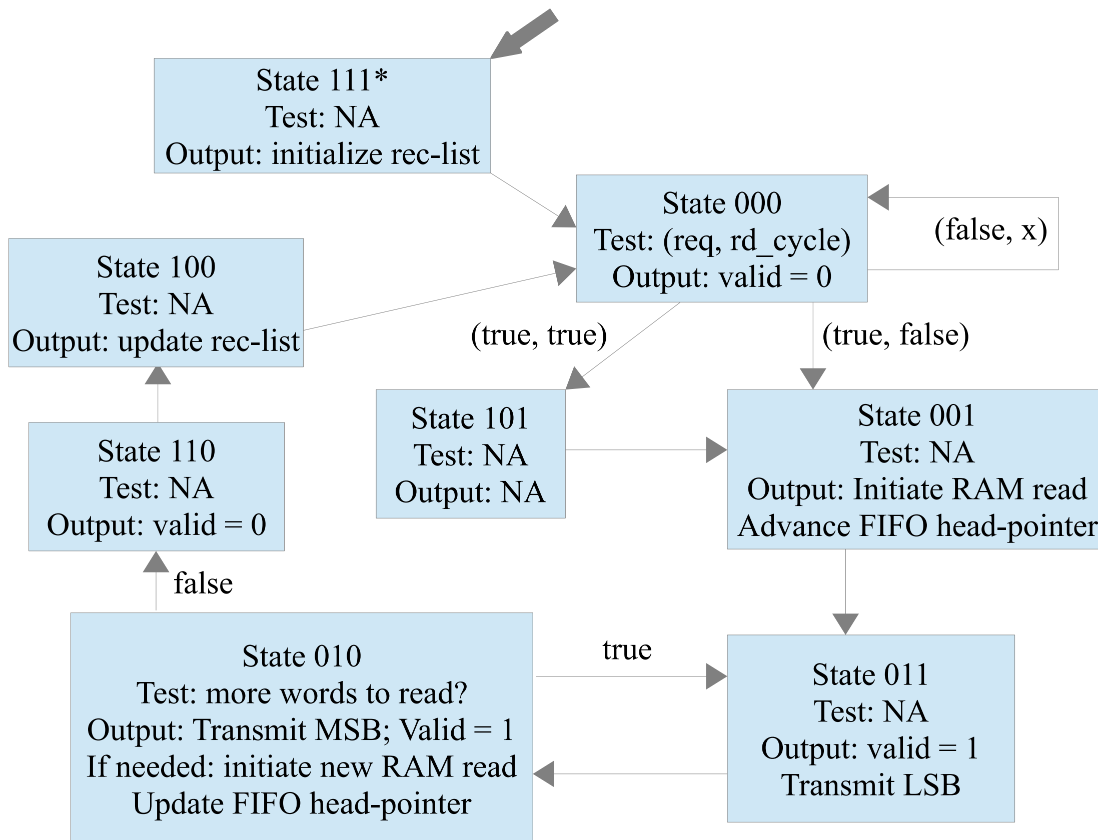
The "reject" signal, used by IENW node, is asserted when FIFO-controller write-FSM state is 0, i==0 (i.e. FIFO access turn of injection network), req[i] == 1 (i.e. active request to write FIFO from IENW), but not enough space available in the respective FIFO (i.e. FIFO0). This ensures it is asserted for just 1 clock cycle. Refer to Section 6.2.2 for the function of this signal.

Note: It is allowed for the upstream processing node to request for more space in the FIFO than required, and then write a smaller packet into it. However, if it requests for smaller space, and writes a larger packet, it may generate an overflow condition, and corrupt the data. **This will be an irrecoverable failure**, and must be avoided.

FIFO-Controller Read-Logic

The FIFO-controller Read-logic FSM is as follows on Figure 32. At reset the FSM starts in state 111, in which it stays for 10 clock cycle, during which it gets to write the RAM 5 times, which it uses to initialize the receive-list. Assuming that no packet is less than 5 words long and even if it is, there is nontrivial IENW handshake delay, the write-logic can finish writing no packet within this time. Hence, it is safe to initialize all the receive-list locations to 0. The initialization of the receive-list could be done using a readmem() function in case of FPGA implementation without needing to have a separate state (111) for it. But this is necessary for an ASIC implementation, where readmem() is not synthesizable. Thus the state has been introduced to ensure consistency at a very little overhead. After initializing the receive-list, the module gets into the idle state 000. In this state, the FSM monitors the req signal on the internal send-module to FIFO-controller bus. If it is received, as well as it is detected that the memory access MUX does not have the turn to give access to read-logic, then the next state is made 001. In case, request was received, and it was also the read-logic's turn to access the memory, then the FSM goes into the 001 state through 1 cycle delay in state 101, thereby aligning the state 001 to the memory access turn of the read-logic. In 001 state, the logic initiates a memory write request from the location maintained in the head-pointer of the FIFO requested. Then it puts the head-pointer forward by one word (modulo the fifo depth) and goes into state 011. In this state the first word of the read packet is already available. So, the valid signal of the internal bus is asserted and the lower half-word of the data is transmitted, while the upper half-word is buffered for transmission in the next cycle, which is always state 010. In the 010 state, the FSM checks if there were more words in the memory that needed to read to satisfy the current FIFO-read request. This is performed by initiating a counter in 001 state by reading the number of via-points requested by the send-module, and decrementing it each time a memory read is requested. In any case, the buffered upper half-word from the last cycle is sent out to the send-module in the 010 state. This state is also aligned to the read-logic's turn to access the memory. So, if more data were to be read from the memory, a new memory read-request is made and the FIFO head-pointer is advanced, and the next state is made 011. However, if there is no more data to be

read, the head pointer is advanced to the location of the next packet in the FIFO, if any, and the next state is 110. In state 110, the valid is made 0, signaling to the send-module that the transfer has finished. The next state is always 100, which is again aligned to the read-logic's turn to memory access. This turn is used to update the receive-list entry for the just received FIFO with the correct FIFO head-pointer. The corresponding rec_flag is made 1 if the FIFO is not empty, else it is made 0. Note that this updation is visible in the RAM output interface in the next clock cycle, i.e. when the state is back to 000. **Thus, the receive-list updation is visible 2 cycles after the valid is made 0** on the internal bus.



* The state '111' lasts for 10 clock cycles after reset deassertion. The 111→111 transition and the counter have been omitted from this diagram in order to avoid cluttering.

Figure 32: FIFO-controller Read-logic FSM

It is also worth noting that when sending out the packet, the lower half-word of the 2nd word in the packet is taken to be the the num_via entry. The logic reads this entry to determine the old number of via-points in the packet, and then updates it with the number of via-points that the send-module has requested to read from this packet. Thus the data sent out to the send-module is consistent in terms of the number of via-points present in the output packet. The old number of via-points is required at the transition from state 010 to 110, i.e. at the end of a read transaction, to determine the head-pointer to the next packet in the FIFO.

6.1.5 Send-list

The send-list has been implemented as a register bank inside the send-module, as shown below on Figure 33. The CPU can access the send-list through a memory-like interface. When the CPU finishes computation for a packet from an input FIFO number j, it updates the entry for the FIFO

no. j in send-list to communicate to the send-module that this packet is now available for sending out. The processor puts the number of via remaining in the packet after processing in the present node into the “new_num_via” field (and does not update the num_via field in the packet itself inside the FIFO, as has been mentioned also before in Section 6.1.1). It also calculates a packet-specific deadlock timeout value to be used by the send-module to detect any deadlock. The processor should also toggle the ready_toggle (so, it has to keep track of its previous value), which will thereby be different from sent_toggle. This difference is, in fact, the signal for the send-module that there is now a packet available in this FIFO to be sent out.

	sent_toggle	ready_toggle	timeout	route_dir	new_num_via
Input FIFO 0					
Input FIFO 1					
Input FIFO 2					
Input FIFO 3					
Input FIFO 4					
Address 5	Padding 0's		Proc node x coord		Proc node y coord

Figure 33: Send-list data-structure implemented as a register bank, accessable from CPU as a range of 6 memory addresses

At the reset all the entries are set to 0. By design, the processor can update an entry only if the sent_toggle == ready_toggle for this entry, and it has no write-access to the sent_toggle. On the other hand, the send-module can update the route_dir, timeout and sent_toggle of an entry when sent_toggle != ready_toggle. This way, mutual exclusion is maintained on these fields which are read-writable from both the send-module and the CPU. The last entry in the table is a read-only field that contains the coordinates of the present processing node inside the whole processing array.

6.1.6 Send-module

The send-module uses the entries in send-list and one more data structure, called arb_tab (i.e. arbitration table) of size 3x5, with 3 bits for each *output* IENW/CNW direction. When the data in one *input* FIFO has been provided access to an output interface, that information is entered into this table, e.g. if a packet from FIFO no. 3 gets access to the output direction 2, then arb_tab entry number 2 is updated to 3 (or binary “011”). When an output interface is free, an invalid direction is entered into the corresponding arb_tab entry, viz. “111”. The send-module FSM has been shown below on Figure 34.

At reset the FSM is in state 0000 with i = 0. It monitors whether the ready_toggle and the sent_toggle entry for FIFO no. i in the send-list are same. If they are, it indicates that there is no data available in this FIFO for sending out, and the state goes to 0010, in which i is incremented modulo 5, and the state comes back to 0000 with a new i. If however, the two toggles are found to be different, that indicates, as explained above in Section 6.1.5, availability of packet to be sent out in the corresponding FIFO, and the next state is made 0001. In this state, the routing direction is read from the send-list entry from FIFO i, and various signals on the external interface in that direction are checked. There are multiple possibilities. If the req in this direction is 0, that means that this interface is free for routing to, and hence the FSM goes into state 0011, in which a write-request is asserted in this direction, by asserting the req signal, along with sentting the “pack” signals to the number of via_points to be written, the value of which is read from the new_num_via field for entry no. i in the send-list, thereby satisfying the protocol requirements as outlined in

Section 6.1.2/Figure 28. The arb_tab entry for this direction is also updated to i. If, however, in 0001 state, it is observed that the req of the required direction is already 1, but the corresponding

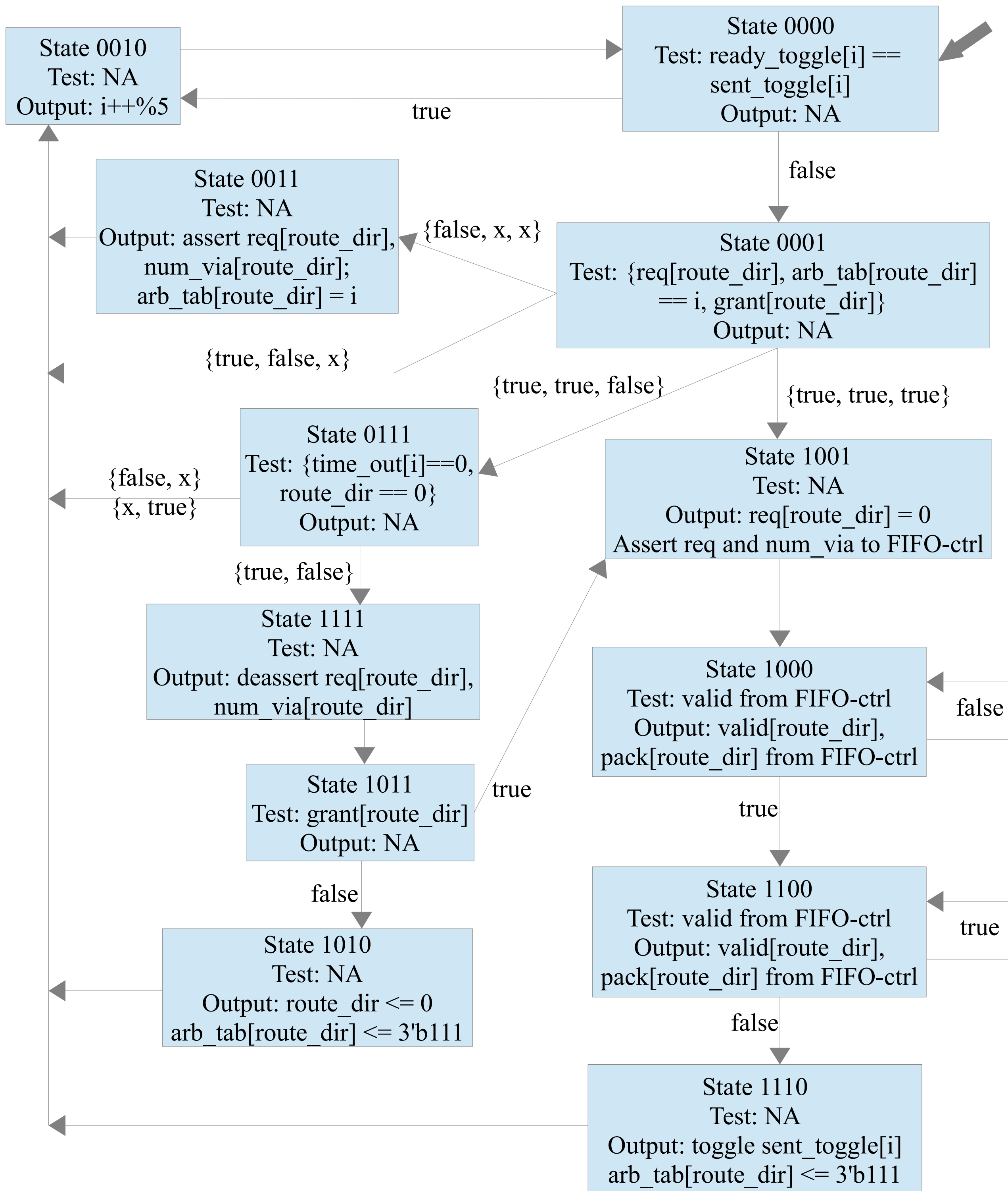


Figure 34: Send-module FSM

arb_tab entry is not i, that indicates that a different FIFO is already requesting access to this interface. In that case, no new request is made, and the next state is made 0010 to try luck with a different value of i. If however, the req is 1 as well as the arb_tab entry indicates that the access has been given to FIFO i, that means that a request has already been made for the present packet under consideration, and the FSM was waiting for a grant from the downstream node. In this case, the

grant is checked. If it is 0, the next state is made 0111, else it is made 1001. When the state is 0111, signifying that the module is still waiting for the grant for an already asserted send request, the next state is 0010 only if either the timeout counter for this request hasn't yet expired or if the routing direction being requested is 0 (i.e. IENW, which has no timeout). In this case, the module waits longer for the grant while trying to send out other packets in the meantime. However, if timeout counter does expire and the routing direction is not 0, the next state is made 1111. In this state, the active request is cancelled as a possible detection of deadlock, necessitating ejection of the packet. The next state is then 1011. The grant on this routing direction is checked for one last time, to detect the corner case of assertion of grant right after the deassertion of req, which is allowed as shown on Figure 28C/Section 6.1.2. If no grant is still received, the state is advanced to 1010, in which arb_tab is updated to make the currently occupied interface free for access by any other FIFO, and the route_dir field of the entry i in the send-list is made 0, implying the rerouting of the packet towards the IENW implying an ejection. If a grant is received in state 1011, however, just as in state 0001, the next state is made 1001 which initiates the packet sending sequence. In this state the external req towards the routing direction is deasserted, in compliance with the bus protocol, and a FIFO read-request is made to the FIFO-controller by asserting the req on the internal send-module to FIFO-controller bus, along with the proper number of via-points, as read from the send-list entry no. i and the fifo number, which is i. It then goes into the next state 1000, where it waits till it receives a valid from the FIFO-controller. During this waiting period the pack and valid signals from the FIFO-controller are routed without buffering to the output external interface. Thus the first data is available on the external interface in the same cycle as the send-module receives it from the FIFO-controller. As soon as this happens, the FSM transits to state 1100. The FSM stays in this state as long as the valid from the FIFO-controller is high, implying a flow of data. As in the previous state, the pack and valid signals from the FIFO-controller are transmitted out directly to the external interface. Once the valid goes down, the state becomes 1110 in the next clock cycle. In this state, the arb_tab entry for the presently occupied routing direction is freed up for use by some other FIFO, and also the sent_toggle field in the send-list entry no. i is toggled to signify the end of packet sending. This updation is visible in the send-list in the next clock-cycle. **Thus, the send-list updation is visible 2 cycles after the valid is made 0** on the internal FIFO-controller Send-module bus. The next state from the states 1010, 1110, 0011 is 0010, in which the i is changed giving an opportunity for a different packet to be sent out, possibly through a different interface.

As it has been shown here, and also in Section 6.1.4, both the send-list and the receive-list updations are visible 2 cycles after the "valid" on the internal bus is made 0 after the completion of a packet reading transaction. Thus, one of the two conditions for proper behavior of the system mentioned in Section 6.1.4, viz. that the receive-list must be updated no later than the updation of the send-list after a successful packet-read, is satisfied.

Timeout Counter

As can be observed, the send-module monitors the timeout counter when it is in state 0111. The registers implementing the "timeout" field in the send-module are reused in the hardware for the registers of these counters. Originally, the timeout decrementing for FIFO i was envisioned in the same 0111 state. It was, however, problematic as instead of keeping track of the length of time spent in waiting for a grant, it counted the number of turns FIFO i got for sending out its packet before a grant was received. This did not correlate well with the absolute time, because it depends on the time spent between two turns given to the same FIFO. This time may vary widely depending how much data might have been transfered from the other FIFO's between two such turns. This was not quite consistent with the idea of a timeout counter. Hence, the updation of this counter has been made more or less independent of the the send-module state machine. Each time the FSM goes into state 0011 and gives access of one external output interface to an input FIFO, the corresponding timeout counter starts down-counting. Once initiated, the decrementing operation happens in an

interval of a user-defined compile/synthesis-time parameter as a power of 2, allowing for a controllable granularity, and the counting stops when the timeout counter expires, i.e. becomes 0, indicating that the corresponding packet sending attempt may be taken to have failed due to a deadlock.

6.1.7 Data Consistency of Shared Resources

There are some shared resources in the design, that may be modified by different modules:

1. FIFO's (FIFO-controller read-logic, write-logic and processor)
2. Receive-list (FIFO-controller read-logic and write-logic)
3. Send-list (processor and send-module)

Because the shared resources may be modified by multiple modules, it is important to ensure mutual exclusion between them for the sake of data consistency. Here is how it is achieved:

FIFO's

The FIFO head-pointers can be modified only by the read-logic, and the FIFO tail-pointers can be modified only by the write-logic. Thus there is no shared resource violation, as long as it is not attempted to write to a FIFO, that does not have enough free-space, or to read from a FIFO, that does not contain enough data to be read. The first problem is solved by the design of the FIFO-control write-logic, which does not grant a request unless there is enough space to write, in conjunction with the send-module design which ensures that exactly as much data is written into a downstream node as was requested space for. The second problem is however trickier. The read-logic of the FIFO-control performs no consistency check before serving a read request from the send-module, leading to possible underflow conditions. However, this reading is ultimately controlled by the processing node software, which tells the send-module how much data should be read from which FIFO. Hence, the software has to ensure that this condition is never encountered. The software can make sure of it by ensuring that it always reads valid data as indicated by the receive and send-lists and that it never tries to increase the number of via-points in a packet, and that it updates the correct send-module entry.

The consistency between the processing software and the FIFO-controller is ensured by always modifying only valid packets in the FIFO, as specified by the receive and send-lists. A packet is shown to be valid in receive-list only some time after it has been fully written by FIFO-controller, and the write-logic will no more update it, and the send-list allows the processor to manipulate it only if it has not already been manipulated by the processor. Thus, the software ensures consistency of the data by following the mechanism described in Section 6.3.

Receive-list

It needs to be noted that the receive list may be updated by both the write FSM and the read FSM inside the FIFO-controller. The access between them is arbitrated by the memory access MUX (Figure 30). The coherency of the data in receive-list is ensured thus:

1. When only read or write is happening, or when read or write are happening from/to two different FIFO's: The receive-list updations are independent and the data is always correct.
2. When the same FIFO is being read and written in parallel and both operations finish at the same cycle, i.e. the state '110' of both state machines occurs in the same clock cycle: The write-FSM state '110' always occurs when rd_cycle is low. That means the read FSM updates the receive list in the next cycle, and the write FSM does the same the cycle after. The write FSM uses the value of the head pointer from the present cycle (i.e. state '110' of both FSM's) for this, and this value is already up-to-date. The read FSM also uses the same

value to update the receive-list, though it may indicate for 1 clock cycle (before being overwritten by the write FSM) that there was no available packet in the FIFO, if the only packet available was the one that had just been written in. This should not be an issue.

3. When the same FIFO is being read and written in parallel and the read operation finishes earlier, i.e. the read FSM state '110' occurs when the write FSM is still in either 010 or 011 state: In this case the read FSM updates the receive list with the correct head pointer, and correct packet availability. When the write operation finishes, the head pointer is still correct (can only be changed by the read FSM), and write FSM uses this value to update the receive list.
4. When the same FIFO is being read and written in parallel and the write operation finishes earlier, i.e. the write FSM state '110' occurs when the read FSM is still in either 010 or 011: In this case, the write FSM updates the receive list to indicate that there was some packet available, but uses a wrong head-pointer, which currently points to the word being read out from the FIFO. This is, however, not a problem, as the processor is not supposed to read this FIFO when a read-operation is in progress (implying ready_flag != sent_flag in the send-list), as is implied in Figure 42, and as has been explained in Section 6.3.

Send-list

In case of send-list, the mutual exclusion, and thus data consistency, is ensured at the hardware level, where an entry can be modified by the processor only if its ready_toggle == sent_toggle, else it can be modified by the send-module. The last address in the send-list is, however, read only, and can be modified by no entity in the design.

6.2 IENW Architecture

Different IENW architectures were proposed and evaluated in [11]. However, a simple “series” connection of all the processing nodes has been deemed sufficient for this network, as it is envisioned to have very low traffic, and thus requiring little sophisticated designing. This structure has been shown on Figure 35, and has been implemented in the hardware. The main design

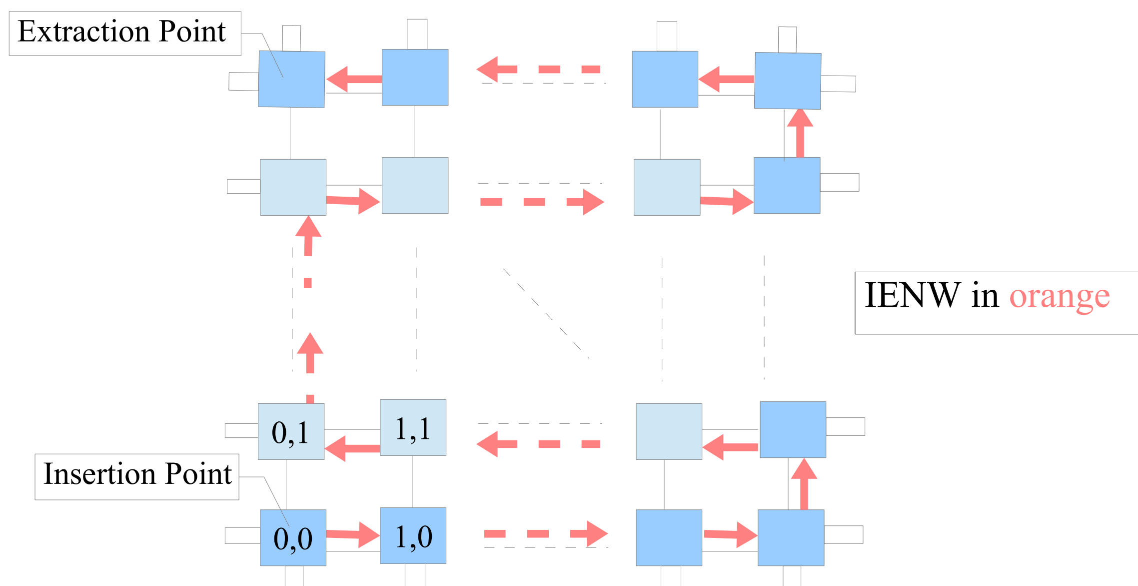


Figure 35: Insertion/Extraction network (Series topology)

constraint was that this network had to interface with the IENW interface of the FIFO-controller as well as the Send-module, possibly through a converter. In order to keep the converter as simple as possible some of the properties and protocols of these interfaces have been duplicated in the IENW, viz. the data-width has been defined as half-word-width (i.e. 16 bits), and all transfers have been taken to be performed in a burst mode with a burst of one complete packet. During the transfer of a word the lower half word is transferred first, followed by the corresponding higher half word.

In order to support the burst transfer, the following flow control strategies were considered:

- Virtual Circuit Switching
- Virtual cut through
- Store and forward

The popular and advanced strategy of wormhole switching cannot support burst transfer, as it requires support for blocking transfers. So, it was not considered.

Virtual circuit switching requires the request for a connection to be made by the source including the number of via points (translating into the packet-size) it wants to send, and the destination address. The destination will grant this request after some time, establishing a dedicated virtual connection between the source and destination, allowing to have burst transfers of complete packets. At the end of the packet transfer, the dedicated virtual connection is broken. This may have high latency due to the communication overhead required to set up the virtual circuit. This is not a serious problem, because the IENW is required to support only a low transfer rate. Still, in order to minimize it as much as possible, the packet forwarding logic inside every node has been designed as a pure combinational logic (2 MUX delays per processing node), with optional introduction of flip-flops (controlled by RTL parameter/generic) on the path, in order to break long timing paths when necessary (Ref: Figure 36).

Virtual cut-through and store-and-forward both require extra packet-sized storage. That is expected to be a high overhead, and thus discarded.

6.2.1 IENW Node Architecture

As explained above, the flow control solution chosen for IENW is a virtual circuit switching approach. The hardware logic for the IENW node at each processing node has been shown in Figure 36. It should be noted that the “destination address” signals are the coordinates of the destination processing node, which are specified as an (x,y) coordinate over the processing array. The coordinates of a node are same as the last entry in its send-list. The address to be specified for a packet to be ejected from the processing array is taken to be '11...11', i.e. all 1's. It has to be made sure that this coordinate does not occur for any processing node in the array, in other words, **the processing array size must not be 2^k x 2^k** where k = `COORD_SIZE`, a compile/synthesis-time parameter that specifies the number of bits required to encode each coordinate in a processing array.

Note that the grant and reject signals are buffered before being used as control signals in the FSM's in order to break potentially long timing paths. Whenever the write-FSM gets a request, it checks whether it was meant for the present node or another node, downstream from it. It sets up the fwd_ctr and fifo_ctr signals accordingly to route these requests. Similarly, the read-FSM monitors the two incoming buses, one after being forwarded by the write-FSM and the other from the Send-module. When there is a request from either of them, it forwards it to the downstream module – after arbitrating between the two if both are active. When a grant is received from the downstream, it is also routed to the correct node by the logic. The requests are, however, blocking. Thus if one request already has access of the output MUXes controlled by the signal_sel, it has the higher priority in the arbitration. This ensures that the requests are stable, allowing the destination nodes to respond, and to easily route the grant back properly. This blocking request strategy may, however,

lead to a deadlock scenario, as described in the next section.

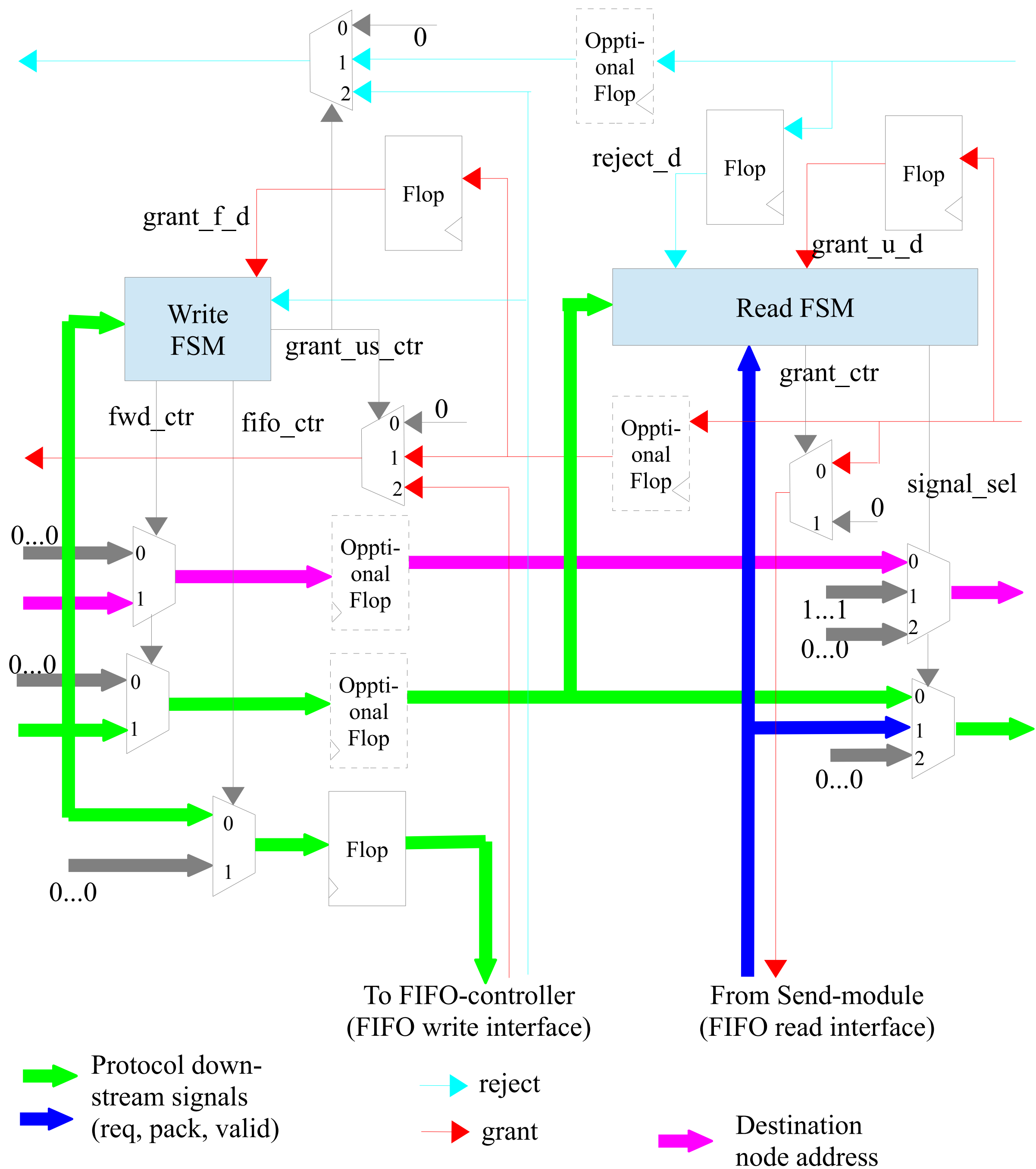


Figure 36: IENW node hardware

6.2.2 Deadlock Avoidance

It is possible to give rise to cyclic waiting involving both IENW and CNW if the IENW uses a blocking request for circuit set-up, as described in the previous section. A simple example of such a probable deadlock is shown below on Figure 37. One way of solving this problem is by making the circuit set-up protocol in the IENW non-blocking, i.e. there needs to be a mechanism to temporarily terminate a transaction request and release the bus for other possibly pending transactions to get access, in case the destination is not able to serve the transaction request presently. This has been implemented through a “reject” pulse, generated for 1 clock-cycle by the FIFO-controller write-

logic when it detects that there is not enough space in the FIFO to serve a FIFO write request received from the IENW. When this signal is received by the IENW, it is transmitted back to the source, which then releases the bus for at least 1 clock cycle by de-asserting the request before re-initiating the same transaction or a different one. During this time the IENW node inside the processing node where the “reject” was generated also de-asserts the request to its FIFO-controller, preparing itself to accept a new request only after its input req signal is de-asserted for at least 1 clock, signifying the termination of the previous rejected request. When the source is a processing node, it releases the bus for exactly 1 clock cycle. Then it checks if there is a pending forwarding request from its upstream node, and forwards it in the next cycle, thus giving it higher priority over the packet to be transmitted from its own FIFO, because this latter one got rejected recently. However, if there is no such pending forwarding request available in the present node, it re-initiates the same transaction after releasing the bus for 1 clock cycle. The IENW driver, to be used for loading packets into the processing array, must also conform to this protocol, which is described in further details below in Section 6.2.2.

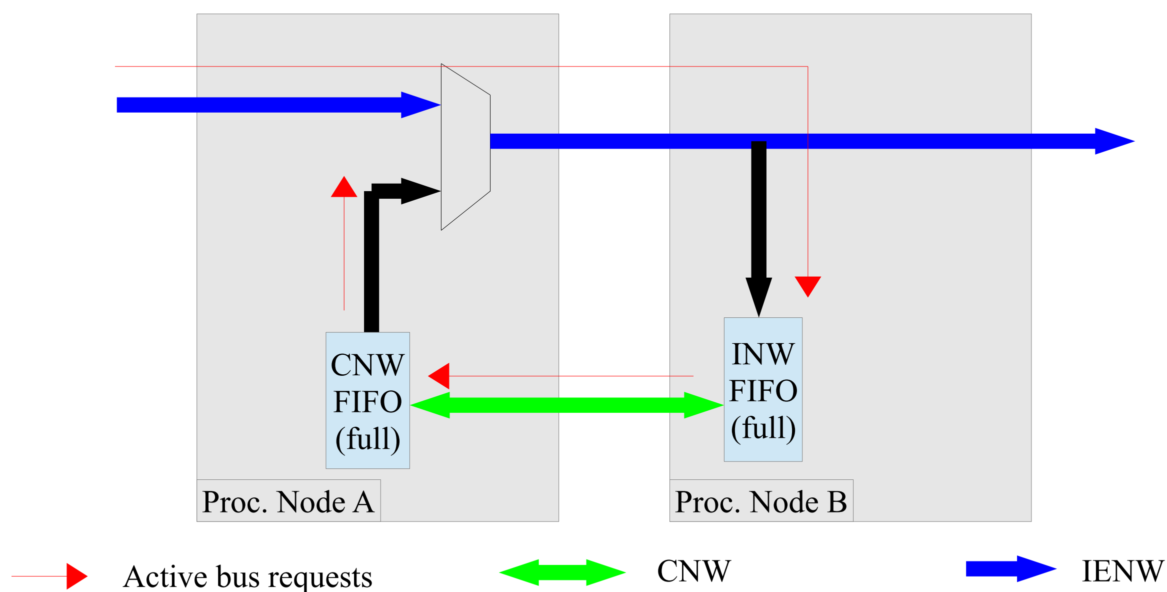


Figure 37: Illustration of how IENW could get deadlocked.

Evaluation of Deadlock Avoidance Strategy

This section aims at providing more details on the function of deadlock avoidance as implemented in the IENW nodes, because it has some important properties that need to be understood for proper evaluation of the hardware. The way the IENW has been implemented, each node may receive two requests to access the interface to the downstream node – one from its upstream node, and one from the local send-module inside the CNW node. If both are present, the node arbitrates between them.

On the other hand, each rejection is routed back through IENW to the corresponding request's source, which then releases the bus for at least 1 clock cycle, and this release is then propagated forward, informing each downstream node that there was a rejection to the previously asserted request. When an upstream node releases the bus due to rejection, the present node gives higher priority to any possible request coming from the local send-module in arbitration in the next cycle. Similarly, if a local request is rejected by the downstream, the present node releases the bus for 1 clock-cycle and gives higher priority in arbitration to any possible request received from an upstream node in the next clock cycle. This behavior can be observed on the waveforms on Figure 38. which has been drawn for a IENW node that instantiates the “optional” flops both on the forward and reverse paths (Figure 36/Section 6.2).

This behavior implies, that if a node has two incoming active bus requests – from upstream as well as local CNW node – and if the one, which it forwarded, gets rejected by the destination, then after each received rejection the node gives bus access to the request that lost bus access in the previous arbitration turn. Hence, the bus access alternates between the two requests if they both keep on getting rejected by their respective destination. Now let us assume that two nodes A (upstream) and B (downstream) are connected back to back, and they both have their local requests active (say, lreq_A and lreq_B respectively), as well as there is an active request received by A from its upstream node (say greq). In this case, B gives bus access alternately to lreq_B and to the request it receives from A, which may be either lreq_A or greq. A in its turn, alternates the bus access between lreq_A and greq. Thus, the bus access granted by B effectively takes the pattern (lreq_B, greq, lreq_B, lreq_A, lreq_B, greq, lreq_B, lreq_A, …), i.e. lreq_B gets bus access 2 times as frequently as either of the two other requests, though all of them do get bus access. With more and more nodes connected in series, the bus access frequency given to the requests with source further away from the destination falls exponentially as a power of 2, i.e. the bus access frequency is halved for every 1 node of additional distance from the destination. Therefore, the mechanism is guaranteed to avoid any deadlock, but the worst-case circuit set-up latency grows exponentially with the number of nodes in the design, if all nodes do request bus access at the same time and most of them cannot be served by their respective destinations. Thus, this is clearly not an efficient deadlock avoidance mechanism under heavy IENW load. But, as explained before, the IENW is expected to have only low loading. Also, all the processing nodes use IENW to send packets to the packet sink outside the hardware accelerator. This packet sink will likely be some big RAM associated to the external processor running the application as shown in Figure 11. So, it is unlikely that any of those transactions will get a reject. Combining these two observations, it can be concluded that, even though the worst case performance of the proposed deadlock avoidance mechanism is quite poor, for the present system it is expected to be completely adequate.

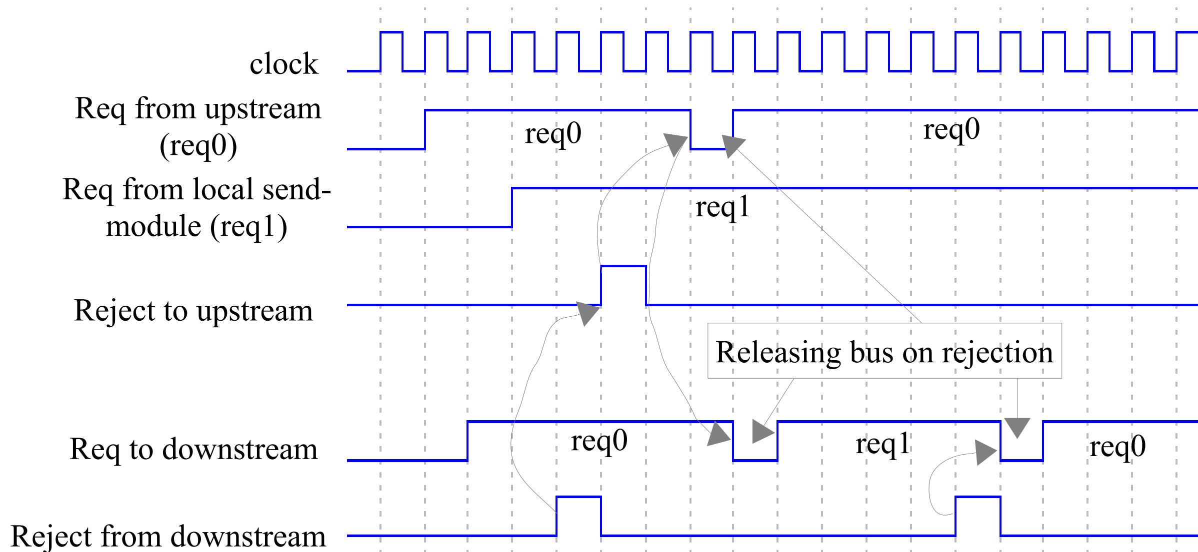
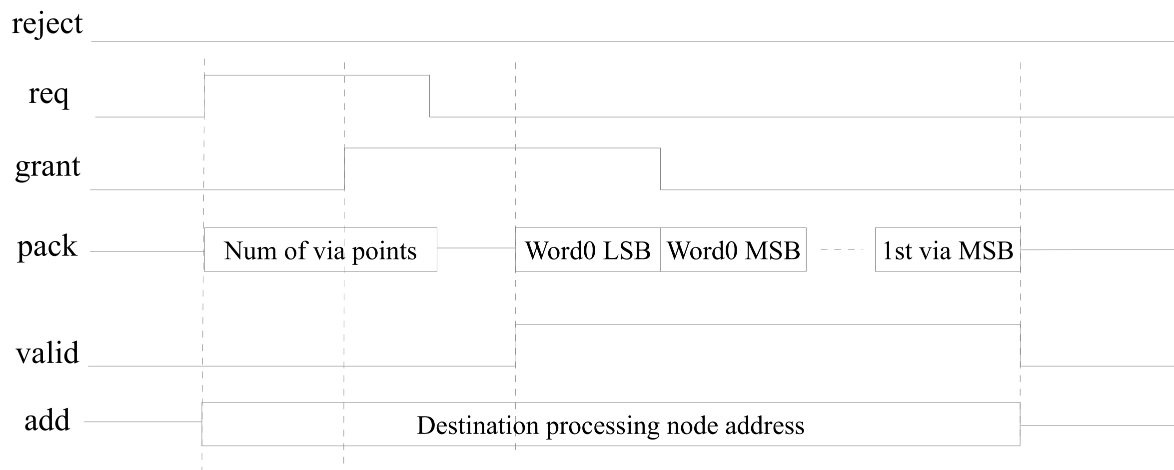


Figure 38: Deadlock-avoidance mechanism at an IENW node (with a flop on both forward and backward paths)

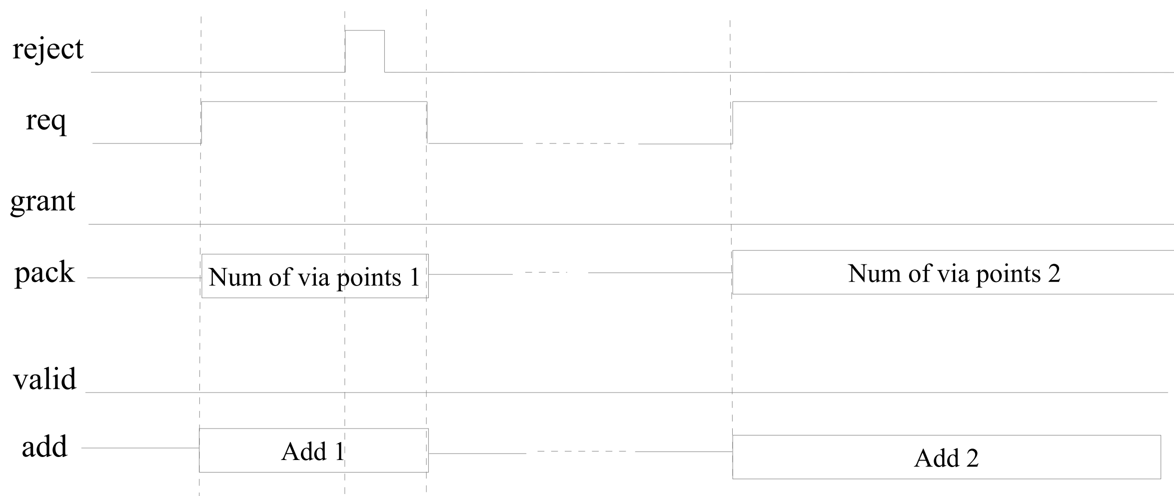
6.2.3 IENW Protocol

The IENW is composed of the following signals going from upstream to downstream node: req, pack, valid, add, and the signals going from downstream to upstream are reject and grant. When a source wants to send some data over the IENW it asserts “req”, and puts the number of via-points in

the packet it intends to write in the “pack” signals, and puts the coordinates of the destination node in the “add” signals. If the destination gives a “grant” after any number of clocks, a virtual circuit has been successfully set up. In that case, the source sends the whole packet in a burst mode after another unspecified number of clocks, putting the data into the “pack” signals, and qualifying it with the “valid”. The “req” may be de-asserted any time after getting the grant and before the finishing of data transfer. The “grant” may be de-asserted any time when the “valid” is high. On the other hand, if the request is rejected by the destination, it asserts the “reject” signal for just one clock-cycle. After this, the “req” has to go low for at least one clock-cycle before a new request can be made. However, the delay between the assertion of reject and de-assertion of req is unspecified.



Waveform A. Successful transfer



Waveform B. Rejected transfer, followed by a new request

Figure 39: IENW bus protocol

6.2.4 IENW Control Logic

As shown in Figure 36, each IENW node is controlled by two FSM's – the read-FSM and the write-FSM. The operation of the write-FSM has been shown on Figure 40. It starts at reset in the state 000. It checks if there is an active request from the upstream node, and if so, whether it is for the

present node, or for another downstream node. In the first case, the request is routed to the local FIFO-controller and the next state is made 011. If there was a request destined for a different downstream node then the request is forwarded downstream and the next state is made 100. And if there was no request the state remains unchanged at 000. When the 100 state, it stays in that state as long as the valid is 0, but either req is still 1 or the grant from upstream is 1, signifying that the previous request is still active. However, if all of them are 0, it signifies abortion of a request, and the state-machine comes back to the initial 000 state. If, however, a valid is received in state 100, it signifies the start of a successful data transfer, and the FSM transitions to state 101. It remains in this state as long as valid remains 1. When it becomes 0, it signifies the end of the transfer, and the FSM comes back to the 000 state. In both states 100 and 101, where data can be transmitted, the incoming signals are forwarded to the next stage.

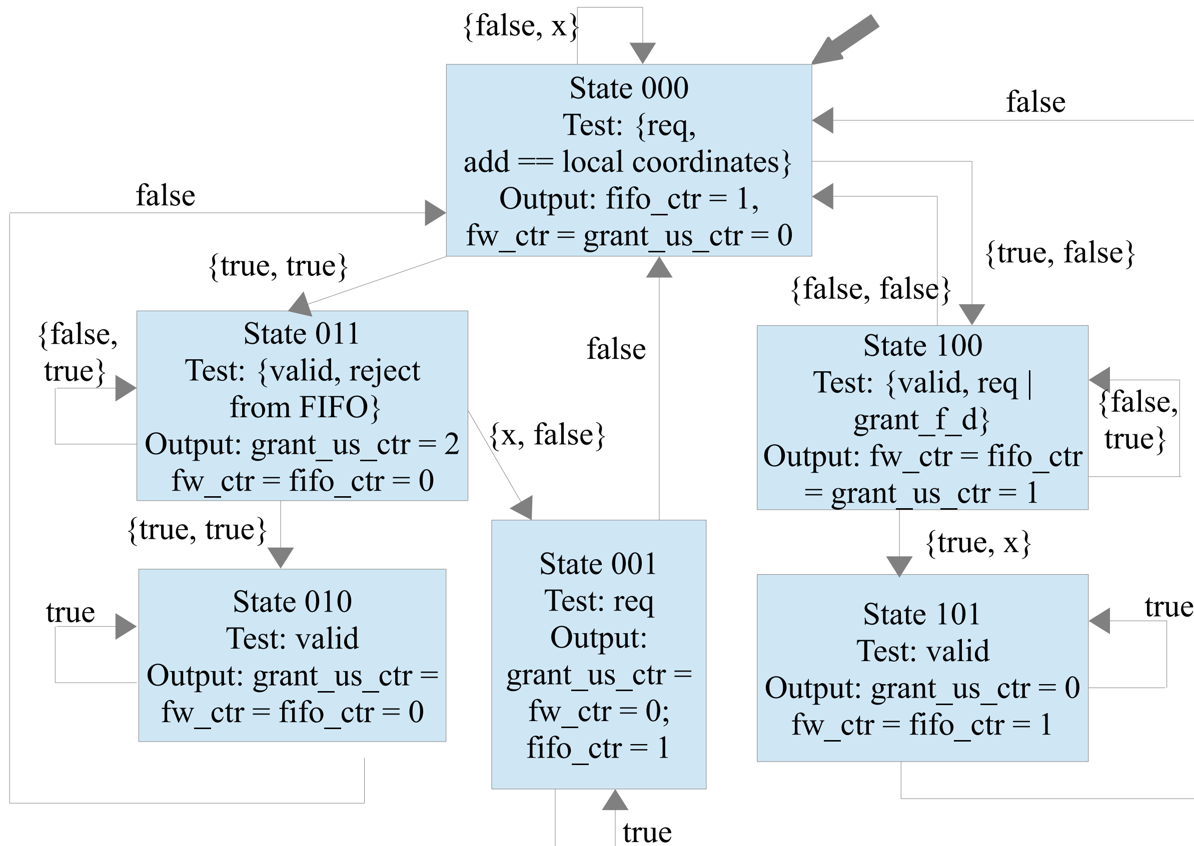


Figure 40: IENW node Write FSM (req, valid are from the upstream node)

When in state 011, i.e. expecting to receive data to be written into the local FIFO, it checks both valid as well as the reject from the local FIFO-controller. If it gets the reject, the FSM transitions into state 001, and disables the request to the FIFO-controller, and the reject is routed upstream. The FSM remains in this state till “req” is de-asserted for a cycle, which puts it back into 000 state, ready to process a new request. If the transaction is however not rejected, but in fact granted, then eventually a valid signal is received in state 011, and the next state achieved is 010. The FSM stays in this state as long as the transfer goes on, i.e. valid is high, after which it comes back to the 000 state. In the 011 abd 010 states, the incoming signals from the upstream are routed to the local FIFO-controller for writing into the FIFO.

The Read-FSM design has been shown on Figure 41 below. This starts at state 000 at reset, and monitors the req signals coming from the local Send-module as well as forwarded by the Write-FSM. If a request from the local Send-module is received it is routed to the next downstream node, and the FSM transitions into state 100. If there was no local request, but a forwarded request was

active, then that request is routed to the next downstream node and the FSM transitions to state 001. This steps performs an arbitration between the two possible incoming requests by statically assigning the local request highest priority. Similar to the forwarding mode of the write-FSM, the read-FSM also monitors the forwarded valid, when it is in the forwarding mode (001). If it is asserted, it signifies the beginning of a data transfer, and the next state is 010, in which it stays as long the data transfer is active, and then goes back to state 000. Otherwise, however in state 001, if the forwarded req or the grant from the downstream are high, it signifies that the request is still active, and the FSM stays in the same state. If however, all these signals are false, it signifies abortion of the previous request, and the state-machine goes back to 000 state, ready to tackle a new request. Note that the local request has higher priority in arbitration, thus if a forwarded request is rejected, and there is an active local request at that cycle, then the local cycle gets bus access in the next round of arbitration.

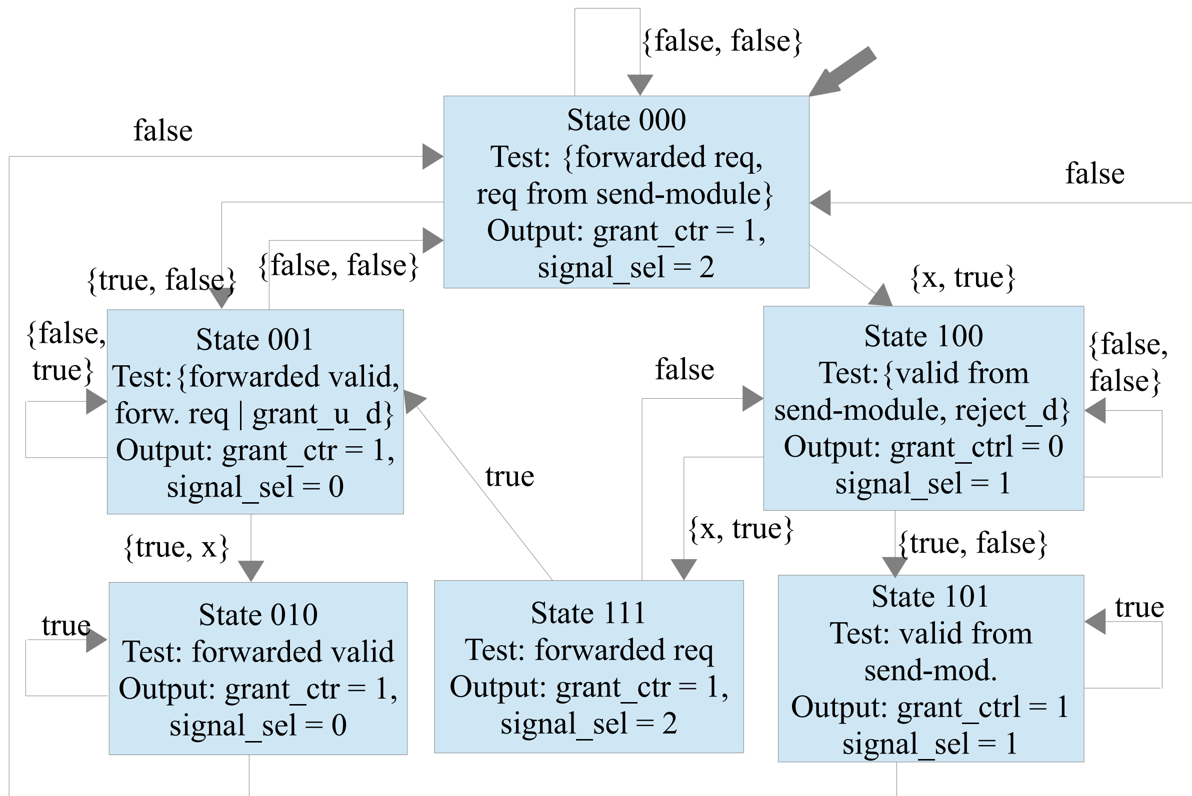


Figure 41: IENW node Read FSM

If the state is 100, i.e. the bus access has been granted to the local request, then it monitos the valid from the local send-module, as well as the “reject” from for the downstream. If neither is received, then the state remains same. If reject is received, it signifies rejection of the present local transaction request. So, the output request is disabled and the state goes to 111. If there is a forwarded request in this state, then that is granted access to the bus, and the state transitions to 001, the forwarding mode. Otherwise, the local request is again granted the bus access and the state goes back to 100. Instead of receiving a reject from downstream, if a valid is received from the local send-module n state 100, it signifies the beginning of a packet transfer and the state transitons to 101, in which it remains till the transfer ends, after which it goes back to the idle 000 state.

Note, when the FSM is in states 001 or 010, the output is connected to the forwarded interface from the upstream, and when it is in states 100 and 101, the output is connected to the local send-module interface. In state 111, the request is disabled. This implements the release of the bus for exactly 1 clock cycle by a processing node, if its local IENW request is rejected.

6.3 Processing Node Software

The processor in a processing node has to run a software to perform the necessary computations on the packets. It is envisioned to execute the following tasks:

1. NoC-to-processor arbitration, i.e. deciding the order of selecting packets from the input FIFO's for processing. It is assumed to implement a sort of round robin arbitration, for which an RR-list with maximum 5 entries is maintained, each input FIFO may feature at most once on the list. Whenever, no packet from a particular FIFO is present on the list, and a packet becomes available in it, that FIFO is added at the end of the list.
2. Routing, i.e. determining the output interface for a packet under processing, using the information from its via-list. (Ref. Section 4.1.4 in [11] for the algorithm)
3. Computation of the cost-contribution of the present map segment.
4. Updation of the packet to add the cost-contribution directly to the packet inside FIFO

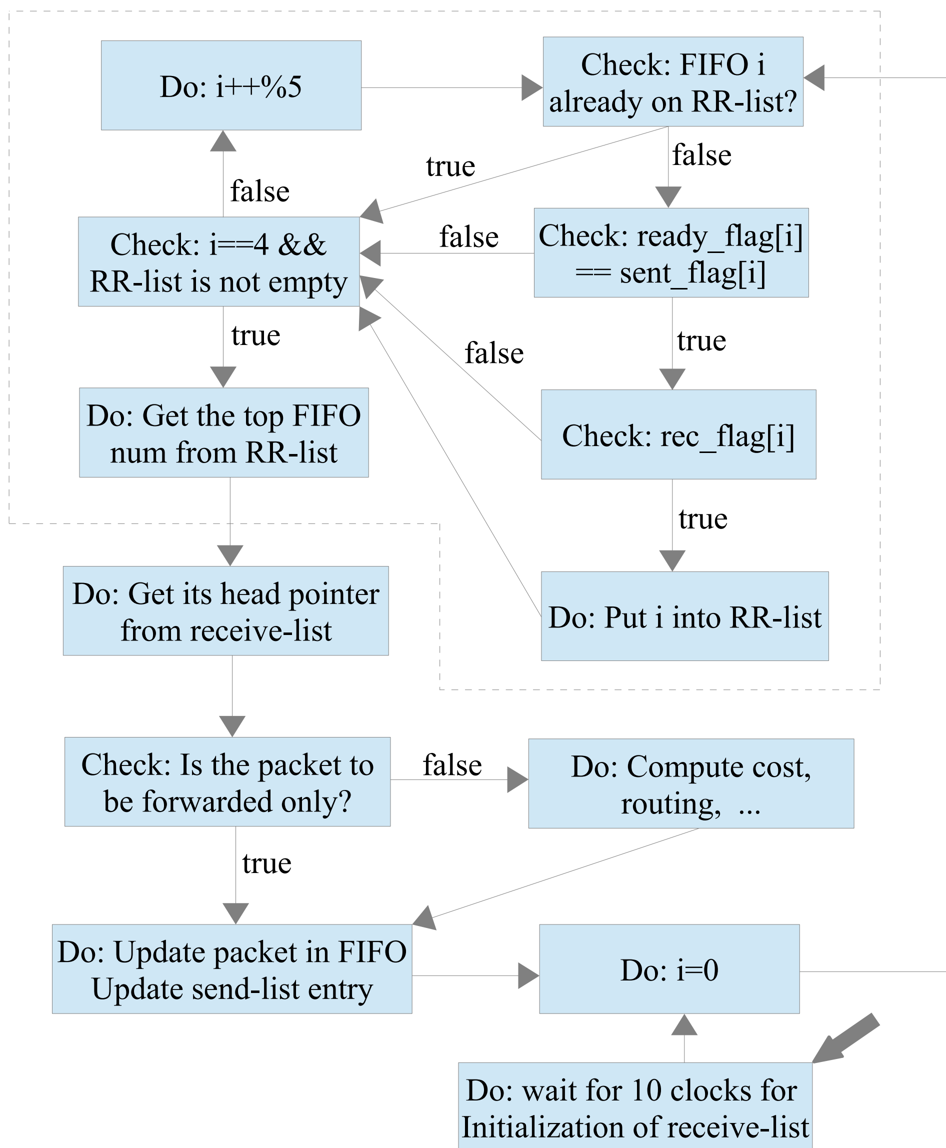


Figure 42: Conceptual state machine/flow-chart for software. The steps inside the dotted lines implement a round-robin arbitration scheme

memory, and number of via points by writing this information into the send-list, but not modifying that field in the FIFO-memory. The FIFO-controller modifies the packet accordingly, when sending out.

The diagram on Figure 42 shows a guideline conceptual state-machine to implement the software. It satisfies the requirement outlined in Section 6.1.4 that the software must first ensure that the ready_toggle == sent_toggle in the send-list for a particular FIFO, meaning that the corresponding FIFO is not being used by the send-module, before it attempts to read that FIFO. This implies a limitation that only the topmost packet in a FIFO can be processed by the processor at any time. The next packet can be processed only after the older topmost packet has been processed and sent out by the send-module.

6.3.1 Virtual Map Records

The conceptual state-machine on Figure 42 describes the normal behavior of the software while operating on the flight trajectory record (FTR) packets, that model the trajectories. However, as described in Section 2.3 and in [11], there are also the VMR packets required for programming the cost-functions to the processing nodes. In fact, for an FPGA implementation of the design without support for on-the-fly reconfiguration of the nodes, VMR's can be done away with, by initializing the respective data in the respective memory at the compile-time, using the readmem() function in the Verilog code. However, for the sake of generality, i.e. to support reconfiguration as well as ASIC implementation, it is important to support VMR packets. But it is clear that the hardware does not treat them any differently from FTR's. Hence the software needs to tackle that difference. The idea is that the hardware views VMR's also as FTR's. When the software reads a VMR from one of the input FIFO's, it should initialize the processor data memory by the cost-function values contained in it, and then instruct the Send-module to eject this packet through IENW, by specifying through the send-list that the new number of via-points in the packet is 0.

More ideas for the development of the software, not necessarily stemming from the CNW/IENW node hardware considerations, have been described in Chapter 10 “Further Works”.

6.4 Conclusion

In this chapter, the hardware design of the processing node, and hence that of the whole processing array, which is just multiple processing nodes instantiated and directly connected to each other without any glue logic, has been described in detail, except for the processor which could not be designed due to limitation of time. Thus the two main components involved in the design were the CNW node and IENW node, which also completely define the two NoC fabrics (CNW and IENW) that connect the processing nodes. One top level design for CNW was already proposed in [11] which was used as the starting point in the present work, which involved defining it in its minute details. The previously offered IENW solutions were, however, not used in the present work, as an even simpler solution was found out. The only piece of design remaining, viz. the processor, can be plugged into the processing node RTL module developed in the present work very easily, through a simple memory-like I/O interface. In Section 6.3 a guideline has also been offered for the development of the software to be run on that processor, which incorporates the hardware assumptions and constraints, thus making it easier for the software developer to start the development quickly without first having to understand the details of the hardware design. Thus, this activity has furnished a very valuable contribution, that can be easily built upon in order to finish the development of this accelerator.

Chapter 7

Hardware: Verification

Verification involves one of the major activities in any hardware development project. It is crucial for ensuring the functional quality of design. In the present work, the hardware RTL was implemented in Verilog, involving multiple processing nodes connected through CNW and IENW as has been explained before, except for the processor. A fully-fledged verification activity was, however, not undertaken due to the lack of time. In any case, the key design requirements were identified, and a set of test-cases were defined that aim to test the design for compliance with these requirements. The test-cases were implemented as completely deterministic directed Verilog testbenches, and the simulations were carried out using the ISim simulator (default simulator of Xilinx ISE). The resultant verification coverage is thus expected to be quite low, and are aimed primarily to detect obvious design flaws, rather than to weed out all possible bugs. The design requirements and the test-cases, developed here, may, however, be used as a reference for building a proper verification plan, and defining coverage goals, as further activities to the present work.

7.1 Design Requirements and Covering Test-Cases

The following table lists the key design requirements, that the hardware needs to be checked for compliance with. Each requirement is covered by one or more test-cases (TB1-7) as listed. The test-cases themselves are explained in the following Section 7.2. The “References” column contains further explanations to the requirements, often providing reference to other parts of this report for detailed explanations.

Requirement #	Requirement Description	Covering Test Cases	References
1	Check CNW grant is not asserted when not enough space in FIFO	TB4	
2a	Interleaving of read and write to same FIFO	TB2, TB4	
2b	Interleaving of read and write to different FIFO's	TB1	
3a	First valid data received from upstream node when rd_cycle == 0	TB2	Figure 30 for rd_cycle
3b	First valid data received from upstream node when rd_cycle == 1	TB2	
4a	Send-module requests data from FIFO-controller when rd_cycle == 0	TB2	
4b	Send-module requests data from FIFO-controller when rd_cycle == 1	TB2	
5a	Changed number of via-points correctly reflected in packets, read out from a FIFO	TB2, TB1, TB7, TB4	
5b	Old number of via-points used when updating the	TB2, TB1,	

	FIFO head pointer. Thus, a full packet is removed from FIFO, when it is read out irrespective of how many via points were requested to be read.	TB7, TB4	
5c	Changed number of via-points used by send-module to request transfer to downstream node	TB1, TB3, TB7, TB4	Change reflected in send-list: new_num_via
6a	Occurrence of timeout, as specified by processor	TB3, TB4	"timeout" in send-list
6b	In case of timeout, change of routing direction to 0 (Ejection Network)	TB3	
7a	Specification of routing direction by processor through send-list.	TB3, TB1, TB7, TB4	"route_dir" field
7b	Specification of new number of via's by processor	TB1, TB3, TB7, TB4	Included in 5c, 5a
7c	Specification of ready_toggle by processor in send-list	TB1, TB3, TB7, TB4	
8	Updation of sent_toggle by send-module in send-list	TB1, TB7, TB4	
9a	Receive list initialization after reset	TB1-4, 7	
9b	Receive list updation after a FIFO read	TB2, TB1, TB7, TB4	
9b.1	Receive list updation after a FIFO read in the same cycle as sent_toggle updation.	TB1	Sections 6.1.4, 6.1.6, 6.3 for explanation
9c	Receive list updation after a FIFO write	TB1-4, 7	
10	Two different FIFO's try to access the same output routing direction simultaneously	TB1	
11	FIFO roll-over – correctness of circular buffer implementation of the FIFO's	TB4	
12	CNW grant asserted just after timeout of a request to a downstream node	TB4	Section 6.1.2 Figure 28C
13a	IENW forwarding request, i.e. request received by a node is not meant for it.	TB5, TB6, TB7	
13b	Packet write from IENW into FIFO	TB5, TB6, TB7, TB4	
13c	Ejection of a packet through IENW	TB5, TB6, TB7	
14a	IENW is capable of parallel read and write, if it is not forwarding a packet.	TB5	

14b	IENW packet forwarding is mutually exclusive to the other two operations (i.e. FIFO read, and FIFO write). None of them can interrupt the other. All transfers are done in burst mode.	TB5	
15a	IENW rejections are routed to correct source	TB6, TB7	
15b	IENW nodes release the bus for 1 clock cycle when they get a reject for a request received from the send-module interface on that node.	TB6, TB7	Sections 6.2.2, 6.2.3
15c	When multiple requests contend for bus access, then all of them are assured to get their turn, even when other requests are not served, and hence do not become inactive.	TB6	Section 6.2.2
15d	When an IENW write request is rejected by FIFO-controller, the IENW node disables this request at once, avoiding getting a new reject/grant at a later point for the same request.	TB6, TB4	
16	Injection of packet into the network through IENW, then computation and transfer through CNW and in the end extraction through IENW	TB7	Overall integration check
17	FIFO-controller generates “reject” to unserviceable IENW request properly	TB4	Sections 6.2.2, 6.1.4

7.2 Details of Test-Cases

Each of the following seven test-cases (TB1-7) has been implemented as a separate Verilog test-bench by the names test_bench1-7. The following section describes them in details:

<u>TB1</u>

Design under test:

CNW-node (Figures 25, 26). All its interfaces, including IENW/CNW interfaces as well as the processor interface are controlled by the test-bench.

Description:

1. Assert req to write a 10-via packet into FIFO2, a 5 via-packet to FIFO3 and a 4 via packet to FIFO4
2. Write them in sequence of getting grant
3. As soon as the 10-via packet is finished writing put into send-list with 10 via points with routing direction 1
4. As soon as send module asserts req to output in direction 1, give it grant
5. This should ensure some overlap between reading and writing into different FIFO's
6. After the writing of 2nd and 3rd packets are over, update the send-list to send them out with number of vias reduced by 1, but both to direction 3.
7. When the send-module asserts the output req in direction 3, give grant. Should happen 2 times for the 2 packets.

8. At the end, the CPU reads all receive list and send list entries

Things verified:

1. FIFO2 reading and FIFO4 writing are concurrent
2. Data integrity of packets - number of vias read, updation of number of vias in packet, routing to correct direction
3. At one point two FIFO's (3 & 4) requested for the same output interface (#3). Correctly handled.
4. After a successful packet send, the receive-list is updated in the same cycle as the sent_toggle is toggled.
5. At the end, the receive list is empty but with correct head pointers
6. At the end, the send-list has ready_toggle == sent_toggle, with 0 for FIFO's 0 and 1, and 1 for FIFO's 2, 3, 4.

TB2

Design under test:

FIFO-controller inside CNW-node (Figures 26, 30-32) + DPRAM. The test-bench controls and emulates the incoming CNW/IENW interfaces, as well as the internal bus between Send-module and FIFO-controller.

Description:

1. Writes 2 packets into FIFO #4
2. Reads both packets out of FIFO #4, but with reduced number of via points
3. Reading out of first packet and writing in of second packet overlap
4. Send-module not used. Only FIFO-controller and memory used.

Things verified:

1. Data integrity of the packets

– including the replacement of the old number of via by new number of via

– required number of via are read out

2. Updation of receive-list

– Initialization by read FSM

– Updation by both read and write FSM's

TB3

Design under test:

CNW-node (Figures 25, 26). All its interfaces, including IENW/CNW interfaces as well as the processor interface are controlled by the test-bench.

Description:

1. A packet with 5 via points is written into FIFO 0
2. When the writing is visible from CPU through receive list, the CPU updates the corresponding send-list entry with a timeout of 20, and routing direction of 2

3. No grant is ever provided to the output interface

Things verified:

1. CPU sees the correct entry in receive list
2. Send-module sees the correct entries in ready-toggle, timeout, new number of vias and take action accordingly, i.e. assert the correct output req with correct new number of vias. The req direction changes to 0 after timeout.

TB4

Design under test:

Processing node except processor, i.e. CNW node + IENW node (Figure 25). All its interfaces, including the IENW interface, the CNW interfaces, as well as the processor interface are controlled by the test-bench.

Description:

1. Write a 10-via point packet into FIFO0 through the IENW (15 words, with a 5 word header)
2. When this write is visible in the receive-list, read out this 10-via point packet through the Left-interface (routing direction = 1), by properly updating the send-list and generating grant. The CPU also changes the number of via points to 3.
3. In parallel, write in a series of 11 6-via point packets (each packet: 11 words) into FIFO0 through IENW. This will fill up 121 words of the 127 words deep FIFO, as well as induce FIFO roll-over.
4. Assert request to write a 12th 6-via point packet through IENW. The FIFO won't have enough space to write this in, and should generate a single-cycle long “reject” pulse. This should also disable the FIFO write-request inside the IENW node.
5. At this point, the CPU allows reading out one packet from FIFO0 through Left-interface. It also changes the number of via-points for this packet to 5.
6. The testbench, however, asserts the necessary grant at the same instant as the req is deasserted by the node, due to timeout. This corresponds to the corner case transfer scenario as in Figure 28C.

Things verified:

1. Roll-over of FIFO
2. No more writing into FIFO when not enough space
3. Proper generation of the 1-cycle long “reject” by FIFO controller, when not enough space in FIFO.
4. Assertion of grant from downstream just after req is deasserted due to timeout, corresponding to Figure 28C scenario, and that it is treated properly by the hardware.

TB5

Design under test:

IENW node (Figures 25, 36). All interafes including the IENW interfaces and the interfaces to the CNW node, which follow CNW protocol, were controlled and emulated by the test-case.

Description:

1. First there is an incoming request on IENW to have a packet forwarded to the next node.

2. Even before a grant is received for this request, a request is received also from the send-module interface.
3. After a while a grant is received on the IENW (to be associated with the first request), which is to be forwarded to the upstream node.
4. After a while, the upstream node starts sends a packet, which should be forwarded to the downstream node.
5. After this packet send is finished, a new request should be raised by the present node towards the downstream node with respect to the already-asserted request received from the send-module interface.
6. A new grant is received with respect to the last request, and data is sent out from the send-module.
7. While this transfer is active, the upstream node makes a new request to write data into the present node's FIFO through the FIFO-controller. The present node should be able to forward this request to the FIFO-controller, even when making a transfer from the send-module interface.
8. The FIFO-controller gives grant, and a corresponding packet transfer is performed. Right after its completion, the upstream node makes a fresh request for another packet to be forwarded. However, the send-module interface was still making its transfer into the IENW. So, the new request should be blocked by the present node.
9. After a while the send-module transfer is completed. The present node should be able to forward the request from its upstream node to its downstream node in the next clock cycle.

Things verified:

1. Both state machines come back to the '000' state at the end of a packet transfer, which is signaled by at least 1 cycle of valid = 0.
2. The node can process all three kinds of requests properly:
    1. Forwarding a request when not destined for itself
    2. Assert FIFO write request through FIFO-controller when received a request for itself
    3. Make an ejection request when received request from Send-module
3. When a forwarding circuit is setup, the Send-module cannot access the bus, and vice-versa.
4. It is possible to read and write the FIFO's of the same node at the same time, i.e. the node can process an incoming IENW request with the same destination address as that of the present node, and a bus access request from the send-module.

TB6

Design under test:

3 IENW nodes connected in series. All their interfaces are controlled and emulated by test-bench.

Description

1. In the beginning a transfer request is made, as if from an upstream node, to be forwards through all the 3 nodes to downstream node.
2. When this request is active, requests are made for IENW transfer from the send-module interface in all the nodes. But these should not get served at this point, as all the nodes are currently forwarding the first request.

3. Once this situation is established, any request received at the most downstream output port is always made to be acknowledged with a reject after 4 clock cycles. This way, it can be verified that the IENW can send out different requests, each time it receives a reject.
4. The first time a reject is received at the upstream node, the corresponding request is de-asserted for exactly 1 clock cycle and the same transaction is request again in the next cycle.
5. After the second reject is received by the upstream node, it de-asserts the request for 1 clock cycle, and then makes a new request to write data into the FIFO of the middle node.
6. When the middle node receives the request to write into its FIFO, a reject is fed from the corresponding FIFO interface.
7. When this reject is received by the upstream node, it waits for a few clock cycles before de-asserting the req, and then after 1 clock cycle, it makes a new request which is again to be forwarded by all 3 nodes.

Things Verified

1. The transaction request received from the upstream node, as well as the send-module interfaces are properly sent to the downstream node, with proper destination address and number of via points.
2. When many requests are active, it is checked that getting a series of rejects gives turn to all the pending requests, even if not in equal priority or frequency.
3. When a FIFO-controller interface generates a reject, the corresponding IENW node disables that FIFO-request immediately, without waiting for it to be diabled by the source.
4. When the IENW bus request of a send-module is rejected, it releases the bus for exactly 1 clock cycle.

TB7

Design under test

The DUT in this test-case is made up of 2 processing nodes (except processors), say A and B, connected to each other through IENW (in the order: packet source → A → B → packet sink) and CNW where A is assumed to be located to the Left of B. The otherwise open interfaces (i.e. Up, Down and Left of A, and Up, Down and Right of B) are looped back onto themselves. This topology is same as that of an 2x1 processing array. The test-bench emulates the packet-source and packet-sink connected to the primary input and output IENW interfaces. The test-bench also controls and emulates the processor interfaces of the two processing nodes.

Description

1. A packet is written through IENW into node B.
2. The receive-list of B is monitored through the CPU interface of B. When the packet writing is visible on this interface, the corresponding Send-list entry is updated through the CPU interface to route the packet to the Left (i.e. to A) through CNW without change of number of vias.
3. The receive-list of A is monitored through the corresponding CPU interface. When that indicates the reception of a packet from B, the corresponding Send-list entry is updated to route this packet out into IENW, with reduced number of via points.
4. When the IENW packet sink receives a request for the first time, it rejects that. However, the same request is expected to be received after a gap of 1 cycle.

5. When the new request is received by the sink, a transaction is granted, and the packet transfer is allowed to complete normally.
6. The content of the packet is checked against the original packet. Only the number of via-points should change – both in the size of the packet and the corresponding header entry in the packet.

Things Verified

1. Processing nodes interface with each other properly, both through the IENW and CNW.
2. The Receive-list and Send-list operations are successful.
3. IENW forwards packets properly, e.g. when source writes to B, A forwards the packet. When A ejects packet to sink, B forwards the packet.
4. IENW handles "reject" properly.
5. Packet data integrity is maintained.

7.3 Conclusion

All the afore-mentioned test-cases have been implemented and used to check compliance of the hardware design – both at the level of its component blocks, as well as at the top-level system level (e.g. TB7). The tests have all passed. Even though, as explained at the beginning of this chapter, this approach is not guaranteed to detect all possible bugs in the design, it does provide a good level of confidence in the soundness of the design and also provides good confidence in the quality of the RTL implementation. While these tests were geared at evaluating the design with respect to its functional requirements, the following chapter is going to present its evaluation with respect to non-functional attributes, like the clock speed achieved and Silicon area required to implement it.

Chapter 8

Hardware: Synthesis and Evaluation

In order to evaluate the RTL implementation of the design described in Chapter 6 for various non-functional attributes, like achievable clock speed, Silicon area, etc. it was necessary to synthesize it. The synthesis is done with the ISE default setup using XST tool, and a Virtex7 FPGA was used as the target device. Virtex7 was chosen as a good benchmark for the capabilities of modern FPGA's. The first synthesis was run on a single node processing array, i.e. an array containing just one processing node, where all CNW interfaces were looped back to themselves and the IENW interfaces were available as primary IO's. The processor interface was also available as primary IO's, as there was no processor design instantiated. This was used to detect problems and evaluate the the design of a single node quickly, and to make any required modification. Then the array size was gradually increase to a realistic 8x8 to get more realistic data.

8.1 Performance Bottleneck and Resolution

The first synthesis runs on a single processing node array could achieve a top frequency of about 180 MHz. The critical path was found to look logically like below:

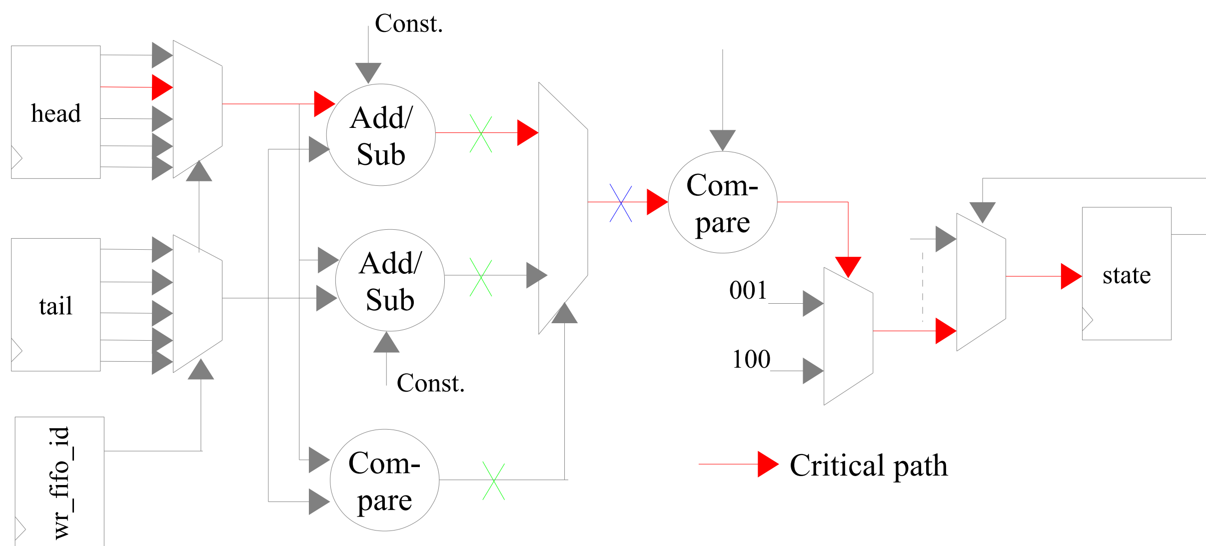


Figure 43: Logical view (not post-synthesis implementation) of one critical path in pre-retiming RTL, with two possible locations of introducing flip-flops marked by green and blue crosses (x).

As indicated on Figure 43, the solution adopted for shortening the critical path was by introducing some flip-flops. In general, such an introduction of flip-flops or pipeline stages would change the behavior of the hardware – often in unacceptable ways. In this particular case, it was found that such a modification of the behavior was, however, acceptable. The output of this logic-cone was found to be relevant only when state == 000. Then, this output determines whether the next state will be 001 or 100. As is clear from the diagram, after the introduction of the new pipeline-stage, the values considered for head, tail and wr_fifo_id would be from one clock-cycle ago, where head and tail are collections of 5 head and tail pointers each corresponding to one FIFO. Referring to Figure 31, the FSM of the write-logic in the FIFO-controller, to which this logic belongs, it can be seen that the state 1 cycle before 000 is always 100, and the value of wr_fifo_id (shown as “i” there) remains unchanged over these two states. In the same way, tail also remains unchanged in states 000

and 100. However, head is modified by the read-FSM (Figure 32), which works independent of the write-FSM, and thus it is possible that it would change within this 1 clock-cycle. It is, however, known that the head-pointer for any FIFO may change in 1 clock-cycle by at most 1 (modulo fifo_depth), showing that one more word has been read out from that FIFO. Thus, it is possible that because of introduction of this new pipeline stage, an older value of tail will be used, which will imply that the space available in the FIFO may be calculated to be one word smaller than what is correct for that clock period. However, it is significant only when a FIFO is being read from, and there is a request to write into the same FIFO, *and* at the same time, the size of the packet about to be written is exactly same as the space available in the FIFO. It is expected that such cases are quite rare, and even if it occurs, it only entails a somewhat delayed grant to the pending write-transaction, and not an outright failure. Thus, the functional impact is expected to be negligible. Hence, this solution was implemented, and the two variants shown by the green and blue crosses on Figure 43 were tried. It was observed that introducing the pipeline stage at the blue cross still makes a part of the former critical path the new critical path, whereas that at the green cross cuts the former critical path more evenly, giving a completely new critical path at a different section of the hardware. This also yields, as expected a higher frequency. In order to reduce the number of required flip-flops for the pipe-line stage, a slight modification was carried out by moving one of the constant additions after the new pipeline stage, producing the following design:

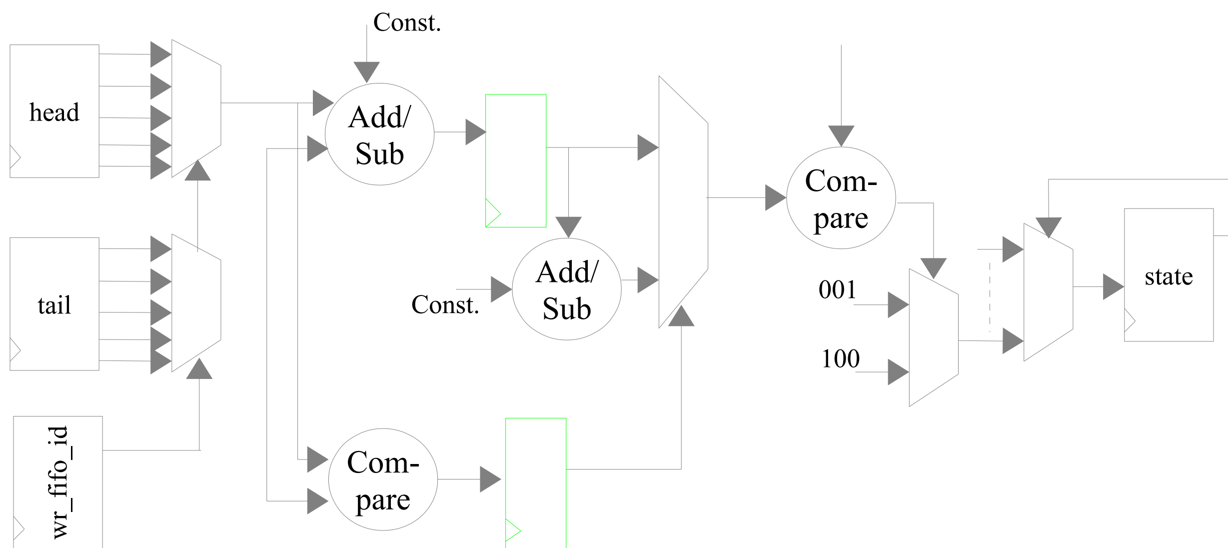


Figure 44: Logical view (not post-synthesis implementation) of the former critical path after retiming through introduction of the pipeline stages shown in green.

With this optimization, the hardware became synthesizable at 245 MHz.

Important hardware parameters/generics used for the synthesis were:

```
WORD_W = 32;     // Width of each word, same as width of the memory interface.
                 //   Must be even.
MEM_ADD_W = 10; // Width of address port of memory. Port on processor
                // is assumed to be 1 bit wider to allow addressing of send-list
HEADER_SIZE = 5; // size of FTR packet header in number of words
FIFO_DEPTH_W = 7; // Depth of FIFO in number of words = (2^FIFO_DEPTH_W -1)
```

The “pack” (equivalent of data) signals on communication interfaces (CNW and IENW) have been designed to have a width of (WORD_W/2), as explained in Chapter 6.

8.2 Hardware Resource Usage (Single Node)

The Virtex7 device targeted can provide the following hardware resources:

Number of slice registers: 607,200
Number of slice LUT's: 303,600
Block RAMs: 1030
The hardware resource usage of the design was found to be as follows. The percentages in the parentheses indicate the percentage of the total available resources in a Virtex7 device.

Processing Node excluding Processor:

The first synthesis was run on a processing array containing just one processing node (including IENW node and CNW node, but excluding the processor), where all CNW interfaces are looped back to themselves, but the IENW interfaces are available as primary IO's. The processor interface was also available as primary IO's. The resource usage observed was:
Number of slice registers: 499 (<0.1%)
Number of slice LUT's: 1462 (<0.5%)
Number of Flip-Flop LUT pairs: 1475
Block RAMs: 1 (< 0.1%)
Max. Clock Frequency: 245 MHz

Processor:

The processor to be instantiated in the processing node has not yet been designed. However, in order to estimate the additional resource that would be required for its instantiation, two popular free designs – Picoblaze and Microblaze – were synthesized [14][15]. One block RAM was used to implement their program memories. The results are as follows:

Picoblaze:
Number of slice registers: 82 (<0.02%)
Number of slice LUT's: 103 (<0.04%)
Number of Flip-Flop LUT pairs: 139
Block RAMs: 1 (< 0.1%)
Max. Clock Frequency: 299 MHz

Microblaze (with one IO-bus interace, light micro-controller configuration: Microblaze MCS):
Number of slice registers: 420 (<0.1%)
Number of slice LUT's: 665 (<0.3%)
Number of Flip-Flop LUT pairs: 888
Block RAMs: 1 (< 0.1%)
Max. Clock Frequency: 310 MHz

Assuming that the processor to be used for the processing node has a size close to that of Microblaze, one processing node would require about 919 registers, 2127 LUTs and 2 block RAM's. The LUT's (thus, combinational logic) part appears to dominate, and determine the maximum number of nodes that can be put on a Virtex7 FPGA to be around (303600/2127) ≈ 140. The actual highest number may, however, be lower because of routing congestion, that may result especially because of the lay-out of the block RAM's in columns inside the FPGA, while the processing array is laid-out in a grid which may often be closer to square in shape, thus leading to long wiring requirements between the logic and the block RAM's. This is one of the effects that was attempted to be studied by gradually synthesizing larger processing arrays, which instantiated the

afore-mentioned Microblaze as a stand-in processor.

8.3 Hardware Evaluation: Multi-node

Synthesis and timing runs were necessary to be executed with larger processing node arrays in order to evaluate the following:

1. To find out how frequently the "optional" flops need to be instantiated in the IENW (Ref: Figure 36)
2. Gather timing and FPGA usage data for more realistic processing node arrays, e.g. 8x8.

In order to perform this task, a top level module "synth_top_8x8" has been built that allows parameterized instantiation of the processing array, allowing to specify the array size through two synthesis-time parameters (thus supporting any array dimensions, notwithstanding the _8x8 in the module name):

```
parameter XSIZE = 8;  // Number of processing nodes in x direction.

parameter YSIZE = 8;  // Number of processing nodes in y direction.
```

It is also possible in the design to specify through two parameters of the processing node module, viz. `FLOP_ON_IENW_REQ` and `FLOP_ON_IENW_GRANT_REJ`, whether flops are inserted in the forward IENW path (i.e. req, add, pack, valid signals) or the reverse path (i.e. grant and reject) respectively for each instance of the processing node. In these synthesis runs, a Microblaze MCS was instantiated as the processor (including its program memory), in order to obtain realistic area data.

Optimal Optional Flop Insertion

Some of the synthesis data used to find out the optimum insertion of "optional" flops are as follows:
1. 2x2 processing array: All nodes have inserted "optional" flops:

  Max. frequency achieved = 247 MHz
  Critical path: Inside the FIFO-controller.

2. 2x2 processing array: Only the first and last nodes have inserted "optional" flops.

  Max. Frequency acheived = 160 MHz
  Critical path: IENW forward path starting from node (0,0) and traversing 2 nodes and ending in node (0, 1), giving a length of 3 stages. Comparison with (1) makes it clear that this path needs to be broken down in order to reach the highest possible frequency.

3. 2x2 processing array: Flops in forwards path on 2nd and 4th nodes. On reverse path (grant/reject) on first and last.

  Max. Frequency acheived = 215 MHz
  Critical path: Still the IENW forward path. This implies that flops must be introduced in more frequently on this path, implying in every processing node on the IENW forward path, i.e. FLOP_ON_IENW_REQ = 1 for all nodes.

4. 2x2 processing array: Flops in IENW forward path in all nodes. On reverse path, flops only in first and last nodes.

  Max. Frequency acheived = 247 MHz (max. Flop-to-flop delay = 4.07 ns)
  Critical path: Inside fifo-controller. This potentially indicates that it may be possible to allow larger intervals between the flop insertion on the reverse path without hampering operating frequency.

5. 2x4 processing array: Flops in IENW forward path in all nodes. On reverse path, flops only in first and last nodes, implying a length of 7 stages.

Max. Frequency acheived = 231 MHz (max. Flop-to-flop delay = 4.32 ns)

Critical path: IENW reverse path (grant signal). As the longest reverse path was 7 stages long with a delay of 4.32 ns, implying a rough delay of 0.617 ns per stage, it seems that the proper interval of instantiating flops of the IENW reverse path without degrading the operating frequency would be floor(4.07/0.617) = 6, where 4.07 ns is the maximum flop-to-flop delay inside the FIFO-controller.

6. 8x8 processing array: Flops in IENW forward path in all nodes. On IENW reverse path, flops at an interval of 6 nodes, including the first and the last node in order to provide isolation to the FPGA packaging pins.

Max. Frequency acheived = 245 MHz.

Critical path: Inside FIFO-control

Details of 8x8 processing array synthesis results:

Minimum period: 4.069ns (Maximum Frequency: 245.773MHz)
Minimum input arrival time before clock: 2.362ns
Maximum output required time after clock: 2.253ns
Maximum combinational path delay: No path found

Selected Device : 7vx485tffg1761-2

Slice Logic Utilization:
Number of Slice Registers: 61634 out of 607200 10%
Number of Slice LUTs: 139942 out of 303600 46%
Number used as Logic: 129062 out of 303600 42%
Number used as Memory: 10880 out of 130800 8%
Number used as RAM: 8192
Number used as SRL: 2688

Slice Logic Distribution:
Number of LUT Flip Flop pairs used: 156037
Number with an unused Flip Flop: 94403 out of 156037 60%
Number with an unused LUT: 16095 out of 156037 10%
Number of fully used LUT-FF pairs: 45539 out of 156037 29%
Number of unique control sets: 3137

IO Utilization:
Number of IOs: 74
Number of bonded IOBs: 74 out of 700 10%

Specific Feature Utilization:
Number of Block RAM/FIFO: 128 out of 1030 12%
Number using Block RAM only: 128
Number of BUFG/BUFGCTRLs: 2 out of 32 6%

8.4 Conclusion

The results obtained through various synthesis runs, as they have been presented in this chapter allow to draw conclusions about some key non-functional performance attributes of the design. It is understood that the design is well-suited for implementation on FPGAs, such as a Xilinx Virtex7 device. On such a device it can run up to a clock frequency of 245 MHz. The clock frequency supported by the instantiated processor may, however, be different, and it is possible to use different clock domains for the communication infrastructure (i.e. IENW node and CNW node) and the processors. On the other hand, it was also clear that a Virtex7 could accommodate a realistic array size of 8x8 processing nodes quite comfortably (at slice LUT usage of 46%). It also did not show any sign of routing congestion, that were anticipated due to the mismatch between the processing array lay-out and the block RAM lay-out inside the FPGA. Maybe such problems would become prominent with even larger processing array sizes. All the wires, except for those connecting to block RAMs, are, however, expected to be very short, because of the very regular 2D structure of the hardware. It might have mitigated the routing congestion more than anticipated. The other important finding from this exercise is that, for the Virtex7 implementation, the optional flops have to be instantiated in each node on the IENW forward path, but they can be instantiated at an interval of 6 nodes on the reverse path.

Overall, Chapters 6-8 have presented an efficient hardware design and RTL implementation of the communication infrastructure involved in the NoC-based hardware envisioned in Chapters 1 and 2. The processor necessary to complete this hardware could not be designed due to lack of time, but the present RTL allows a simple single-interface plug-in opportunity for the processor whenever it is implemented. Basic evaluation of the RTL has been performed in terms of functional requirements (by verification) as well as non-functional attributes (by synthesis and timing), and it is clear that the present design fulfills its envisioned design goals quite well.

Chapter 9

Discussion and Conclusion

9.1 Accelerator Model

As explained in Chapter 1, the present work provides a partial solution to the problem of designing and implementing an efficient and scalable hardware accelerator meant for parallelly computing costs of trajectories on a 2-D map. The design-work was carried out first at a relatively higher level of abstraction using the hardware-software co-design approach for the accelerator, as summarized in Chapter 2, and executed in the previous semester. A top-level architecture was worked out, and then implemented in a SystemC model, containing both the hardware and the software components. The model thus developed is cycle-approximate, i.e. it tries to model the cycle-accurate behavior of the system under some simplifying assumptions. This model was thoroughly simulated to evaluate various performance parameters for the design as documented in Chapter 7 of [11], which may be utilized in choosing the optimal design parameters, e.g. number of map folding layers, FIFO depths, deadlock time-outs, etc. However, for the final decision on these parameters the model should be informed with the correct values of some influencing factors, e.g. the time taken by a processing node to process a packet, etc. when those values are known, which may probably be found only after the processor and software implementations for the accelerator are available. Thus, the finalization of this model and the hardware-software system implementation are, after all, tightly coupled and may involve a few iterations.

9.2 Application

The accelerator can, in any case, be meaningfully utilized only by an external application. Thus, it is supposed to be a part of a larger computation system, which can use it to parallelize and speed up trajectory cost computations. In the present semester (spring 2014), one of the activities undertaken was therefore to study and define an application which may benefit from this accelerator hardware. As mentioned in Chapter 3, the most obvious field of application was in planning optimal trajectories, using heuristics that would need comparing many trajectories quickly according to some cost defined on them. Through a survey of the existing literature, it was found out that this was indeed a very important field for many military as well as civilian applications. Some of these examples have been summarized in Chapter 3. It was, however, observed that they use many different optimization heuristics, including genetic algorithm, but most of them do involve computing cost (or equivalently, fitness) of several trajectories. These sources, however, did not go into the details of the implementation of the algorithms. But as explained above, an ideal algorithm to benefit from the proposed hardware would be parallel in nature, thus exploiting the parallelization offered by the hardware. So, more literature was surveyed to find out ways to parallelize some of the most popular optimization algorithms, like simulated annealing and genetic algorithm (which was used by one of the applications described in Chapter 3).

Building on the findings of the surveyed literature, one simulated annealing-based and another genetic algorithm-based simple application was planned for development and evaluation, as described in Chapter 4. It was, however, found early that the genetic algorithm (GA) was expected to provide opportunities of parallelization, that are better suited for the proposed hardware, and GA was also used by one of the applications reviewed in the literature [4]. Hence, a simplified example application was designed based on a parallel genetic algorithm, drawing inspiration from this particular application. At the same time, a high level abstraction of the hardware accelerator was

also derived from the findings of the SystemC model simulations, mentioned before. Both this high-level model, and the application were implemented in C using POSIX threads, as explained in Chapter 4. This high level system model was subsequently simulated, as reported in Chapter 5, and it was found that it could take advantage of the hardware quite optimally. Just like the SystemC model, this high-level model can also be further refined by incorporating hardware data, like the time taken by a processing node to process a packet. This may then be utilize to fine-tune different design decisions for the application, for example the dispatch-rate (or through-put) adaptation mechanism (Ref. Section 4.3.5).

9.3 Implementation

After having thus established the soundness of the proposed design, the next step was to actually implement the system, which was taken up as another activity in the present semester (spring 2014), though all of its subtasks could not be completed due to lack of time. The three main components of the design are the hardware for the communication infrastructure, the hardware for the processors, and the software to be run on these processors. Only the first of these three activities could be performed, given the time constraints. A thorough micro-architecture was defined for the communication infrastructure (Chapter 6), and it was implemented in Verilog RTL. This also included defining the bus protocols and providing an interface to easily plug in the processor when designed, as well as to spell out software constraints and suggest one skeletal design of the software (Ref: Section 6.3).

This hardware design was subsequently unit-tested to weed out any obvious functional problems, by identifying the required functional behavior and defining directed tests to check for them. These tests were implemented as Verilog test-benches. The details of this activity have been provided in Chapter 7. Even though all the test-benches described have been implemented and used to qualify the hardware, the verification cannot be taken to be exhaustive, because it was simple deterministic directed testing. It is, however, expected that the requirements and test-cases developed for this activities and described in Chapter 7, can be used to help develop a fully fledged verification plan, and execute it.

Apart from functional evaluation, the hardware was also studied for its non-functional quality parameters, e.g. maximum clock-frequency and Silicon area, as described in Chapter 8. Xilinx Virtex7 devices were here targeted as the implementation platform. Because the proper processor hardware was still unavailable, a simple Microblaze micro-controller system was used to replace it [15], assuming that the final processor design will have similar area and speed characteristics. It was found that the communication infrastructure required Silicon area resources (1475 flip-flop LUT pairs per processing node) on the same order as the Microblaze processors (888 flip-flop LUT pairs per node). There probably exist opportunities to optimize the hardware, but it is clear that the over-all impact will be diminished by the size of the processors, and hence, it is taken as a relatively lower priority. On the other hand, it is observed that Virtex7 can comfortably accommodate a realistic processing array of size 8x8, providing another vital piece of evidence of the practicality of the design.

9.3.1 Model-Implementation Differences

As hinted in Section 9.1 above, the SystemC model and the implementation may involve some tightly coupled iterations to direct each other's development. The hardware implemented indeed contains some differences from the original SystemC model developed last semester, viz:

1. The write transactions into the different input FIFO's in a processing node can be parallel in the SystemC model. This is not so in the hardware, due to memory sharing between the

FIFO's.

2. The IENW in SystemC model is rooted – actually it has 2 roots – one for the injection network (rooted at the point of injection) and another for the ejection network (rooted at the point of ejection). The hardware implements a series connection of all nodes.

Ideally the SystemC model should incorporate these changes as feed-backs from the hardware design to be properly aligned; otherwise, it is more optimistic than the hardware. However, the impact of these differences was estimated to be relatively small. The SystemC code already models the FIFO read transactions in complete alignment with the hardware; the difference lies only in the FIFO write. Because most of the FIFO writes are driven by a send-module performing FIFO-read, the actual CNW bandwidth utilized is constrained in the model by the slower read transaction. Thus, the behavior is expected to be similar to that of the hardware. The model may still be somewhat optimistic, as the bandwidth loss in the hardware in handshaking between the nodes involved may be underestimated in the model. Even so, except for the handshake delays, transmission of a 20 via-point packet with 5-word header takes only 50 clock-cycles, because each via-point is represented in one word, and the CNW links are half-word wide (Ref: Section 6.1.4). On the other hand, it is unlikely that a simple processor executing the fairly complex software described in Section 6.3 and Chapter 10 will be able to finish the computation for one packet in that time. Similarly, though the model IENW provides high bandwidth, it is actually utilized for a small bandwidth – 1 packet is sent in/out in some hundreds of clock-cycles on an average. Thus, it is expected that the model behavior is still sufficiently aligned to the hardware, and the concentration was focused on other fields, like implementation, in order to speed up the development process.

9.4 Conclusions

As mentioned at the beginning (Section 1.1), the proposed hardware system was primarily aimed at efficiently parallelizing the relatively complicated problem of cost-estimation of trajectories on 2D maps. As would be quite clear from the simulation results presented in Section 5.4, applications may indeed utilize the proposed hardware to achieve efficient parallelization by optimally loading the hardware. Results presented in Section 5.3 prove that the quality penalty for the over-all result of the application run, incurred due to using this hardware for cost computation, is also quite limited. Therefore, the primary requirements of the system can be said to have been satisfied quite well by the proposed solution.

There are, however, some additional implementation-oriented requirements for the design which have also been presented in the afore-mentioned Section 1.1. The design is required to be scalable in order to be able to cope with changing performance requirements. The NoC-based homogeneous multiprocessor architecture proposed in Section 1.3 provides an excellent solution to this problem, providing unrestricted scalability in principle. In practical terms, only the available Silicon area is expected to impose any restriction on this.

In addition to this, the preferred implementation medium was expected to be FPGA's, providing the possibility for rapid prototyping and hardware scaling to accommodate changing performance requirements (as supported by the scalable design). While the design has been executed in a way to support ASIC implementation directly without any design change, whenever possible it does take advantage of available features of FPGA's, e.g. optimizing the design using a dual port RAM (Ref: Section 6.1), implemented as an FPGA block RAM. However, because of the specific lay-out of the block RAM's inside an FPGA, there was some concern of routing congestions, as the design may not always be laid out in a matching fashion at the time of placement-and-routing. It was, however, observed that this problem is not significant even for quite practical processing array dimensions of 8x8, when targeted to a Virtex7 device (Ref: Section 8.3). This way it is proven that the design is

indeed well-suited for FPGA implementations.

It can be finally concluded that, even though only a part of the design implementation has been completed so far (viz. NoC communication infrastructure hardware), the design has been modeled and studied thoroughly, and its feasibility and utility have been demonstrated at all different levels of abstraction, going from application level down to FPGA prototyping level. It has thus been shown to meet all the design requirements that were laid out at the beginning. As the first part of the system development activity, it provides the required confidence in the design to proceed further, as well as defines concrete guide-lines to direct the future works to complete the development.

Chapter 10

Further Work

As described in Sections 1.4 and 1.5, some tasks still remain to be performed in order to complete the initially stated design goals, i.e. to build a fully functional hardware accelerator for performing trajectory cost computations efficiently and parallelly. The main open tasks, as is apparent from this previous discussion, are:

1. Processor hardware:

   The hardware for the core computing element at every processing node needs to be designed and implemented. It can either be a standard off-the-shelf processor (possibly with some modifications/extensions) or a processor designed from scratch. If the second solution is chosen, it will also be necessary to design a corresponding compiler in order to able to program it efficiently and accurately. Irrespective of which solution is chosen, it is clear, as discussed in [11], that the processor must be able to handle square-root, division (or equivalently, inverse) and multiplication operations efficiently, as they are necessary for packet routing and trajectory cost computations. It will also be preferred to use a 32-bit processor (e.g. Microblaze [15]), because that is the system word-width assumed, and because that matches the word-width of the block RAM's in a Xilinx FPGA, which is a likely candidate as the prototyping platform of the design. The data present in the packets themselves are expected to be fixed-point, but the software running may need to use some floating-point internal variables. This point needs to be studied in more detail, and will certainly affect the requirements for the processor.

2. Processing node software:

   The processor in every processing node will have to execute a software program in order to perform certain tasks. Its duties will likely consist of choosing the order of packet computation (NoC to processor arbitration), packet routing computations (for the NoC), computing and accumulating the local cost-contribution to the packet (using the length and direction of the trajectory in the current map segment as calculated during routing computation), and initiating packet transmission when finished. A rough sketch of these activities have been described in Section 6.3, and the routing algorithm has been explained in Section 4.1.4 of [11]. These operations involve square-root, multiplication and division/inversion. It will be nice to have hardware support for these functions. Otherwise, they will also have to have their own software solutions. As explained above, the software may need to use floating-point internal variables to achieve the required accuracy, but that will also be expensive in terms of performance and probably hardware cost, assuming the hardware is made to support floating-point arithmetic. Therefore, it is important to consider these factors when designing the software and optimize it accordingly.

   There are also some other features that are to be supported by the software. It should be able to process VMR packets (Ref: Section 2.3, 6.3.1) on the fly, and thus be able to reconfigure the processing nodes whenever necessary. This will make it possible to use the accelerator for slowly changing cost-functions, e.g. when the cost is determined by the weather conditions in a geographical region. The software also has to be able to handle both scalar and vector cost-functions, depending on the application, as explained in Chapter 1. It should also support (linear) combinations of such cost-functions. One advanced feature could be to support different map resolutions at different sections of the map, e.g. at flatter sections, a coarser map division should suffice, but at rougher/jagged sections (i.e. where the cost-

function is highly variable), a finer division will be required. Such a feature cannot be implemented solely by the accelerator without significant design change, and has to be implemented primarily by the external application, but it should be studied if the processing node software needs to incorporate some features to make it possible for the application. As will be explained below, it seems it does need to incorporate some support. Note, however, that the simple application developed during the present work, and described in Chapter 4, does not implement this advanced feature.

3. Prototyping:

   When the complete hardware RTL and software binary code for the targeted processor are available, they should be used to prototype the accelerator on an FPGA. As explained in Chapter 8, Xilinx Virtex7 seems to be a good choice for such a platform, as it can easily accommodate a realistic processing array of size 8x8 using approximately half of its hardware resources. The hardware of the processor was not available, as explained before; so, a Microblaze micro-controller system was instantiated instead for the hardware resource utilization estimates. Once the prototype is available, various test-cases can be run on it to test the system, as well as an application, like the one described in Chapter 4, running on an external CPU (e.g. personal computer) should also be run, using the prototype as its hardware accelerator to measure the effectiveness of the whole design.

Beyond the tasks described just above and apparent from Figure 3, there is another important task that needs to be performed:

4. Communication Infrastructure Verification:

   As described in Chapter 7, the RTL of the communicate infrastructure, consisting of the CNW and IENW nodes and their mutual connections building the CNW and IENW, have only been unit-tested, and not thoroughly verified. However, the information and guide-lines required to develop a comprehensive constrained random test-suit have been produced in Chapter 7, comprising the system requirements to be tested, and ideas for the test-cases that can test them. Obviously, this activity may prompt bug fixes in the hardware. A similar activity will also need to be performed for the processor if it is designed locally.

Beyond the design of the hardware accelerator itself, lies the external application, that uses it to speed up trajectory cost computations. As explained before it is important to have a realistic application in order to be able to evaluate and demonstrate the effectiveness of the hardware developed. Such a basic application has been developed and described in Chapter 4 of this document, but as hinted above, there exist scopes for implementing more advanced features, e.g. reconfiguring the accelerator on the fly, and support for different resolutions in different parts of the map. This second feature is expected to be quite important for both performance and accuracy of the results, but it is also expected to be relative complicated to implement. The idea is to transform a standard single-resolution map into a multi-resolution map, by extending the more finely divided representations of some map segments to the original map (the blue and orange sections on Figure 45), while making sure that the nodes associated with the original segments do not process those segments any more (shown by the dark green squares on Figure 45B). Then the logical map folding concept may be used to fit the extended map into the intended size of processing array as usual. Now the softwares running on the various processing nodes are made aware of the boundaries between various sections of the map (represented on Figure 45B by different colors), and whenever the routing calculations of a packet tries to take it across one of these boundaries, the software ejects it from the processing array (using a mechanism similar to CNW deadlock resolution) including the information of the coordinates of the attempted crossing and the partial cost computed for the trajectory until that point. The application can then use this information to re-launch this packet, using the previously computed crossing point as the new start-point, and launch it in the section of

the extended map with the proper resolution. Thus, it should be possible to implement this functionality using the presently envisioned hardware, but using software and application support.

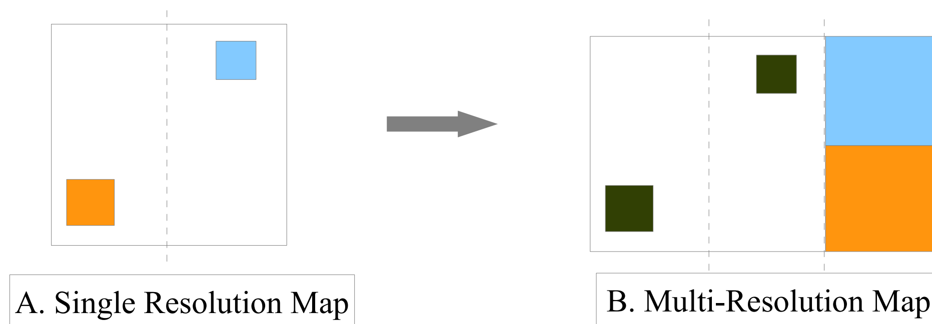


Figure 45: Transforming a single-resolution map to multi-resolution map, with the dotted lines showing some of the foldings. The blue and orange regions on A have relatively fast changing cost-functions, i.e. they are "jagged". On B, finer resolution views of these areas have been appended to the map on A.

As would be obvious from this discussion, some additional algorithms will need be devised to

- determine the optimal resolutions to use for different sections of the map
- determine the optimal map extension to be used (as shown on Figure 45). Note that, in general, the higher resolution sections could be appended to the original map in any direction and order. So, it is important to find out which of these will be the best to use.

As described in Section 9.3.1, there are some mismatches between the SystemC model of the accelerator and the hardware designed. The differences were not felt to be significant enough to prompt an immediate modification of the model, but it will be nice in the long run to align the model to the implementation.

Thus, the presently known major open points have been summarized above, and will need to be closed in order to be be able to fully develop and use the system envisioned. It is expected that by executing these tasks, an excellent scalable hardware accelerator for parallelizing trajectory cost computation efficiently can be realized. The system has been designed and evaluated mainly with an FPGA prototype in mind, but whenever necessary, steps have been taken to make it suitable for direct implementation on ASIC as well (e.g. receive-list initialization through hardware, rather than readmem() as explained in Section 6.1.4) without any modifications, even if it may not be the optimal design (e.g. using a dual port RAM as shown on Figure 26) for ASIC.

Reference

[1] LaValle, Steven M. "Planning Algorithms". *Cambridge University Press,* 2006.

[2] Lindemann, Stephen R; LaValle, Steven M. "A Multiresolution Approach for Motion Planning Under Differential Constraints ". *Proceedings IEEE International Conference on Robotics and Automation*, 2006.

[3] Kadamba, Pierre T; Meerkov, Semyon M; Zeitz, Frederick H. III. "Optimal Path Planning for Unmanned Combat Aerial Vehicles to Defeat Radar Tracking ". *Journal of Guidance, Control, and Dynamics*, Vol. 29, No. 2, March–April 2006.

[4] Obermeyer, Karl J. "Path Planning for a UAV Performing Reconnaissance of Static Ground Targets in Terrain ". *AIAA Guidance, Navigation, and Control Conference*, Chicago, August 2009.

[5] Kamyar, Reza; Taheri, Ahsan. "Aircraft Optimal Terrain/Threat-Based Trajectory Planning and Control ". *Journal of Guidance, Control, and Dynamics*, Vol. 37, No. 2, March–April 2014.

[6] Dogan, Atilla; Zengin, Ugur. "Unmanned Aerial Vehicle Dynamic-Target Pursuit by Using Probabilistic Threat Exposure Map ". *Journal of Guidance, Control, and Dynamics,* Vol. 29, No. 4, July–August 2006

[7] Cantú-Paz, Erick. "A Survey of Parallel Genetic Algorithms ". *Réseaux et systèmes répartis calculateurs parallèles,* Vol. 10, No. 2, 1998.

[8] Onbaşoğlu, Esin; Özdamar, Linet. "Parallel Simulated Annealing Algorithms in Global Optimization ". *Journal of Global Optimization* 19: 27–50, 2001.

[9] Ram, D. Janaki; Sreenivas, T. H; Subramaniam, K. Ganapathy. "Parallel Simulated Annealing Algorithms " . *Journal of Parallel and Distributed Computing*, Article no. 37, 207-212, 1996.

[10] Rudolph, Günter. "Massively Parallel Simulated Annealing and its Relation to Evolutionary Algorithms ". *Evolutionary Computation*, Volume 1 Issue 4, Pages 361-383, Winter 1993.

[11] Jana, Dibyajyoti. "Scalable FPGA Fabric for Parallelizing 2D -Trajectory Cost Calculations : System Modeling and Micro-architecture ". Autumn Semester Project, NTNU. Dec 2013.

[12] Jantsch, Axel; Tenhunen, Hannu. "Networks on Chip". *Kluwer Academic Publishers/Springer Publishers.*

[13] Nurmi, Jari (Editor); Tenhunen, Hannu (Editor); Isoaho, Jouni (Editor), Jantsch, Axel (Editor). "Interconnect-Centric Design for Advanced SOC and NOC (Mathematics and Its Applications)". *Kluwer Academic Publishers/Springer Publishers.*
Chapter 10: ARBITRATION AND ROUTING SCHEMES FOR
ON-CHIP PACKET NETWORKS
by Heikki Kariniemi and Jari Nurmi

[14] Xilinx Inc. “Picoblaze 8-bit Microcontroller”
<http://www.xilinx.com/products/intellectual-property/picoblaze.htm>

[15] Xilinx Inc. “Microblaze Soft Processor”
<http://www.xilinx.com/tools/microblaze.htm>

Appendix A. Guide to Codes

This appendix will provide a guide to understanding the codes developed during the course of this work, viz. the high level C model, the Verilog RTL for the NoC communication infrastructure. This guide may be utilized to use and adapt the codes for future needs. Refer to Figure 3 for a depiction of the levels of abstraction, each of them represents.

A.1 High Level C Model

As depicted on Figure 3, this model contains a high-level model of the hardware accelerator, and an application based on parallel genetic algorithm, implemented using three POSIX threads, called “dispatcher”, “gatherer” and “genetic operator” (Ref: Section 4.3.2). A pseudo-code is presented below to explain the organization. Compare it with Figure 13, which represents the corresponding logical structure of the system.

```
void* Gen_operator (void *dummy) { // “genetic operator” thread
int gen[NUM_POP]; // Current generation of each population
int pop = 0;       // Serial no. of current population to process

while(1) {
// Proceed only if all the packets for current population have been received in
//    population table
   if(pop_tab_cnt[pop] = POP_SIZE) {

// enable_migr is whether there should be a migration in current iteration and
//    population, but its calculation is not shown here.
// Sort the packets in population table for population no. “pop”, and return
//    the number of packets in it which have had the cost successfully computed,
//    if there weren't enough of them to derive new generation, else replace old
//    packets with new generation and return -1.
// At each call of apply_g_ops(), migr_cand is updated with packets in
//    pop_tab[pop] available as migration candidates for the next population.
      num_success = apply_g_ops(pop_tab[pop], enable_migr, migr_cand);

      if(num_success == -1) { // New generation available in pop_tab[pop] :
                              // Elit part of the population in the beginning
         pop_tab_cnt[pop] = ELIT_NUM;
        gen[pop]++; // Generation of present population advanced

// Elit packets are not sent out . The rest are written into dispatch-queue
         for(int i=ELIT_NUM; i<POP_SIZE; i++) {
            copy_ftr(pop_tab[pop][i], &ftr_tmp);
            write_queue(&dispatch_queue, ftr_tmp);
         }

      } else {  // Too many ejected packets preventing deriving new generation
                //    Unsuccessful packets to be resent for finishinng.
                // pop_tab[pop] has been sorted; with unsuccessful packets at end
         pop_tab_cnt[pop] = num_success;

         for(int i=num_success; i<POP_SIZE; i++) {
            copy_ftr(pop_tab[pop][i], &ftr_tmp);
            write_queue(&dispatch_queue, ftr_tmp);
} } } } }
```

The function prototype of write_queue() and apply_g_ops() have been somewhat simplified, and

some initializations, end conditions, mutual exclusion, loop iteration control, etc. have been omitted for the sake of clarity.

```
void* Dispatcher(void *dummy) { // "dispatcher" thread
ftr_t *ftr;
unsigned long long clk_num = 0;
unsigned long long next_eject = EJECT_PER; // Ref: Section 4.3.4 for
EJECT_PER
unsigned long long next_inject = 0;
int num_act_packets = 0;  // Number of packets inside the hardware

while(1) {
//-------------- Generation of outputs from hardware -------------------//
   if(next_eject <= next_inject) {  // It's not yet time to inject
      clk_num = next_eject;
      next_eject += EJECT_PER; // Next clock number when data will be output
// Simulation clock tick is aligned to the end of an EJECT_PER period. Now,
//    packets will be output from re-ordering queue to HW output queue.
      while((ftr=read_oqueue(clk_num)) != NULL) { // Any packet with finish
                   // time before clk_num is output normally
         write_queue(&hw_output_queue, ftr);
         num_act_packets--;
// Dispatch rate adaptation logic informed of successful completion of packet
         adapt_disp_rate(&throughput_adp, false);
      }

// Drop packets using the the logic in Section 4.3.4
      while((num_act_packets-EJECT_THR0)/ (float) (EJECT_THR1-EJECT_THR0) >
rand() / (float) RAND_MAX) { // Keep on ejecting till ejection fails once
          ftr = eject_oqueue(num_act_packets); // Random packet dropped
          num_act_packets--;
          write_queue(&hw_output_queue, ftr);
// Dispatch rate adaptation logic informed of a dropped packet
          adapt_disp_rate(&throughput_adp, true);
      }
   }

//--------------- Feeding one packet into hardware ----------------------//
   else if(dispq_h != NULL) {  // Dispatch queue not empty
      clk_num = next_inject;
      next_inject += throughput_adp; // Next clock no. for data-input to HW

      ftr = read_queue(&dispatch_queue);
      num_act_packets++;

//--------------- Computation inside hardware ----------------------//
      diff_time = traj_cost(ftr); // Compute trajectory cost, and return the
// time HW would take for this computation using the logic in Section 4.3.2
// Write this packet into the output reordering queue stampling it with
//    its expected finish-time
      write_oqueue(ftr, clk_num+diff_time);
   }
} }
```

As before some of the function prototypes have been simplified and loop control, end condition check, corner case handling, etc. have been omitted. The adapt_disp_rate() function maintains an internal FIFO of static booleans containing the status (successfully finished vs dropped) of the recent output packets from the hardware. It uses this information to adat the dispatch rate ("throughput_adp") using the algorithm described in Section 4.3.5. The parameter DISP_ADP_PER

described there controls the depth of the FIFO maintained by adapt_disp_rate().

```
void* Gatherer(void *dummy) { // "gatherer" thread
ftr_t *ftr;

while(1) {
   if(hwop_h != NULL) { // HW OP available
      ftr = read_queue(&hw_output_queue);

// Compute the population and the serial number of the packet inside
//    that population using the packet ID
      int pop = (ftr->ID/POP_SIZE)%NUM_POP;
      int idx = (ftr->ID)%POP_SIZE;

// ftr is put into the slot implied by its ID - also dropped packets
//   (i.e. those with no_via > 0 when received by this thread)
      pop_tab[pop][idx]-> data = ftr->data;
      pop_tab[pop][idx]-> no_via = ftr->no_via;

      delete ftr;

      pop_tab_cnt[pop]++;
} } }
```

These functions have been defined in the "genetic.cpp" file, with the funtions called by them are in "genetic.cpp" file (those that share mutexes), "aux.cpp" file (most of the other functions) and "traj_cost.cpp" (traj_cost() function and the functions it calls). The main() function contains the logic to initialize the populations in the population table as well as to put them into the dispatch-queue. After this initialization, the main() function launches the 3 threads described above.

Using the Model

The model is written in C++ (mostly using the C subset), but utilizes the POSIX libraries for thread management. Therefore, it can be run only on a system that has support for POSIX threads. In order to compile with POSIX thread "-pthread" option is required by "g++" compiler. The compilation command is however available in the "runme" file, and hence can simply be sourced to compile the code:
> source runme

To run the code
> ./a.out
or to run with a specific simulation seed
> ./a.out <seed>

When thus run, it reads the simulation and design parameters available in simparam.h and swparam.h and runs the simulation accordingly. The simulation output contains the used "simulation seed" at the very top. This can be used to reproduce the same simulation run later, as explained above. It also prints the cost of the best trajectory in every population every 20 iterations of the genetic algorithm. Thus, the progress of the convergence of the results can be observed. Additionally, these runs will dump the following log-files:

- adapt_dispatch.log
- load_profile.log

These two logs can be visualized graphically by using gnuplot, displaying the evolution of the dispatch rate, and the number of active packets inside the hardware over the period of the simulation. In order to do that, two simple gnuplot scripts have been written which can be called, as

follow. To plot the dispatch rate adaptation profile:
> gnuplot plotscript_adapt

and to plot the HW load profile:
> gnuplot plotscript_load

As explained above, the simulation and the design are specified primarily by the files "simparam.h" and "swparam.h". Some of the more important parameters are explained here, and they can be varied to simulate different scenarios:

```
// Ref: Section 4.3.2
#define BASE_PROC_TIME 70 // Basic data processing time for 1 packet
                          //    in a processing node (in clocks)
#define VIA_PROC_TIME 20  // Extra routing calculation time for every extra
                          //    via point inside the segment (in clocks)

// Ref: Section 4.3.1
#define POP_SIZE 100  // Total number of packets in each population
#define NUM_GEN 200   // Total number of GA iterations run. Ideally should be
                      //    equal to the num of generations, which may be
                      //    lower due to packet recomputation
#define NUM_POP 10    // Total number of populations
#define ELIT_PERC 5   // Percentage of total population that is elite
#define MUT_PROB 0.5  // Probability of gene mutation
#define INSERT_PROB 0.1  // Probability of gene insertion
#define MIGR_ENABLE true // Migration is enabled
#define MIGR_GEN_GAP 10 // Number of generations between two migration events
#define MIGR_PERC 2 // Percentage of population that is selected for
                    //    migration to another population.

// Min. percentage of pop_size that must be available (including elite
// packets and currently computed packets) to derive the next generation from
#define SUCC_PERC 0.8 // Ref. Section 4.3.2

// Ejection rate control - HW . Ref Section 4.3.4
#define EJECT_PER  THROUGHPUT
#define EJECT_THR0 20
#define EJECT_THR1 25


// Initial throughput in clocks/packet: Adaptation mechanism may change
//    the actual throughput . Ref Section 4.3.5
#define THROUGHPUT 200

// Dispatch rate control - SW/HW . Ref Section 4.3.5
#define DISP_ADP_ENA true // Dispatch rate adaptation is enabled
#define DISP_ADP_PER 31  /* In terms of number of output packets */
#define MIN_EJ_RATIO 0.05
#define MAX_EJ_RATIO (1-SUCC_PERC)
#define RELAX_RATE 1.0
#define TIGHT_RATE 0.1
```

The cost-data for the map are defined as an array in the beginning of the "traj_cost.cpp" file as a constant array:

```
const float cost[...][...] = {{} … {}};
```

This needs to be updated to update the map data.

A.2 Verilog RTL Design

The hardware architecture of the NoC communication infrastructure has already been described in detail in Chapter 6. The present section describes the module hierarchies in the RTL code that have been used in synthesis and verification, and correlates them to the blocks in the afore-mentioned design description.

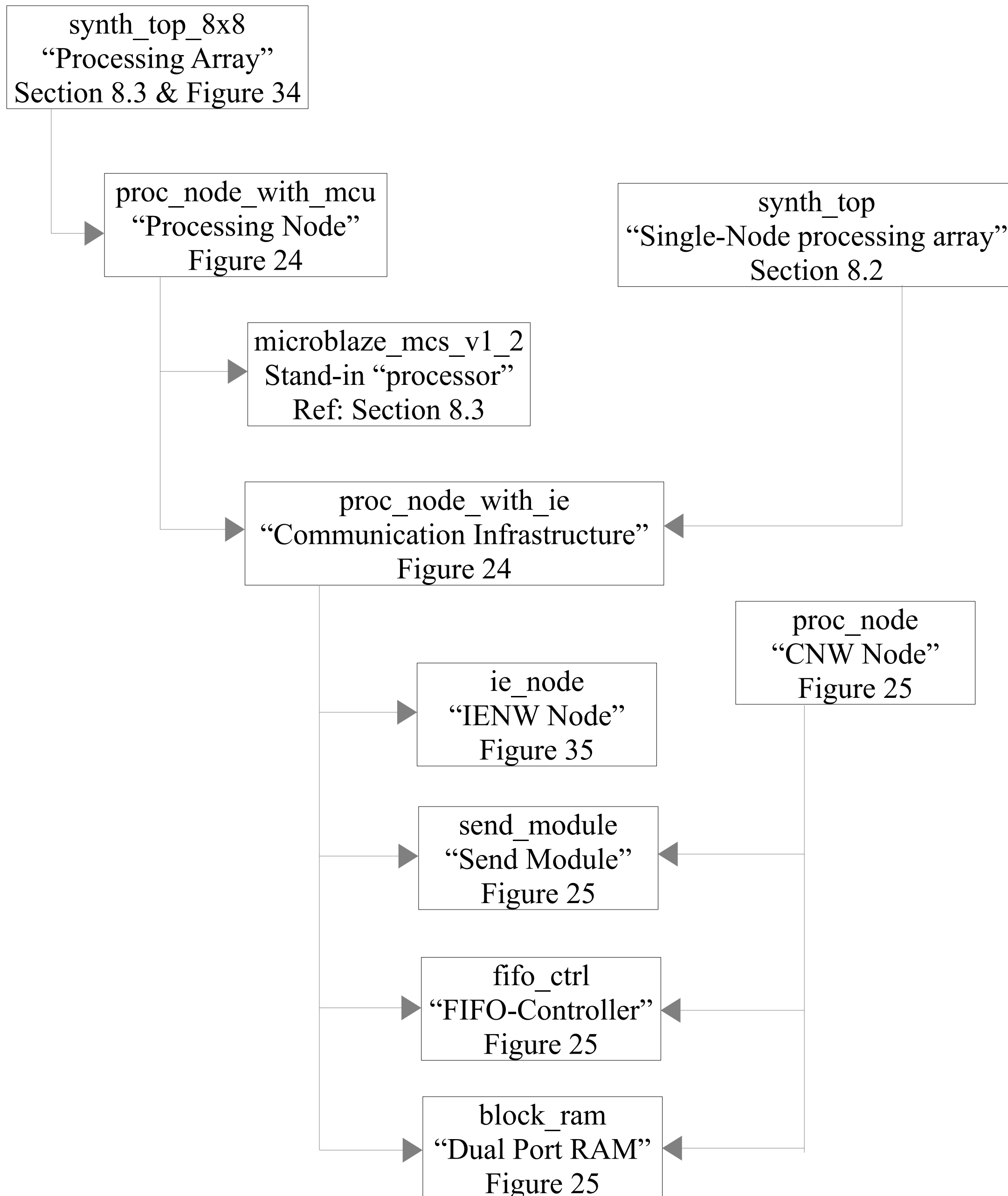


Figure 46: RTL hierarchies. In each entry, the first line gives the Verilog module name and the second line gives the corresponding design block, whose reference is given in the third line.

A.3 SystemC Model

As explained in Chapter 2, the SystemC model for the hardware accelerator was developed in autumn 2013 semester [11]. It was re-used in the present semester to assist in developing the high level hardware model using the simulations presented in Section 5.2. Slight modifications were made in the test-bench of the SystemC model for running these simulations, but the model itself remained unchanged, as did the regression environment developed around the model.

In order to run a regression (i.e. many simulations in sequence), the regress.sh file needs to be updated, which looks something like the following:

```
gcc -o new_simparam new_simparam.c
./new_simparam 10000, 20, 70, 700, 19, 19, 190, 190, 1500; source ./script.sh
```

The first line compiles a C program that generates the simparam.h file which controls the synthesis runs. All subsequent lines have the same structure, the first part calls this program to generate a simparam.h file, and the second part (source ./script.sh) runs the corresponding simulation, and at the end of that run moves the corresponding simparam.h and the log files into the "results" directory after appending .x to their names, where x is the run number read from the file run_num.dat, and incremented by every call to new_simparam. The line "./new_simparam 10000, 20, 70, 700, 19, 19, 190, 190, 1500" generates a simparam.h file with the following parameters:

```
const sc_time BASE_DL_TOUT (10000, SC_NS); // Basic timeout for deadlock detect
const sc_time DL_TOUT_COST (20, SC_NS);  // Time out increment for each unit
                                         // cost already invested into the packet
#define NUM_PACK 1500  // Number of packets in simulation
#define THROUGHPUT 700 // Number of clock cycles between launching 2 packets

#define XSIZE 10  // Xsize of processing array
#define YSIZE 10  // Ysize of processing array

#define XLEVELS 19  // Number of layers due to folding along X axis
#define YLEVELS 19  // Number of layers due to folding along Y axis
// Map Xsize = 19x10 = 190; Map Ysize = 19x10 = 190

#define BYTES_FIFO_DEPTH 70  // Depth of FIFO's in bytes.
```

Once the regress.sh file is ready, it can simply be sourced to run the full regression as:
> source regress.sh

In order to run the test-cases emulating the converging GA runs, as done in Section 5.2, the testbench.cpp was updated to produce 4 via trajectories with the middle two via-points chosen in a constrained random manner:

```
test_ftr = new ftr_t(id, num_via, via_x, via_y, false); // num_via = 4
                                  // (via_x, via_y) is the start point

// The middle 2 via-points are within two unit squares:
//    (intx1.<>, inty1.<>) and (intx2.<>, inty2.<>). Ref: Section 5.2
   test_ftr->via_x[1] = intx1 + rand()/((float) RAND_MAX);
   test_ftr->via_y[1] = inty1 + rand()/((float) RAND_MAX);
   test_ftr->via_x[2] = intx2 + rand()/((float) RAND_MAX);
   test_ftr->via_y[2] = inty2 + rand()/((float) RAND_MAX);
```

Appendix B. Selected Simulation Data

This appendix will list out the simulation results that have been presented only graphically on Figures 19-21 (Section 5.3) for easier visualization. These were based on 4 sets of simulation runs, comprising of 2 different population configurations (1 population of 100 packets and 2 populations of 50 packets), each run under two different packet dropping/ejection probabilities (viz. 0.0 and 0.9). Each row below displays the cost of the best trajectory (i.e. the trajectory with the minimum cost) among all the populations being used during a specific simulation run after the elapse of specific numbers of generations of the concerned populations.

Case 1: 1 Population of 100 packets; ejection probability = 0.0

No. of elapsed generations		19	59	99	199
Cost of the best trajectory	Run 1	186.47	173.13	170.35	168.80
	Run 2	186.81	172.91	170.38	169.43
	Run 3	194.06	172.30	169.45	169.09
	Run 4	182.98	172.00	169.35	168.87
	Run 5	183.55	173.67	171.03	169.31
Best Cost		**182.98**	**172.00**	**169.35**	**168.80**
Average		**186.77**	**172.80**	**170.11**	**169.10**
Estimated Standard Deviation		**4.415**	**0.665**	**0.705**	**0.272**

Case 2: 1 Population of 100 packets; ejection probability = 0.9

No. of elapsed generations		19	59	99	199
Cost of the best trajectory	Run 1	204.55	179.84	172.66	170.14
	Run 2	245.39	222.63	193.45	170.05
	Run 3	192.43	179.11	174.30	169.51
	Run 4	200.30	180.58	172.02	170.48
	Run 5	193.74	179.52	176.40	170.67
Best Cost		**192.43**	**179.11**	**172.02**	**169.51**
Average		**207.28**	**188.34**	**177.77**	**170.17**
Estimated SD		**21.865**	**19.178**	**8.930**	**0.446**

Case 3: 2 Populations of 50 packets each; ejection probability = 0.0

No. of elapsed generations		19	59	99	199
Cost of the best trajectory	Run 1	176.62	174.04	170.38	169.12
	Run 2	189.37	172.28	170.65	169.35
	Run 3	176.81	170.89	169.71	169.11
	Run 4	190.70	175.61	171.91	169.31
	Run 5	197.36	177.71	174.69	170.65
Best Cost		**176.62**	**170.89**	**169.71**	**169.11**
Average		**186.17**	**174.11**	**171.47**	**169.51**
Estimated SD		**9.149**	**2.689**	**1.970**	**0.648**

Case 4: 2 Populations of 50 packets each; ejection probability = 0.9

No. of elapsed generations		19	59	99	199
Cost of the best trajectory	Run 1	188.12	176.34	173.35	172.60
	Run 2	219.62	181.26	176.94	170.65
	Run 3	228.51	181.83	178.23	175.50
	Run 4	209.57	179.85	174.94	171.42
	Run 5	205.43	184.77	175.85	172.20
Best Cost		**188.12**	**176.34**	**173.35**	**170.65**
Average		**210.25**	**180.81**	**175.86**	**172.47**
Estimated SD		**15.286**	**3.075**	**1.866**	**1.850**

These results were chosen from among many other simulation runs with many different configurations, and also only those runs which converged towards the global minimum, avoiding the other local minimum, as explained in Section 5.3, because otherwise the results could not be compared properly. The results presented here may be deemed somewhat unrealistic because of the relatively small population-size. More realistic data was obtained by simulating systems where the performance of a single 1000 packet population was compared with that of 10 populations of 100 packets each. The results show similar trends as above, but the differences were less dramatic, and hence not as amenable to graphical representation. In any case, the data itself is presented on the next pages for the sake of completeness.

Case 5: 1 Population of 1000 packets; ejection probability = 0.0

No. of elapsed generations		19	59	99	199
Cost of the best trajec-tory	Run 1	176.77	169.64	168.87	168.53
	Run 2	176.65	169.34	168.93	168.70
	Run 3	176.09	169.32	168.84	168.80
	Run 4	181.00	169.39	168.85	168.65
	Run 5	176.79	170.65	168.85	168.54
Best Cost		**176.09**	**169.32**	**168.84**	**168.53**
Average		**177.46**	**169.67**	**168.87**	**168.64**
Estimated SD		**1.999**	**0.564**	**0.036**	**0.113**

Case 6: 1 Population of 1000 packets; ejection probability = 0.9

No. of elapsed generations		19	59	99	199
Cost of the best trajec-tory	Run 1	186.15	174.26	171.01	169.43
	Run 2	179.34	172.91	170.44	169.01
	Run 3	190.82	173.85	172.02	169.47
	Run 4	186.41	173.78	169.89	169.27
	Run 5	196.23	174.92	171.73	169.63
Best Cost		**179.34**	**172.91**	**169.89**	**169.01**
Average		**187.79**	**173.94**	**171.02**	**169.36**
Estimated SD		**6.253**	**0.734**	**0.883**	**0.235**

Case 7: 10 Populations of 100 packets each; ejection probability = 0.0

No. of elapsed generations		19	59	99	199
Cost of the best trajec-tory	Run 1	179.94	170.20	169.02	168.57
	Run 2	181.24	170.18	168.88	168.65
	Run 3	182.61	170.30	169.23	168.74
	Run 4	178.05	171.11	168.79	168.62
	Run 5	183.00	170.79	169.28	168.71
Best Cost		**178.05**	**170.18**	**168.79**	**168.57**
Average		**180.97**	**170.52**	**169.04**	**168.66**
Estimated SD		**2.029**	**0.415**	**0.213**	**0.068**

Case 8: 10 Populations of 100 packets each; ejection probability = 0.9

No. of elapsed generations		19	59	99	199
Cost of the best trajec-tory	Run 1	186.13	174.37	172.31	169.45
	Run 2	183.44	174.72	171.32	169.25
	Run 3	195.06	174.32	172.14	169.18
	Run 4	192.25	175.10	172.23	169.57
	Run 5	184.91	174.87	170.87	168.82
Best Cost		**183.44**	**174.32**	**170.87**	**168.82**
Average		**188.36**	**174.68**	**171.77**	**169.25**
Estimated SD		**5.028**	**0.332**	**0.643**	**0.288**