UNIVERSITY OF SOUTHAMPTON

FACULTY OF PHYSICAL SCIENCES AND ENGINEERING

Electronics and Computer Science

# Parallel Discrete Event Simulation on the SpiNNaker Engine

by

**Chuan Bai**

Thesis for the degree of Doctor of Philosophy

May 2013

UNIVERSITY OF SOUTHAMPTON

# ABSTRACT

FACULTY OF PHYSICAL SCIENCES AND ENGINEERING

Electronics and Computer Science

Thesis for the degree of Doctor of Philosophy

## PARALLEL DISCRETE EVENT SIMULATION ON THE SPINNAKER ENGINE

Chuan Bai

The SpiNNaker engine is a multiprocessor system, designed with a scalable interconnection system to perform real-time neural network simulation. The scalable property of the SpiNNaker system has the potential of providing high computation power making it suitable for solving certain large scale systems, such as neural networks. In addition, biological neural systems are intrinsically non-deterministic, and there are a number of design axioms of SpiNNaker that made it ideally suited to the simulation of systems with such properties.

Interesting though they are, the non-deterministic attributes of SpiNNaker-based simulation are not the focus of this thesis. The high computational power available, coupled with the extremely low inter-chip communication cost, made SpiNNaker an attractive platform for other application areas in addition to its principal goal. One such problem is *parallel discrete event simulation* (PDES), which is the focus of this work.

*Discrete event simulation* is a simple yet powerful algorithmic technique. *Parallel discrete event simulation,* on the other hand, is much more complicated due tothe increase in complexity arising from the need to keep simulation data synchronized in a distributed environment. This property of PDES makes it a suitable candidate for generic simulation evaluation. Based on this insight, this thesis carries out the evaluation of the generic simulation capability of the SpiNNaker platform using a specially built framework running on the conventional parallel processing cluster to model the actual SpiNNaker system. In addition, a novel load balancing technique was also introduced and evaluated in this project.

In memory of the greatest mother…

# Contents

# List of Figures

# Declaration of Authorship

I, Chuan Bai

declare that the thesis entitled

Parallel Discrete Event Simulation on the SpiNNaker Engine

and the work presented in the thesis are both my own, and have been generated by me as the result of my own original research. I confirm that:

- this work was done wholly or mainly while in candidature for a research degree at this University;
- where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated;
- where I have consulted the published work of others, this is always clearly attributed;
- where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work;
- I have acknowledged all main sources of help;
- where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself;
- none of this work has been published before submission, or [delete as appropriate] parts of this work have been published as: [please list references]

Siged: ……..

Date:…….

# Acknowledgements

I would like to thank the following people for their help and support throughout all these years:

Firstly, I would like to thank my brilliant supervisor Professor Andrew Brown, for his guidance and advice in the entire process. The single greatest thing that I got out of this project is his philosophy of approaching a problem. It is deeply stamped onto my heart and I will keep on following it for rest of my life.

My parents gave me their greatest support in good and bad times for all these years. They minimized the interruptions caused by the critical condition my mum had suffered for three years and I am grateful for all the love they gave me.

My wife was a great supporter who is simply brilliant and always level-headed. She is a gift from up above, and I am thankful for this.

Abdeldjalil Belouettar contributed a piece of the software that this project requires, and with whom I had many productive discussions.

Finally, thanks to all the others who helped me, Dr Chenxu Zhao, Dr Ruiqi Chen and Dr Dafeng Zhou and my colleagues in what used to be the Electronics Systems and Devices Group.

# Chapter 1 Introduction

Over the last few decades, the size of integrated circuits has grown rapidly and the task of understanding and modelling the behaviour of them has become an increasingly difficult job: for instance, a typical central processor unit (CPU) today would contain around 700M transistors. Although the transistor count increases exponentially according to Moore's Law [1], the speed at which we can model them has not increased proportionally. As the size of circuits has increased exponentially, the number of objects that need to be modelled in the development stage has also risen. However, the computation power available to simulate the model and behaviour of the circuit has not increased accordingly. One way to solve this problem is to parallelise the simulation and modelling.

One of the first and the most cited parallel techniques was created by K.M. Chandy and J. Misra [2]. They introduced a method to parallelize a simulation system, and others [3–5] subsequently developed different versions and improvements to Chandy-Misra method later on. However, the growth speed for the size of simulated circuits continued to outpace the computation capacity growth of this parallel simulation technique. Jefferson [6], [7] devised a brand new method to deal with this problem, which allows the system to execute the simulation as soon as possible at the cost of extra system resources and periodic synchronizations. However, soon this performance race between simulation power and its target forced the simulation system to move to more abstract levels. Circuit simulation focus had shifted from analogue simulation to digital gate level simulation, and then moved even further towards behavioural and functional simulation. The simulation speed improved dramatically, but it also brought the problem of lacking detailed information about the simulated system, which reduced the accuracy when the simulation was trying to determine the switching activity distribution and other detailed physical parameters.

The introduction of SpiNNaker system is a big step forward in this computational race. The SpiNNaker system consists of an array of ARM chips which are interconnected to form a powerful computation system. SpiNNaker has the ability to expand its computation power to match the increasing size of the target problem. It was designed to simulate neural network problems, but in this project we will exploit the ability of the SpiNNaker system to deal with discrete event simulation. The nature of these two simulation systems is very different: while neural networks are non-deterministic systems, discrete event simulation is a deterministic problem. In a deterministic system, the order of event execution for the same simulated object is consistent for every simulation, whereas a non-deterministic system accepts randomness in *execution* and does not necessarily produce the same order of execution for every simulation. By successfully designing and implementing a deterministic parallel discrete event simulation system for SpiNNaker, this project elegantly illustrates the capabilities of the SpiNNaker in generic simulation, by estimating the potential benefits that might be obtained through its highly scalable and high performance communication network.

Since the nature of this hardware is a massive parallel platform, the simulated problem must be capable of creating large complex computations in an automated and verifiable way. This is not hard to find given the expertise of our group, which has long-established experience in dealing with discrete event simulation that targets digital circuits. Since the 1990s, our research group has been capable of designing and implementing a behavioural hardware description language synthesis tool. By bridging this expertise with the new SpiNNaker platform, the quest for generating simulated problems is solved. Therefore, discrete event simulation was chosen as the simulated problem for this project.

There are different approaches to implementing a parallel discrete event simulator. In this thesis, three different types of simulation techniques will be examined and the Chandy-Misra technique was chosen to be the final candidate. To compensate for the conservative nature of the Chandy-Misra technique, a dynamic load balancing simulation technique is also brought into the final version.

The main ***hypothesis*** in this project is that the scalability in computational power and high performance communication network properties of the SpiNNaker platform should bring extra benefit in terms of performance in comparison with a conventional parallel cluster. Although the computational power of an individual processor in SpiNNaker is not as powerful as they are in a conventional cluster, the chip-to-chip communication network in SpiNNaker should outperform its peers in conventional clusters. This is because the additional hardware and software layers required by the conventional cluster slow down the responsiveness of the overall network.

The second ***hypothesis*** is that by breaking the conventional boundaries created during static partitioning, the dynamic load balancing process should be able to create a workload redistribution mechanism that is more dynamic as well as accurate. The conventional dynamic load balancing technique may create unnecessary boundaries between different partitions for any generic simulation targeting the SpiNNaker platform. This is because the size of a simulated problem mapped to each physical processor tends to be higher in a conventional simulation. A workload distribution mechanism that is optimized to deal with a more fragmented simulation problem should be devised for the SpiNNaker platform, and this hypothesis proposed a way to address this unique requirement.

Although the simulation system has been designed and emulated, the actual SpiNNaker hardware was not available throughout this project. Thus it was not possible to transfer the simulation system from the development cluster system to the actual SpiNNaker system. The results shown in this thesis are generated on conventional architecture computer clusters and desktop computers. However, the speedup introduced by SpiNNaker can be estimated by comparing the performance in these environments and the design specifications of SpiNNaker.

This project made two main contributions:

- Exploring generic simulation software capability of the special purpose built SpiNNaker platform. This project emulates the data structures, communication patterns and simulation control of the SpiNNaker platform, and provides a guideline for future generic simulation porting to this new platform.
- Creating and evaluating a novel algorithm to implement a dynamic load balancing (DLB) technique which can accurately reallocate the workload of conservative simulation by breaking the static partitioning boundaries.

In chapter 2, the background information about the SpiNNaker platform, different types of parallel simulation techniques, and the basis of how to manage a distributed system using message passing interface (MPI) are explained in detail. An introduction to the basic data structures involved and an investigation of different simulation technique are discussed in chapter 3. After the discovery process, the implementation of SpiNNaker Discrete Event Simulation (SDES) is revealed in chapter 4. The performance results of SDES are shown and discussed in chapter 5. Final comments and a future vision for the development follow in chapter 6.

# Chapter 2 Background

## 2.1 *SpiNNaker*

SpiNNaker (Spiking Neural Network Architecture) is a massively parallel, interrupt-driven, multi-processor computing system, comprising over a million ARM9 cores embedded in a hardware packet-passing communication infrastructure. The SpiNNaker engine [8] consists of a set of 65000 nodes, each node realised as a single silicon die. Each node consists of 18 ARM9 cores (processors), each tightly coupled to 64Kbytes of local data memory (TCDRAM) and 32Kbytes of instruction memory (TCIRAM), a multicast communications router [9] (six self-timed inter-node bidirectional links, plus an associative routing table), 128Mbytes of SDRAM, an Ethernet module, and boot, test and debugging interfaces - Figure 1 [10]. (65000 x 18 gives 1170000 cores; 65000 x 128M gives 8320Gbytes.) The nodes are connected in a triangular mesh (a two-dimensional rectangular plane where the opposite edges identify, giving a toroidal computing surface) - Figure 2 [11], [12]. A few of the nodes have their Ethernet ports connected to the outside world, supporting conventional access to the machine, although the bandwidth of this channel is clearly extremely limited. Of the 18 cores on each node, one is arbitrarily selected to be the **monitor processor**, which plays a special part in the routing infrastructure described below.

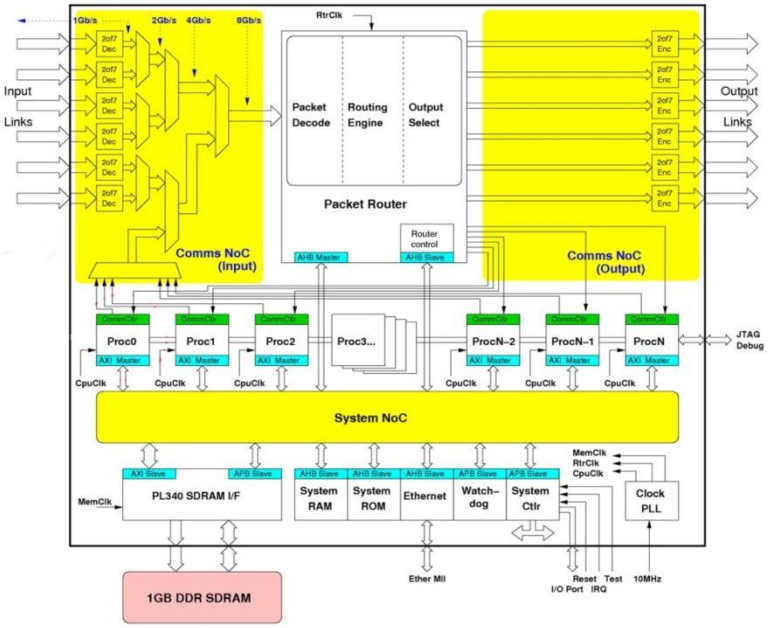



Figure 1 SpiNNaker block diagram

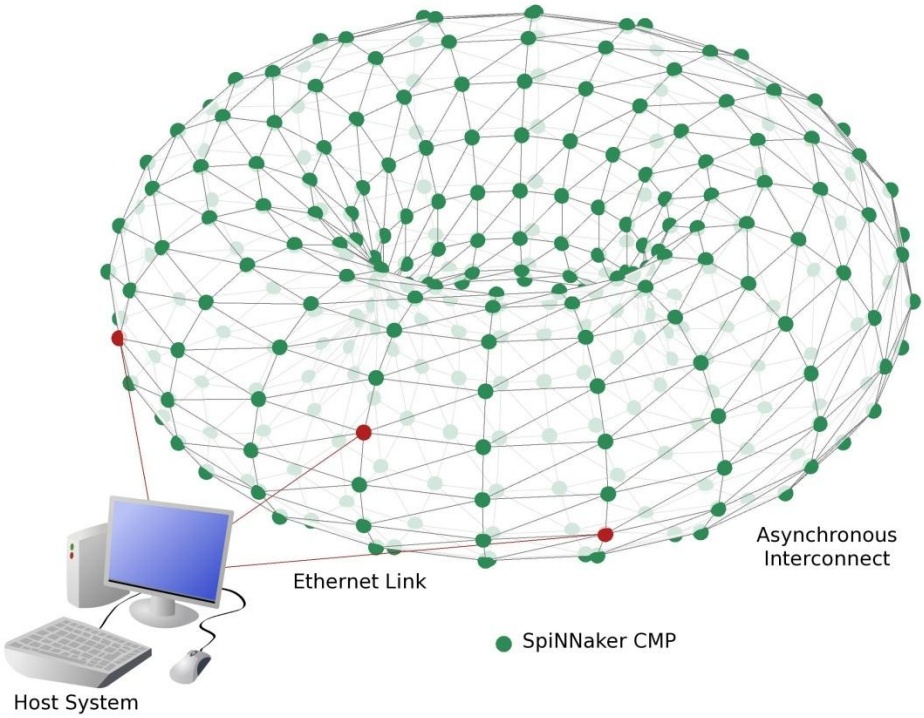



Figure 2 SpiNNaker machine

SpiNNaker is designed to perform artificial neural network simulations that support a network size ranging from thousands to millions of neurons with varying degrees of connectivity. The designed capacity of the overall SpiNNaker system is one million processing cores, which is expected to be able to simulate over a billion neurons in real-

time. The processing cores have independent functional and identical Chip-Multiprocessors (CMP) which provide sufficient memory and processing power to perform large-scale neural simulation in real-time. Each CMP is called a node in the context of this thesis. Performing the same simulation on a personal computer might take days.

### 2.1.1 Communication Mechanism

The principal communication mechanism between cores is by packet switching of a 64 bit hardware supported quantum of information that passes between nodes, controlled by the communications router. There are four types of communication, each handled independently by the router [13] (Figure 3).

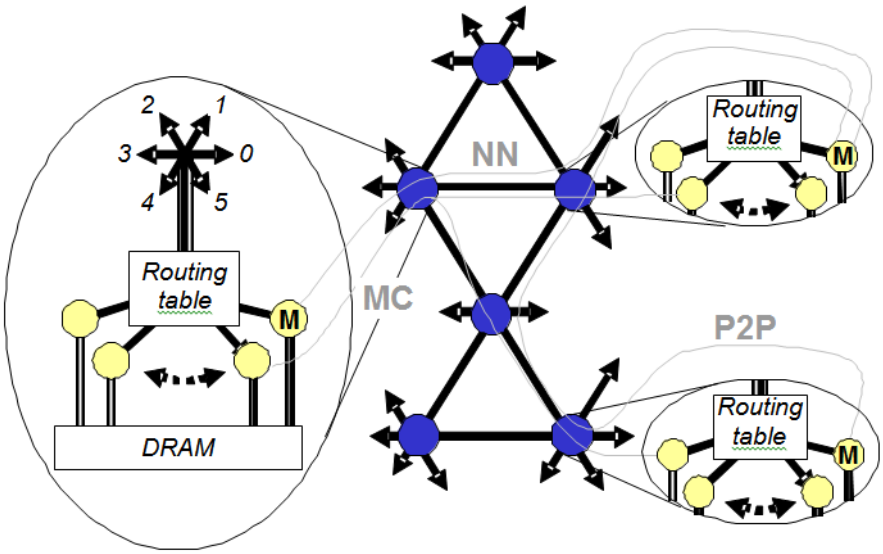



Figure 3 Packet mediated communication

There are four types of packets, which are the multicast (MC) packet, the point-to-point (P2P) packet, the nearest-neighbour (NN) packet, and fixed route packet (FR). The multicast routing mechanism enables a SpiNNaker core to communicate with multiple recipients which are the processing cores either situated locally or globally. The routing of the packets is defined by the MC routing table, which itself depends on the mapping between the simulated problem and the SpiNNaker topology. The multicast packet itself does not contain any information regarding its recipients; it includes just sender identification using a 32-bit address. This information is used by the router to choose the appropriate routing path for the multicast packet. The multicast routing table which

is defined in the initialization stage, which along with other aspects of the software configuration will be explained in section 2.1.2 . When loading a simulation problem onto SpiNNaker, the multicast routing table will be set up specifically for each simulated problem.

The P2P packet specifies both sender and receiver monitor processors. It can be initiated by both monitor and slave cores. The P2P packet contains a 16-bit source node address, a 16-bit target node address, plus an arbitrary 32-bit payload. It may be launched *from* any core, but will only be delivered to the monitor processor on the target node. The routing table for P2P communication is loaded in the booting stage which is independent from the simulated target problem. The network topology of the SpiNNaker system defines the P2P routing table, not the simulated problem.

The NN packet is passed from one node monitor processor to its neighbouring node monitor processor. Similar to the P2P packet, the NN packet can be initiated by either monitor or slave cores but is only delivered to the target monitor core. The routing for the NN packet is realized in hardware and requires no software initialization.

The FR packet is used as a fast routing mechanism *from* any core *to* the monitor core on the node containing a live Ethernet connection. It is an elegant way of passing information to the host system using minimal resource.

For multicast routing, the router stores 1024 programmable associative multicast routing entries, which are capable of dynamic modification. Multicast packets are routed according to these entries. The routing entries are masks that are compared with the routing target information contained within an incoming message. The masked result will be looked up in the multicast table, which outputs a one-hot encoded routing indicator. Each bit in the routing indicator enables or disables the message to be passed on to its corresponding connection, as shown in Figure 4.

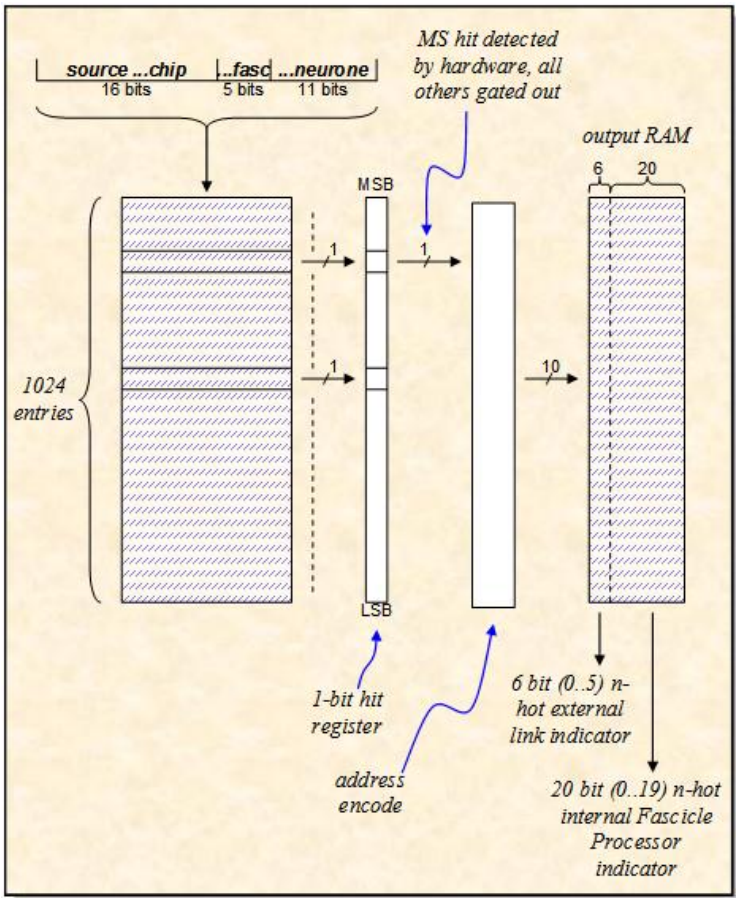



Figure 4 Multicast routing [10]

For P2P packets, router will establish the routing direction for P2P packets automatically. This routing information is stored in a 256 entry x 24-bit SRAM lookup table. The first 16-bits are the address masking bit and the rest is the encoded output and its control bits. In the end, a P2P packet will be processed to point to one of the entries shown in Figure 5. If a packet is targeting an external SpiNNaker node, the P2P routing will produce an index between zero and five, whereas if the P2P packet target is the current node, the packet will be routed to the monitor core on the current SpiNNaker node.

| Table Entry | OP Port | Direction |
|---|---|---|
| 000 | Tx0 | E |
| 001 | Tx1 | NE |
| 010 | Tx2 | N |
| 011 | Tx3 | W |
| 100 | Tx4 | SW |
| 101 | Tx5 | S |
| 11X | Monitor | Local |

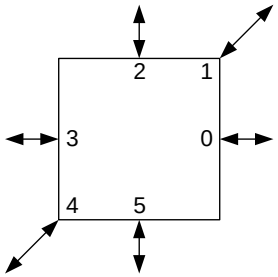



Figure 5 Link direction and P2P table entries

The NN packets are used to initialize the system and to perform run time flood-fill and debug functions. The NN packet is a responsive message. For example, if a read NN packet is sent, the return message will contain the requested data, whereas a write NN packet will receive an empty packet. Unlike MC packet processing, it does not require the use of the routing table to process the packet. Further information about the packet format is available in Appendix C.

### 2.1.2 Programming Model

SpiNNaker is an interrupt driven computation system. By default, cores are quiescent. When a packet arrives at a communication router, depending on the type and target data contained in the packet, it may be routed to one or more out-going ports on the node, and/or one or more local cores. When a packet arrives at a core, an interrupt is generated. The local core processes this packet and returns to sleep.

There are two initialization stages when preparing SpiNNaker for a simulation problem – booting and initialization stages. With conventional architectures, configuring the system to perform a simulation is trivial. Normally these architectures grant user the ability to specify the network topology and initialization data with a wide bandwidth, hence good data visibility. However, this is not the case for the SpiNNaker system. *Firstly*, the high core count in SpiNNaker makes it impossible to initialize the configuration in great detail. Assigning only a single byte configuration for each core will result in $10^6$ bytes of data, since the overall design capacity of the entire SpiNNaker platform is 1M cores. *Secondly,* the limited bandwidth to the host system via Ethernet makes data visibility almost non-existent. A more elegant technique, allowing SpiNNaker to configure itself is necessary.

SpiNNaker is a system still very much under development. As such, the software infrastructure is evolving at a rapid rate; things change. What follows is a description of a possible boot sequence. The *boot* stage deals with the hardware configuration when the SpiNNaker system starts up. Three sets of configuration data are created by the end of the boot stage. The ***α-ping*** phase creates a map of all working nodes within the SpiNNaker system after the system is started up. With a system size of a million cores, it must be assumed that faulty connections and nodes exist. The α-*ping* phase is initiated by an Ethernet-injected packet, which flood-fills and causes each node to ping its six neighbours along the way. Any non-responsive broken communication link will hence be identified. If a single node is dead, all six connections to this node will show up dead. At the end of this phase, a graph of all working nodes has been created.

The ***tree crawler*** phase embeds a tree structure into the toroidal node topology. As faulty nodes will exist in the SpiNNaker system, the assumption that all nodes will form a complete toroidal topology of network does not hold. This embedded tree structure provides an alternative access to all the nodes, when system-wide actions are required, although the tree crawler only uses local NN communications to form this tree structure of nodes. The root of this node tree structure is the node with Ethernet connection to the host computer. All other nodes are assigned a unique 16-bit identifier.

The ***P2P builder*** phase establishes a routing table in each node to determine the best way of passing a packet for an arbitrary pair of nodes. This is achieved by allowing each node to send out a computational wave front; along the paths a back pointer to the original node is stored in each passing node. This enables each local node to build up a table of which of its ports provides the best route to a given node, which enables the communications router to efficiently pass P2P packets without any intervention from any core.

Following the boot stage, a basic configuration of the hardware system has been established. The *initialization* stage maps the simulation problem to the node topology of SpiNNaker. By this stage, the SpiNNaker cores are equipped with the working node graph, NN and P2P packet routing mechanisms. The MC routing mechanism is not problem independent; the MC routing table configuration depends on the simulation

problem and the mapping of the simulated problem onto the working node graph. The MC routing mechanism allows a packet to be transmitted to an arbitrary number of recipients. Hence in the initialization stage, the MC routing table will be filled with routing entries generated by the simulation problem network topology.

The initial data setup and final result collection are performed via a few Ethernet connections in the mesh. These Ethernet connected nodes broadcast or gather data via point-to-point (P2P) or nearest neighbour (NN) packets.

Although this project is targeted at the SpiNNaker simulation platform, by the end of this project the SpiNNaker chip was not available to be used as the simulation platform. As a result, all the work and results are based on the emulation of the SpiNNaker platform on a computing cluster.

## 2.2 *Discrete Event Simulation*

Discrete event simulation is a technique that represents the *activity* of a discrete system as a time-ordered sequence of events. The events in the sequence change the state variables of the system at distinct, discrete times. For example, in a lift system, a press of a button in the lift is an event, and it changes the state of the lift from the *idle* to the *moving* state. When the lift reaches the target level, the lift opens the door and enters the idle state again. This is a classic example of a discrete event system. In this general discrete system, events are “button pressed” and “lift reached target level”. Both of them switch the state of the lift between idle and moving states. Thus the concept of the lift can be abstracted as a set of events and states.

A discrete digital circuit consists of wires and gates. Wires transmit signals that are driven by gates. Applying the analogy above to a discrete digital circuit, the events represent changes of output signal values of gates. The output events of gates will propagate to other gates and so on and so forth. The events in a circuit die out when they do not change the current value of the fan-out gates at the discrete time instant, or they reach the outputs of the circuit.

In modern high speed circuits (and most noticeably in asynchronous systems), the notion of system state is sometimes embodied in gate *topologies* as well as the gates themselves. In the context of this work, this is regarded as a modelling detail and is not discussed further.

A spectrum of techniques exists for the simulation of circuits (systems). *Continuous* and *discrete* simulations occupy widely-spaced positions in this spectrum; they are completely different. [14] Throughout this thesis, "circuit simulation" is taken to mean "discrete digital circuit simulation" unless stated otherwise.

### 2.2.1 Serial Simulation

Discrete circuit simulation is normally implemented using a queue-processing model. The basic simulation model consists of three components: the first component is the set of *state variables* in the circuit, which are the values stored in the gates. The second component is the *eventlist* that stores all the pending events within the system. The sources of these "future" events are the gates or inputs of the circuit. The third and last is the *current time* value. This is the reference point that is used to decide whether the event happens in the virtual past or in the virtual future. With these three parts, a simple discrete circuit simulation system can be implemented.

#### *Algebra Taxonomy*

Among the three key components, the state variables in discrete event circuit simulation are very different to the other variables in discrete event simulation. In digital circuit simulation, the *state* variables have two sets of data, which are the connectivity information and the gate information.

Connectivity information shows the relationship between different gates within the circuit. Gate information holds the logic type and state of a gate. Logic *gates* assert

logic levels that represent state components and logic *types* define the relationship both temporal and functional between the inputs and outputs.

During simulation a gate can only assert a finite number of logic levels. Furthermore, there are different standards for representing logic levels within a circuit. For example, there are Verilog HDL (Hardware Description Language, IEEE standard 1364) [15] and VHDL (Very-high-speed integrated circuit Hardware Description Language, IEEE standard 1164) [16] standards. They are the most popular hardware description languages for digital circuit modelling and provide a different set of logic values. These are a good starting point to find a suitable set of logic levels to represent the states of logic gates in the final circuit simulator, the SpiNNaker Discrete Event Simulator (SDES).

The Verilog standard has four values to define the states of a logic gate, which are '0' (Low), '1' (High), 'Z' (High Impedance) and 'X' (Unknown) [17]. Values '0', '1' and 'Z' correspond to assertion or de-assertion of a signal, and un-driven state. In reality, a signal will only have these three states, but simulators can support additional logic levels. The 'X' state represents unknown, in which the simulator cannot determine whether the value of the signal is '0' or '1'. The 'Z' denotes a three-state condition where a signal is disconnected from its driving gate.

The VHDL standard has nine states to describe the value of a signal, an additional (See Table 1). They represent an abstraction of a real voltage source or a condition that does not necessarily exist in reality.

| character | value |
|---|---|
| U | uninitialized |
| X | strong drive, unknown logic value |
| 0 | strong drive, logic zero |
| 1 | strong drive, logic one |
| Z | high impedance |
| W | weak drive, unknown logic value |
| L | weak drive, logic zero |
| H | weak drive, logic one |
| - | don't care |
| C | conflict |

Table 1 Standard VHDL logic values

The ‘U’ state means a gate has not been assigned with a value since the start of simulation. The value can propagate (like ‘X’), thereby providing the designer with an indication of the source of the value (which may not necessarily be enormous). The ‘-‘(don’t care) state is sometimes used in logic synthesis tools and test vectors [18]. The strong and weak drives depend on the output resistance of the gate driver (an equivalent circuit is shown in Figure 6). Illustration of high impedance and unknown states are shown in Figure 7.

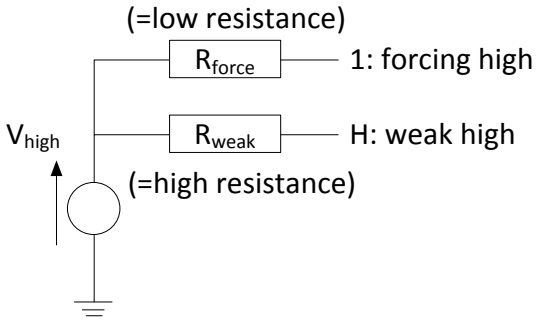



Figure 6 Abstraction of real voltage sources represented as logic levels

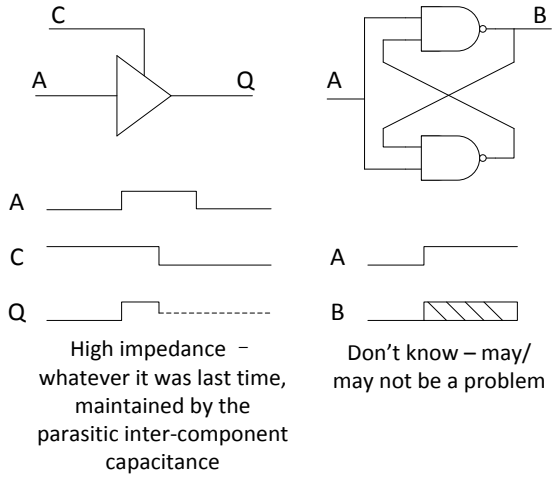



Figure 7 Metavalue cases

There is another state called 'C' which is conflict state. This state is only implemented in the ModelSim software [19] to distinguish from the 'X' (Unknown) state. Although it is not required by the VHDL standard, by introducing this extra state helps designers quickly identify this simple design fault. A sample circuit that can generate a 'C' state is shown in Figure 8.

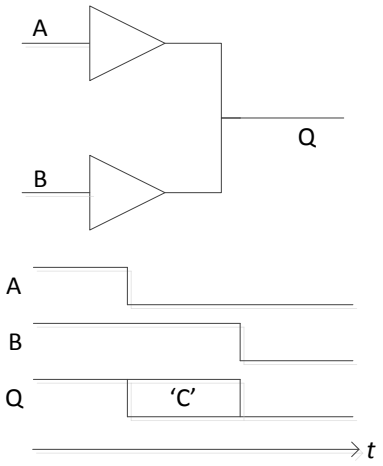



Figure 8 Conflict state case

In the SDES case, a key function of the simulator is to find an efficient way to simulate discrete event systems in parallel. Therefore, the complexity of the state information within the simulation is not a major concern of the SDES. Moreover, there is no tri-state logic in the SDES. As a result, only four states, which are '0','1','U','X', are chosen to be included in the logic state set within the SDES system. States '0' and '1' are the very

basics of digital circuits which cannot be replaced by any other states. The uninitialized value is used to assign each logic gate within simulated circuit at the start of each simulation. The unknown state is only employed whenever a state value is neither '0' nor '1' after a gate has its first value assignment, i.e. it presents the 'X', 'Z', 'W', 'L', 'H' and '-' states in the VHDL standard.

Based on the four values implemented in SDES, the truth tables for eight common types of logic gate can be defined. The truth tables for each type of logic gate are shown in Figure 9.

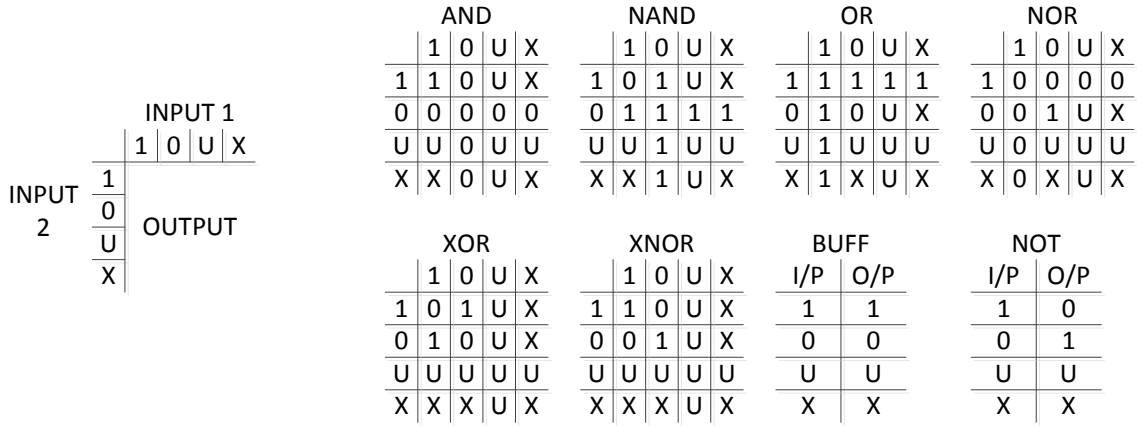



Figure 9 Truth tables for eight types of logic gate

### *Components in Discrete Event Simulation*

In this thesis, SDES only deals with digital electronic circuit simulation, therefore all the example simulated targets are digital circuits, but it does not mean that the SDES technique is limited to digital circuit simulation. The underlying principles are the same for all discrete event systems.

As an example, consider the simple D-type latch circuit shown in Figure 10. The latch circuit is a type of storage component within a digital circuit. The reason to choose a D-type latch as the example circuit is because it has a unique feedback loop and the difference between the minimum and maximum delay from input to output is significant,

which can provide a good demonstration for the behaviour of parallel simulation methods.

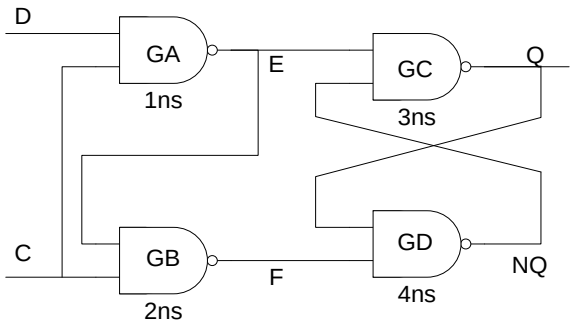



Figure 10 The D-type latch circuit

There are three types of variables involved during the simulation for the circuit shown in Figure 10, events, state variables and the current time. They describe the current status of the circuit.

***Events:***

The *events* are triples: {gate ID, logic level, execution time}. Take event "D = 1 @ 50 ns" as an example. It means that the output of gate D will change its value to '1' at the discrete time 50 ns. However, this is *not* an assertion that the value was not '1' before 50 ns. Events have to be executed in the ascending order of their timestamp to produce the correct result. This is the causality rule of circuit simulation [20]. It implies two properties: *first*, the events need to be stored in the same data structure for comparison purpose; *second*, events with the lowest timestamp will always be executed before any other events. This is the idea of an *eventlist*, which stores and sorts events into a queue of ascending events according to their timestamps.

Moreover, the time stamp for generated output events is calculated based on the timestamp of the causal input event *plus* the propagation delay of the generating gate. For instance, the event "D = 1 @ 50 ns" will generate an event for E @ 50+1 ns, the extra 1 ns is the propagation delay of the *nand* gate (GA). In reality, the propagation delays for logic gates within a circuit are different from gate to gate. Delays exist even between inputs on a supposedly balanced gate. They also vary as a function of past activity history, marginal difference in manufacturing process. Even in the simulation

world, the propagation delays of gates might differ from each other. As a result, a mixture of timestamps in output events will be fed into the eventlist. An event list is sorted in ascending order, the events with the lowest timestamp is located at the bottom (beginning) of a list. If two events share the same virtual time, the first added event will be at a lower position than the latter one. An example event list is shown in Table 2.

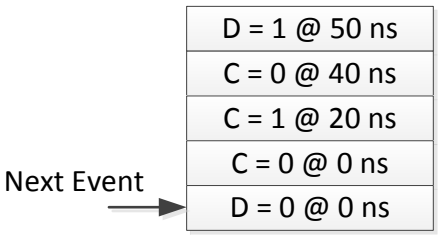


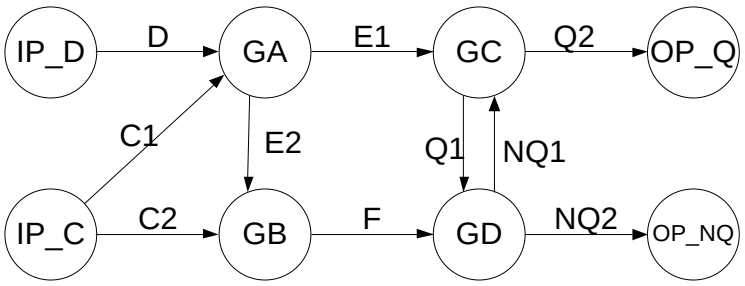

Table 2 Sample sorted eventlist for the D-type latch circuit

The software in this project is written in C++; Standard Template Library (STL) is used extensively as well. The *STL::list* container is the best choice for building the eventlist, compared with *vector* container, because the need to insert entries to the middle of the container [21]. Although the *STL::heap* container has an even lower insertion time $\Theta(log(n))$ compared with the list ($\Theta(n)$), the removal time for the *STL::heap* structure is also $\Theta(log(n))$. The removal time for list is $\Theta(1)$. In order to reduce the insertion time further, the simulation system collects the generated events *unsorted*, and insert them to the main eventlist in batch later on. This reduces the insertion search time, as new events are, in general, situated in a tight time zone.

***State Variables:***

The state variables in the system can be abstracted to a connection map and a table of gate properties. The connection map contains direction information on each of the connections, as a signal can only be driven by a single gate. Multiple gates driving a same signal are *not allowed* in SDES. This is prevented by a simple circuit design rule check. If any connection violates this rule, the program will stop the SDES from simulation. The directional information of connections matters, as it distinguishes the inputs and outputs for a gate. The most suitable container for a directed connection map is the directed graph (digraph). In a digraph, two sets of data are stored, nodes and arcs. The first set is the node information, where the detailed data of a node and the pointers of arcs related to the gate are stored (table of gate properties). The second set stores the arc information, which includes the data on the arc and the starting and ending node

pointers (connection map). Mapping a circuit to a digraph means the gates become the nodes and the wires become the arcs in the digraph. The equivalent graph for the latch circuit is shown below (Figure 11). In this project, digraph container in the STLplus library written by Andy Rushton is used [22].




| Gate Info. | GA | GB | GC | GD | IP_D | IP_C | OP_Q | OP_NQ |
|---|---|---|---|---|---|---|---|---|
| Gate Type | NAND | NAND | NAND | NAND | I/P | I/P | O/P | O/P |
| Logic Level | U | U | U | U | U | U | U | U |

| Arc Info. | C1 | C2 | D | E1 | E2 | F | Q1 | Q2 | NQ1 | NQ2 |
|---|---|---|---|---|---|---|---|---|---|---|
| Arc From | IP_C | IP_C | IP_D | GA | GA | GB | GC | GC | GD | GD |
| Arc To | GA | GB | GA | GC | GB | GD | GD | OP_Q | GC | OP_NQ |

Figure 11 Directed graph representation of the D-type latch circuit

There are two ways of representing a circuit as a graph: device-on-device (DoD) and device-on-nets (DoN). The nodes in the DoD format represent devices only, the wire information is stored in the arc. This causes the problem where multiple output arcs must hold a duplicated version of the wire information. This is not a problem for DoN representation, as both device and wire information is stored in the nodes of the digraph. Arcs in DoN representation only provide directional information. The latch circuit in DoN representation is shown in Figure 12.

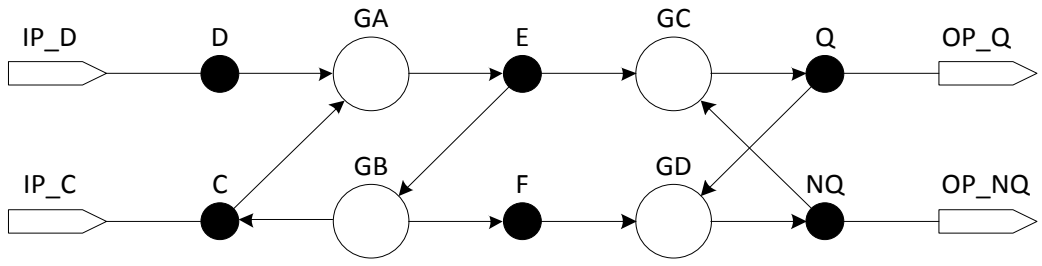


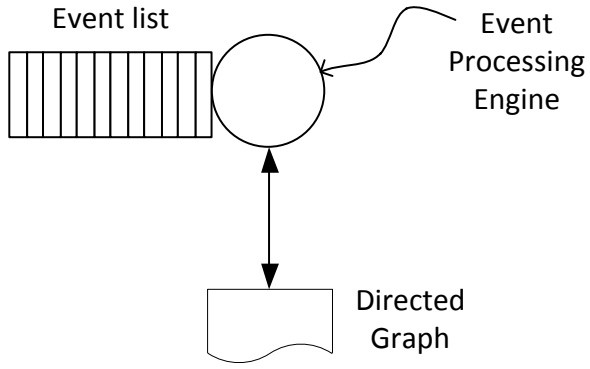

Figure 12 Device-on-Nets representation of latch circuit

It is easy to translate DoN to DoD but not the other way round. A gate with multiple discrete (separate) outputs signifies this problem. The reason for this project to choose DoD over DoN representation is because DoD is a more compact way of representing a circuit. And it also reduces the search candidates within the graph by half when the simulator tries to fetch the value of a fan-in gate.

***Current Time:***

The last component of the simulation system is the *current time*. In a serial or sequential simulation, this is simply usually a long integer. It provides a reference point for the simulator to distinguish past, present and future events. In the sample latch circuit, at virtual time 20 ns, the event "C = 0 @ 0 ns" is a past event which was discarded at the time of execution, whereas the event "C = 1 @ 20 ns" need to be processed and "D = 1 @ 50 ns" should be stored in the eventlist for future processing.

Simply stacking up events is meaningless without a processing unit. The simulation process has full access to all the components within the system. Its duty is to process all events within the system until the stoppage time specified by the user is reached or until the "future events" list is empty. When processing an event, it modifies the state value of logic gates and generates new events accordingly. After processing the generated events, any generated events will be fedback to the event list which will be sorted and processed later on. The abstraction is shown in Figure 13.




```
while (Q not empty) {
        pop_event();        // advance virtual T
        process_event();
        handle_event();
        sorted_insert();
        if (t > stop t)
                break;
}
```

Figure 13 Abstraction of the simulation system

### 2.2.2 Parallel Simulation

In *discrete simulation*, events genuinely may occur in parallel. In a simulation on a conventional sequential computing system, the event list and its associated handler constitute a processing bottleneck. When serial simulation speed cannot keep up with the ever growing size of simulation problem, a solution is to distribute the problem in parallel and let the collective computation power speed up the process. However, along with the speed that it brings, problems arise from the simulation technique itself, mainly because the need to maintain simulated causality throughout the distributed system.

For the rest of this thesis we will refer to the logical process (LP) as being the individual processing element in a parallel system. In general (but not necessarily) a LP is mapped onto a physical processor; a physical processor may host one or more LPs. In parallel simulation, the circuit under simulation (CuS) is partitioned between the LPs. Each LP maintains its own event loop. This partitioning immediately gives rise to a problem impacting on the three components outlined above.

To represent reality, events must be executed in global temporal order; but each individual processor only has knowledge of its own (local) sub-circuit. Each processor will be advancing simulation time with respect to wall clock time at its own rate, necessarily a function of the activity of its local sub-circuit. Wall clock time represents the passage of real time in the way that human perceives. Simulated time of a core processor can easily become wildly divergent.

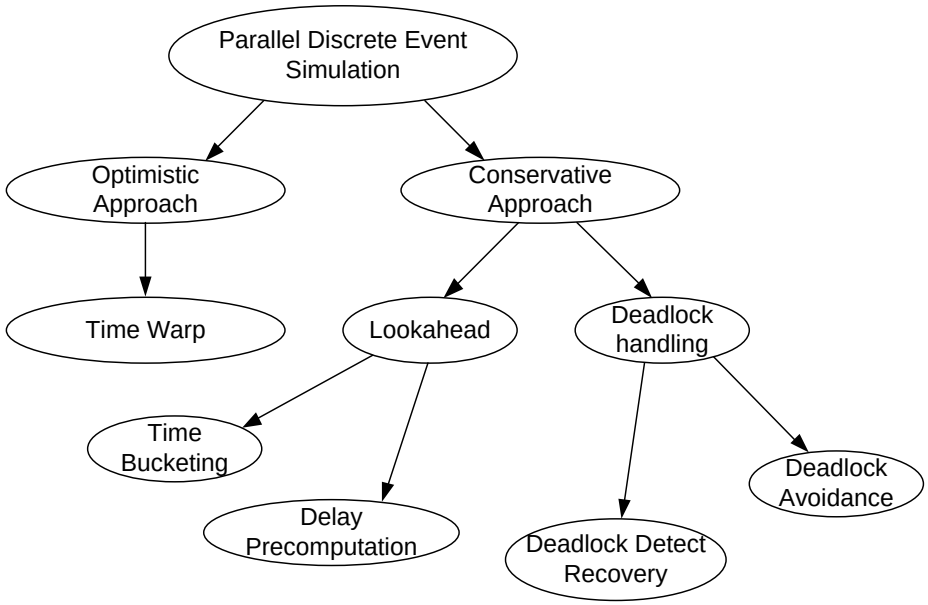



Figure 14 Tree of PDES approaches

In general, there is a spectrum of approaches. This spectrum (Figure 14) can be divided into two principal sections, conservative and optimistic.[23], [24]

The conservative approach guarantees the order of execution, which strictly follows causality between the events. By doing so, the problem outlined above will be solved automatically as the system is executing events sequentially. In conservative simulation, no LP, which performs part of a simulation in parallel, can execute an event that violates the causality rule. The *causality rule* states that every LP processes events in non-decreasing timestamp order. However, this method cannot fully utilize the parallel computation power. In most cases, the LPs will hold and wait for other LPs to send events for it to process, until a stop signal arrives.

The optimistic approach on the other hand, breaks causality to exploit the local parallelism inside a sub-circuit at the cost of additional system resource. It speculatively simulates, then conditionally restores and re-evaluates parts of the system to maintain the correctness of the simulation. The details of these techniques are explained in the sections below.

## *Common Problems*

As described above, a circuit simulation problem primarily consists of three types of variables. These are the state variables, events and the current time. When a system simulation is parallelized, these variables must be split into many parts. A local LP, which is mapped to a physical processor, usually holds the parts that are assigned to the local LPs (see Figure 15). This causes subsequent problems for each individual component when the simulator tries to simulate the system as a whole. These are solved in different ways in conservative and optimistic algorithms.

### *Order of Event Simulation*

After splitting a circuit, the *first* problem is the order of event execution. In serial simulation, new events are generated with full knowledge of the entire circuit state, and they are stored in the same event list. This is sorted in ascending order according to the time stamps, and the event with the lowest time stamp will always be at the head of the list, and will pop out first. When the queue is popped, the order of execution is guaranteed. However, after splitting the circuit into many parts, slower LPs in the system might generate events that happen in the virtual past of the other LPs. This causes the simulation system to break the causality rule and produce incorrect results.

The latch circuit can show the differences when simulating the events in parallel. Assuming that the circuit is divided into two parts: gate GA and GC, and gate GB and GD, the parts are mapped to LP A and LP B respectively (Figure 15).

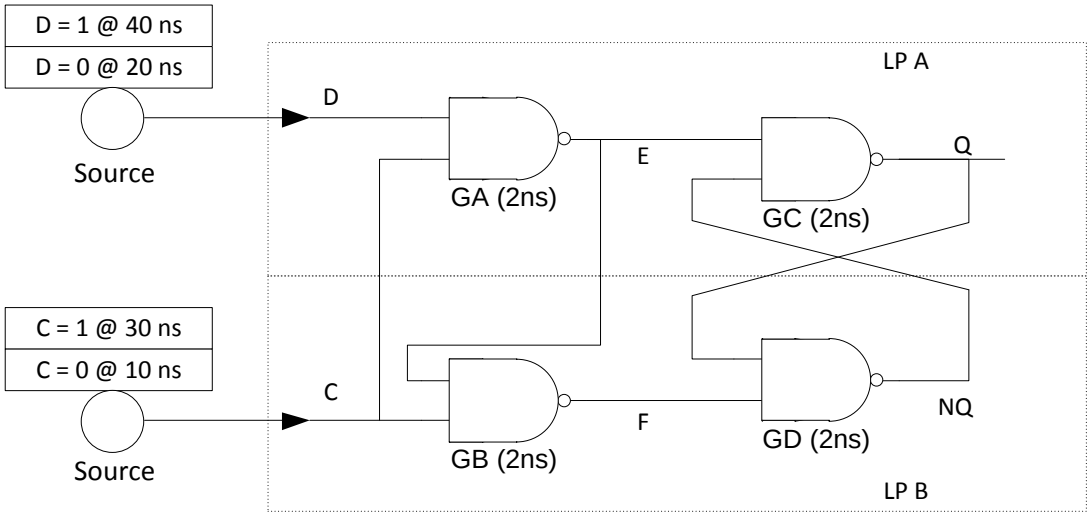



Figure 15 Abstraction of a parallel system

By evaluating events for gate GA and GB, the sequences of events are shown in a timeline in Figure 16. Take event "C = 0 @ 10 ns" as an example, it generates "E=1 @ 12 ns" (C nand D = '0' nand 'U' = 1) and "F = 1 @ 12 ns" (C nand E = '0' nand 'U' = 1). And then "E = 1 @ 12 ns" triggers another event "F = 1 @ 14 ns". As this event indicates a same value as the previous event "F = 1 @ 12 ns", no further processing is required.

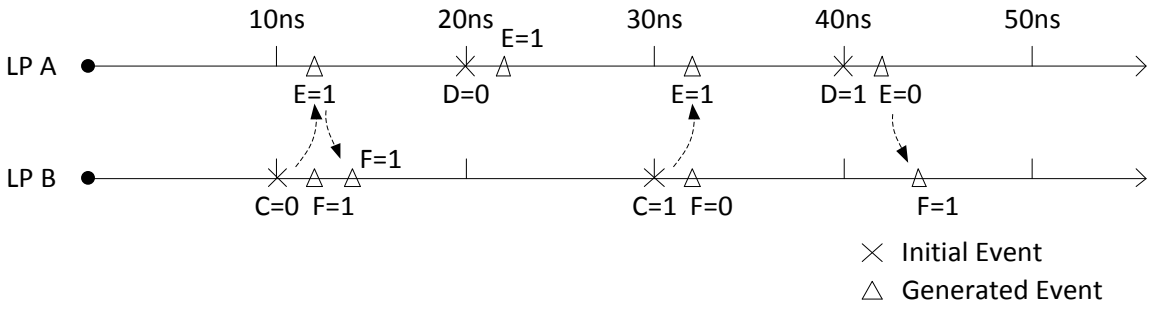



Figure 16 Events in timeline

However by isolating the LP A from LP B (Figure 17), the input events on D will cause "E = 1 @ 22 ns" and "E = U @ 42 ns" to be generated. However, if LP A processes events under an ASAP policy which can breach the causality rule, at virtual time 50 ns without any external events, wire E will show a value 'U', due to the fact that signal D and C carry value 1 and U respectively. However, this misses out the event "E = 1 @ 12 ns", which can be captured if LP A wait for this event to be executed before processing

event “D = 0 @ 20 ns” This is a typical example of a causality violation error during the simulation process.

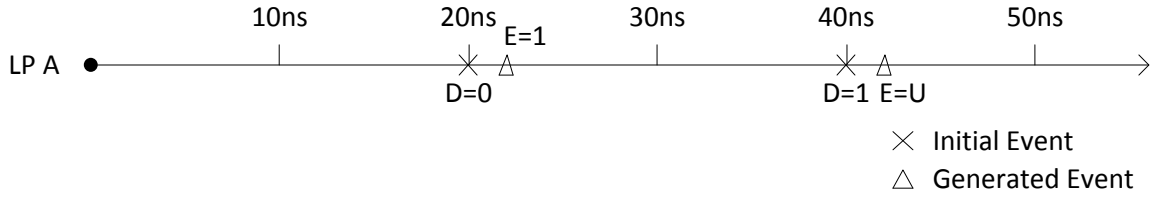



Figure 17 Isolated timeline in LP A

***Synchronization***

The *second* problem is the synchronization of current time, which is the current local virtual time on each LP. All events that have timestamps lower than this current local time are executed and discarded. Any event has a timestamp higher than this current time will be inserted in the event list. LPs only know their local virtual time, as they hold different state variable sets and pending tasks. The problem lies in how to make sure the execution order is correctly maintained across LPs, so that it does not violate the causality rule. In other words, how might one guarantee the simulation moves on monotonically (and correctly) and hence reaches the end of the simulation.

### *Optimistic Approaches*

The causality rule (order of execution) can only be solved by making the simulation tolerating the breaching of causality rule. There is essentially only one way of doing this, known broadly as *time warp rollback*, although many variations are possible (discussed later). Time warp requires a “rollback” mechanism which can restore any wrong doings whenever a breach of the causality rule happens. This causes optimistic approaches to have a high resource requirement due to the need of storing and restoring the states. However, allowing a temporary breach in causality enables a LP to execute events following as soon as possible (ASAP) rule, irrespective of the consequences to other LPs. Time warp technique are widely employed by other applications, such as simulation of communication network [25], battlefield scenarios [26], biological phenomena [27], and computer systems [28].

In a primitive time warp rollback system [6], [7], the current times of each LP are unsynchronized from each other and local simulations are carried out concurrently. They process all available events as soon as possible. LPs save the state of their local circuit fragment and the pending event list on a regular basis. These stored data are the *checkpoints* for the simulation, which costs more memory to execute compared with its sequential counterparts [29]. An example is shown in Figure 18.

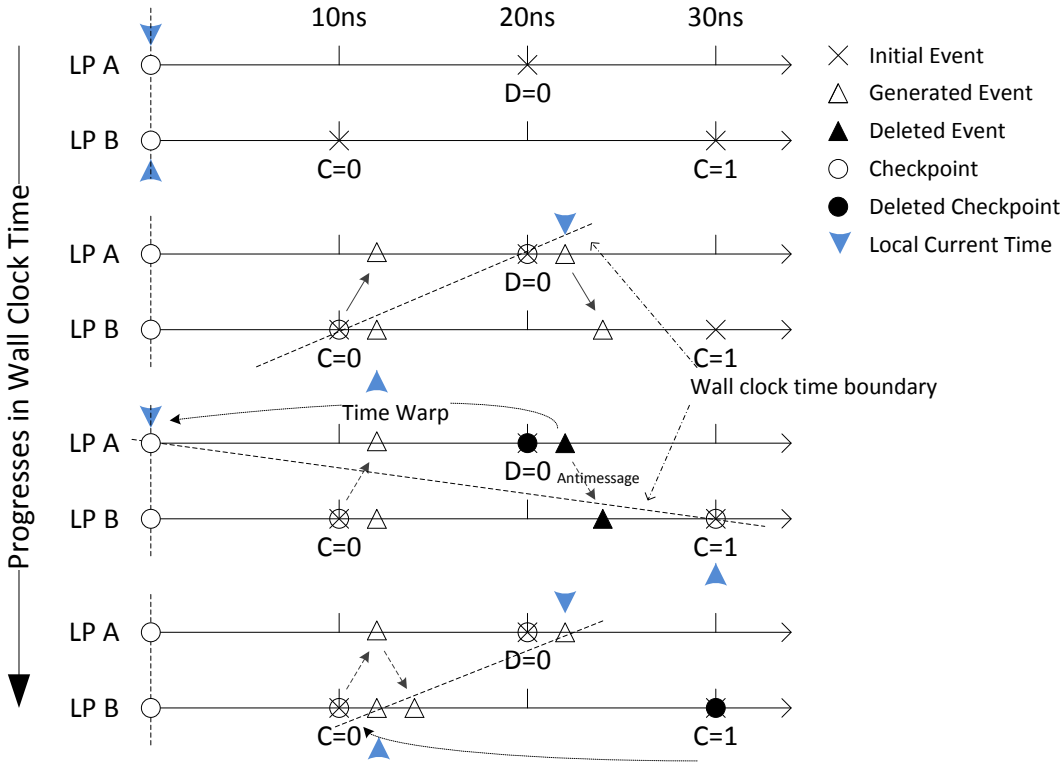



Figure 18 Rollback example

LPs exchange their knowledge of the circuit via *event messages.* Any new arrival of events at a LP will be checked to see whether it carries a virtual future timestamp or not. If it has a future timestamp compared to the current simulation time, the event will be stored in the pending event list and simulation continues. If it carries a *past* timestamp, which is called *straggler event*, the receiving LP will send messages to LPs which are connected to local output signals to cancel all previous sent messages which have timestamp later than the checkpoint timestamp. These messages are called *anti-messages*. Besides sending the anti-message, LPs roll back to the nearest checkpoint before the timestamp of the new arrived event. Then they add the new events to the restored pending event list and re-evaluate the circuit to generate new outputs. The

simulation therefore corrects itself by cancelling the old calculation results, re-evaluating the circuit with new pending events and resending the correct results.

In the previous example (Figure 18) causality is violated. Correct results can be regenerated by rolling back the simulation to a previous checkpoint before the incoming *straggler event*, which carries a timestamp lower than current local virtual time. There are four pairs of timelines in the graph which organized in ascending order of the wall clock time. The *first* pair shows the initial state of the system. There are two checkpoints at virtual time 0 ns and three external events are pending events. All signals in the system hold value "U".

When both LPs start simulation, and process the first events in their pending event lists (*second* pair of timeline), both LPs create new checkpoints to log the system status. New events are also created after the first events are processed. Recall the values of these events in Figure 16, new events in LP A is event "E = 1 @ 22 ns", and event "F = 1@ 12 ns" in LP B. As signal C is connected to an external LP (LP A), the event "C = 0 @ 10 ns" is sent from LP B to LP A. Assume that at the same time, LP A processes the event "E = 1 @ 22 ns" and decides to send out this event to LP B.

However, external event "C = 0 @ 10 ns" arrives at LP A. This is lower than the current time 22 ns in LP A, and LP A initiates the time warp process (*third* pair of timeline). LP A starts sending anti-message to LP B to cancel the event message "E = 1 @ 22ns". The timestamp in this straggler event is smaller than the last checkpoint (20 ns), as a result the simulation needs to rollback further to the checkpoint at 0 ns. The signal values and event list are restored. The checkpoints along the way are removed. The new external event "C = 0 @ 10 ns" is stored in the pending event list, which prepares LP A to proceed simulation with the new information.

Assume the LP B has reached 30 ns by the time the anti-message arrives from LP A (*fourth* pair of timeline), LP B now has a straggler event that tries to remove a processed

event. Time warp is required for LP B to process this straggler event. It finds the nearest checkpoint with a timestamp lower than the one carried by the straggler event, which is the checkpoint at 10 ns. While LP B enters the time warp process, LP A begins reprocessing the events. And the process repeats itself until it reaches the simulation end time specified by the user in the beginning of the simulation.

When an *anti-message* is received by a LP, it will cancel the corresponding positive message either in the past event list or the pending event list. If an anti-message carries a past timestamp, the system will roll back to the nearest past checkpoint before the anti-message timestamp and re-evaluate the circuit again. If the anti-message is a future event, it will simply cancel the event from the future list. However unlike positive events, anti-messages do not correspond to any physical circuit activities. Also, in a distributed system, an anti-message is allowed to arrive at target LP before the *positive* message arrives—a consequence of random communication delays. Therefore, if an anti-message arrives first in the future list and there is no identical positive message in the event list, the anti-message will be queued up in the event list, just like a positive event. When simulation passes through the timestamp of the anti-message, it will directly store it into the past event list without processing it.

Since the global lowest timestamp event will always be processed sooner or later, and the re-evaluation of events after the rollback will always generate the same results before this global lowest timestamp, the process will eventually pick up, and because the simulation does not want rollback any further than necessary, the state saving mechanism comes into solve this problem.

In the direct cancellation technique, the new checkpoint state is created at the timestamp of the straggler event. This means if LP has a saved state at 0 ns, the current time is 100 ns, and the straggler event is at 50 ns, LP will roll back to 0 ns and re-evaluate up to 50 ns. At 50 ns, a new saved state is created at this point. This helps the simulator to reduce the rollback distance when processing the next straggler event.

*Synchronization* is very important aspect of the optimistic approach. The synchronization process in the optimistic approach has two parts, one in the garbage collection stage, the other one in time synchronization. During simulation, every checkpoint has a copy of local circuit and a pending event list of the sub-graph. Potentially, the number of checkpoints will grow as the simulation carries on. Therefore, a garbage collection service is very necessary, as it allows LPs to dump out of date checkpoints to save memory space.

Time synchronization provides a way for all LPs to determine the lowest timestamp across the entire simulation system. This provides a boundary for garbage collection to determine whether checkpoints have expired or not. This time boundary is called *Global Virtual Time* (GVT). Any local virtual time is allowed to rollback to a lower value, but this must not be lower than the GVT. The GVT itself is not allowed to reduce. This is because the GVT indicates the lowest timestamp that exists across the system and, by definition, no event can be generated with a lower timestamp than GVT.

Three methods were proposed to speed up the GVT estimation process. The first method maintains an evenly distributed checkpoint rather than keeping all checkpoints to save memory. [30], [31] The second method keeps track of local event causality pattern and predicts the safe local virtual time for garbage collection. [32] The last method focuses on the memory management of the checkpoint information by prioritising the processing of checkpoints according to their size. [33]

In summary, each LP repeats the execution, communication, garbage collection and rollback detection cycle until the system reaches the end of the simulation. This is the general behaviour of optimistic approach.

There are many modifications that can be built on to the basic time-warp rollback model. In a multi-processor system, message passing is inherently expensive. In any system, the execution of a state save/restore is also expensive, albeit for different reasons. Many variations on the basic theme, playing off these costs against each other, are possible. Pre-eminent amongst the extensions are *lazy cancellation, wolfs call, time windowing, incremental state saving* and *clustering* techniques. The following text explains these in detail.

Firstly, *lazy cancellation* [34], [35] is a technique which reduces the communication density within the system. In the basic time warp model, once the *local* LP is rolled back to a past checkpoint, any previous sent messages by *local* LP *after* the checkpoint timestamp will be cancelled immediately. This is called direct cancellation. In lazy cancellation, a LP will rollback and re-evaluate the circuit to the virtual time prior to rollback, i.e. the current virtual time before any straggler event arrived at local LP. Then, it compares the new output events with the sent output events log. Newly generated events will be sent to output LPs and missing events will be sent as anti-messages to cancel the effects caused by them [36]. Although this reduces the amount of messages sent by LPs, it does not guarantee to make the simulation faster. Any incorrect events sent will have extra time to generate more incorrect results, but it can reduce the number of rollbacks within the system. As a result, lazy cancellation can reduce the number of communication events and rollbacks but the simulation time will generally match the basic time warp model. Further performance analysis can be found in [37].

Secondly, *wolfs call* [38] allows anti-messages to be transferred with priority over other positive messages. Whenever a LP receives an anti-message, it will stop processing positive messages and start processing the anti-message immediately. This will stop the incorrect calculation as quickly as possible. However, it can also cause correct as well as incorrect simulation to stop, which is a drawback of this technique. And other proactive cancellation methods which exploit event causality to prevent cascaded rollbacks were developed. [39]

Thirdly, *time windowing* is a technique which reduces the number of synchronizations and excessively optimistic calculations within the system [40]. In the basic model, LPs simulate the circuit as quickly as possible and there is no limit to how far in the future they can simulate without synchronization. This means that LPs which have a lighter workload or more processing power will spend much of their time processing error prone future work. This is because the correctness of results drops as each simulation moves further away from the last synchronization point [41]. New events close to the synchronization point tend to have a higher correctness, as the events are caused by synchronized correct events. As a simulation gets further away from synchronization point, most of the events generated are based on *local* knowledge.

The *time window technique* [42–44] sets a limit on how far LPs can run away from the synchronization point, and hence reduces system resource waste. In addition to setting a limit on the furthest time a LP can simulate, it is also necessary to set a limit on how often it can synchronize. If too much synchronization occurs in the simulation, this will dominate the runtime; there is a trade-off between memory space and simulation speed. This trade-off can be controlled by changing the time limit between the current time and the last synchronization time. This time limiting technique is called the *time windowing technique*. Many other variations of time windowing technique focuses on the optimization of the size of the window is available in [45–47].

Fourthly, *incremental state saving* is a technique which allows quick rollback to a near virtual past state [48]. Instead of saving the circuit state as a whole, in the basic optimistic model, incremental state saving only saves the *changes* in the circuit state from time to time. It minimizes the number of events to be stored in the memory and it also reduces the large overhead required to copy a large memory space. The drawback of this technique appears when LPs try to rollback a long distance in the past time. It is necessary to undo *all* the events between the target and current time, which can take longer than loading and restoring a single checkpoint directly.

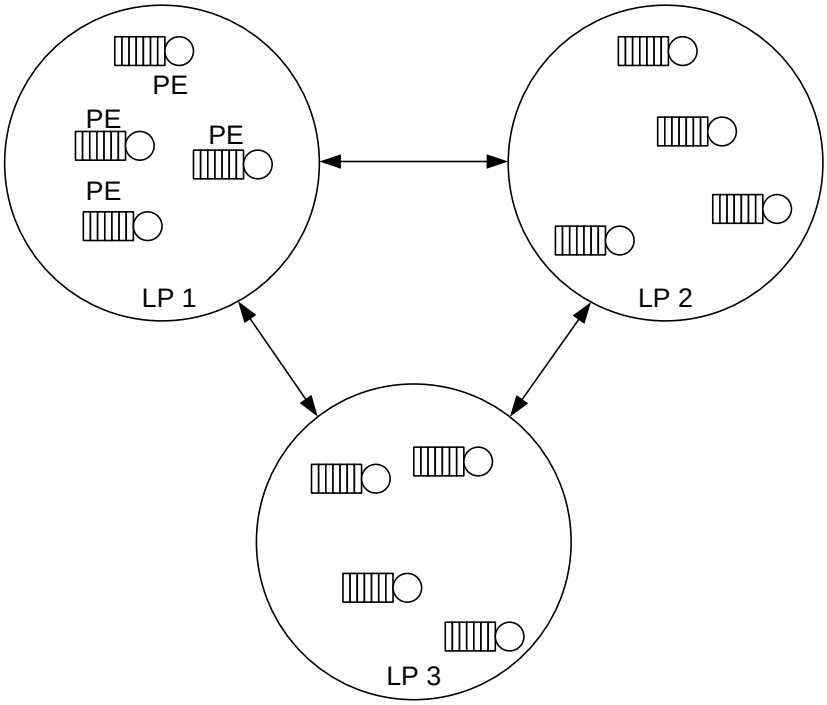



Figure 19 Clustering model

Finally, the *clustering technique* [49–51] splits the partitioned circuit on a LP into smaller sub-circuits and simulates them in several clusters (Figure 19). Each cluster acts as a local processing element (PE). A PE is a complete functional simulation system but the difference is that they only hold a small part of a partitioned circuit, and they share the computation power of a single LP. LPs only need to re-evaluate parts of the local partitioned circuit after a rollback. It saves computation power and avoids large blocks of memory access. The drawback is the overhead required to manage the PEs (which is not required if the LPs do not split the local circuit).

### *Conservative Approaches*

Unlike optimistic approaches, conservative methods solve the causality problem by obeying the rule at all times, rather than tolerating transient breaches of it. They all establish virtual temporal boundaries to limit the maximum time that a LP can simulate; no event will be processed beyond this boundary time. By following this rule, conservative approaches do not require anti-messages. This saves memory and rollback processing time, but reduces the parallelism in simulation. There is clearly a trade-off when choosing the simulation approach. Before making any choices, the various kinds of conservative approaches will be explained in the section below.

There are two dominant categories of conservative approaches; lookahead method and deadlock handling (see Figure 14). The *lookahead* category may be further subdivided into *time windowing* and *pre-computation service time* lookahead methods. The *deadlock handling* category may be further subdivided into *deadlock-avoidance* and *deadlock detection-recovery* methods.

All the conservative approaches employ the concept of virtual temporal boundaries between different LPs. Null messages do not contain state variable switching information. They only carry information about the current timestamp of the sending LP [52], [53]. This null message indicates the lowest external timestamp that the sending LP will generate. Therefore, the receiver can process other valid events safely. Null messages prevent the case where a chained series of LPs all waiting for its predecessors to update their virtual temporal boundaries. A deadlock case is presented in Figure 20, which will be explained later this chapter.

Unlike optimistic approaches, conservative approaches do not have a generic algorithm like the time warp rollback algorithm for optimistic approaches. They apply the null messages in different simulation stages, and solve the boundary problem by actively propagating the timing information. In order to demonstrate the differences, a common circuit configuration (Figure 20) is employed to explain each individual conservative approach.

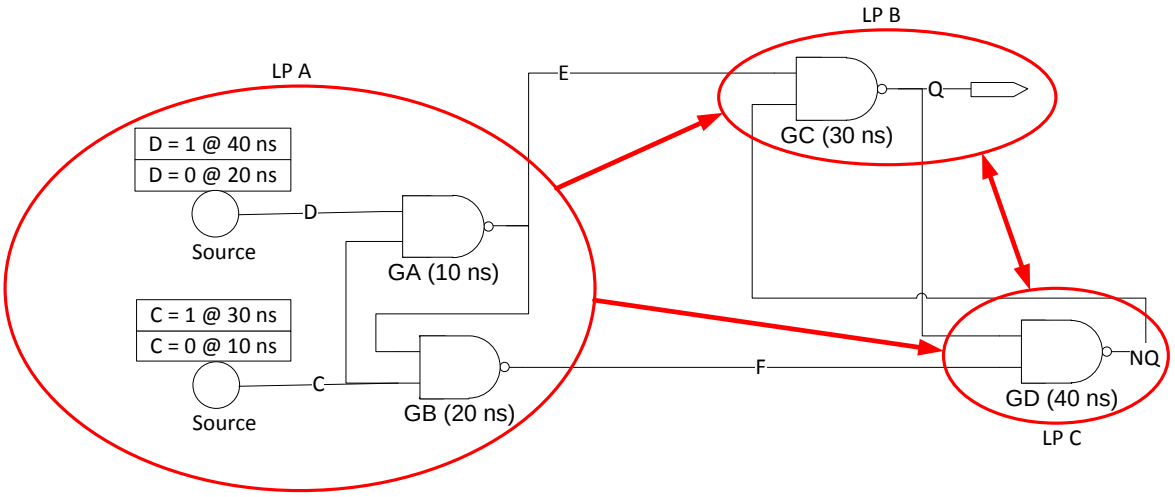



Figure 20 Mapping latch circuit

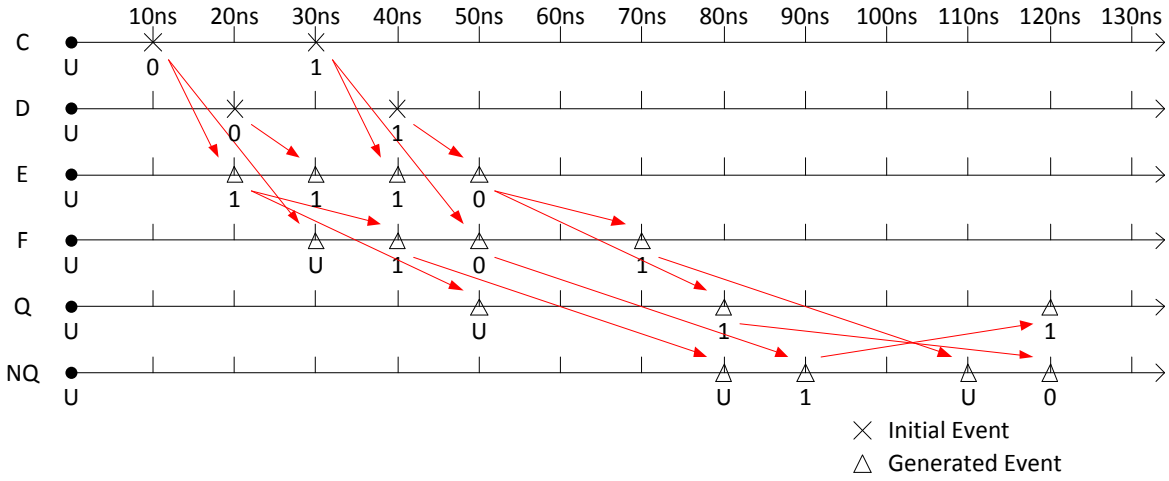



Figure 21 Event sequence of latch circuit

Referring to Figure 20, the example is a latch circuit. Four NAND gates are mapped to three different LPs. The connectivity between LPs is shown in red circuits and lines. There are events on each input wire, D and C. The event sequence in the circuit is presented in Figure 21. The red arrows represent the flow of events between different signals. If the value of a signal changes, this event will trigger a propagation of event for other components, if not, no further propagation is required. The event sequence between LPs, which has a component mapping shown in Figure 20, is shown in Figure 21.

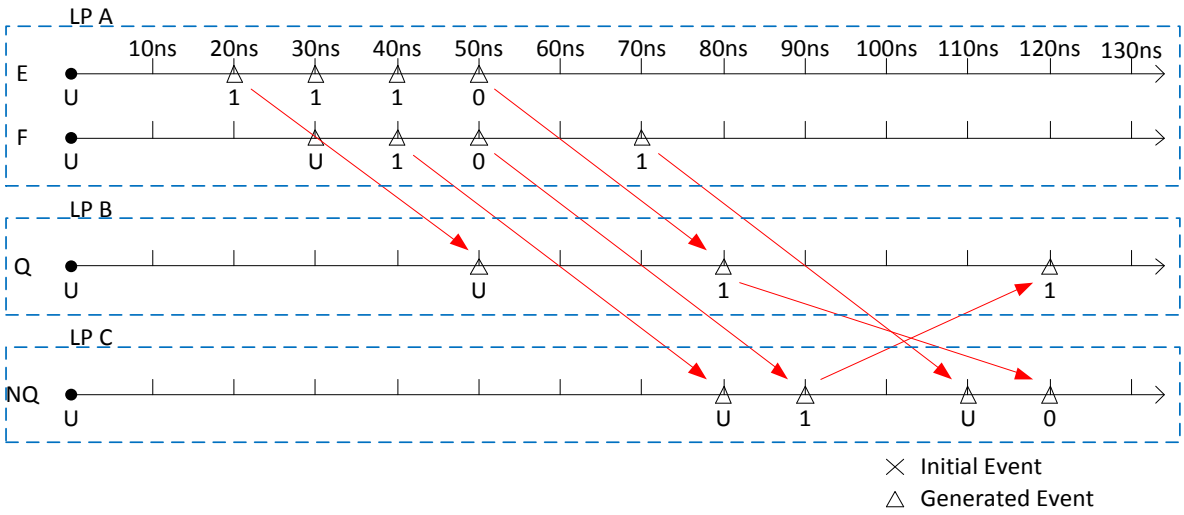



Figure 22 Event sequence between LPs

Only cross LPs events are shown in the Figure 22. Based on the figure, there are three input events to LP B. This figure shows the correct sequence of events during this simulation. In parallel simulation, this circuit forms a deadlock if causality is maintained and no null message is involved. In conservative simulation, each LP must follow a simple rule which is the local lowest timestamp event is always safe to be processed. As LP A holds no external input connection, the lowest timestamp event in LP A is always safe to be processed. However this is not the case for LP B and LP C. Both of these LPs have two input connections. Take LP B as an example, input events are “E = 1 @ 20 ns”, “E = 0 @ 50 ns” and “NQ = 1 @ 90 ns”, as can be seen from Figure 22. The first two events are not eligible for processing, and the third event is not generated in the simple parallel simulation. LP B always compares the timestamp of local events and events from every input connection. If any of the input connection has no incoming event, a default value 0 ns is assigned.

In the case of event “E = 1 @ 20 ns” arrives at LP B, the input connection between LP B and LP C is empty, as no event is sent between the two LPs. While the input event is held in the event queue in LP B, LP B needs a new timestamp from LP C to replace the 0 ns timestamp for the input connection relating to LP C. Without this new update, LP B cannot proceed with the pending events, since by doing so LP B could violate the

causality rule. This forms a deadlock situation. The rest of this section explains how different algorithms solve this deadlock problem respectively.

Lookahead methods set a variable to the minimum delay between the inputs and outputs of a LP. Upon the arrival of external events the lowest next-output-event-time is calculated for each fan-out LP. These fan-out LPs then calculate their new boundary accordingly. The lookahead method actively and constantly updates the safe boundary and the algorithm is free from deadlock. However, some lookahead simulation technique performs a more aggressive processing strategy, and local rollback might be introduced if the speculative simulation is proven incorrect. Local rollback is, as the same implies, a local process only. No messages are sent to other LPs. Although the internal state of the current LP is rolled back, the computational input of this is tiny.

Time bucketing technique [4], [41] targets shared memory parallel computing system. This technique split the simulation into many different sequential segments. Between these segments, the events are being transferred to update any cross LPs activities. The system constantly finds the timestamp of the next cross LP event ($t_{ne}$) in the simulation system and then allows all the LPs to simulate up to timestamp ($t_{ne}$). Regular synchronization is required to determine the time horizon, which is defined by every cross LP events. Figure 23 shows the distribution of time horizons, which are defined by the initiation timestamp at the first cross LP event since the last time horizon.

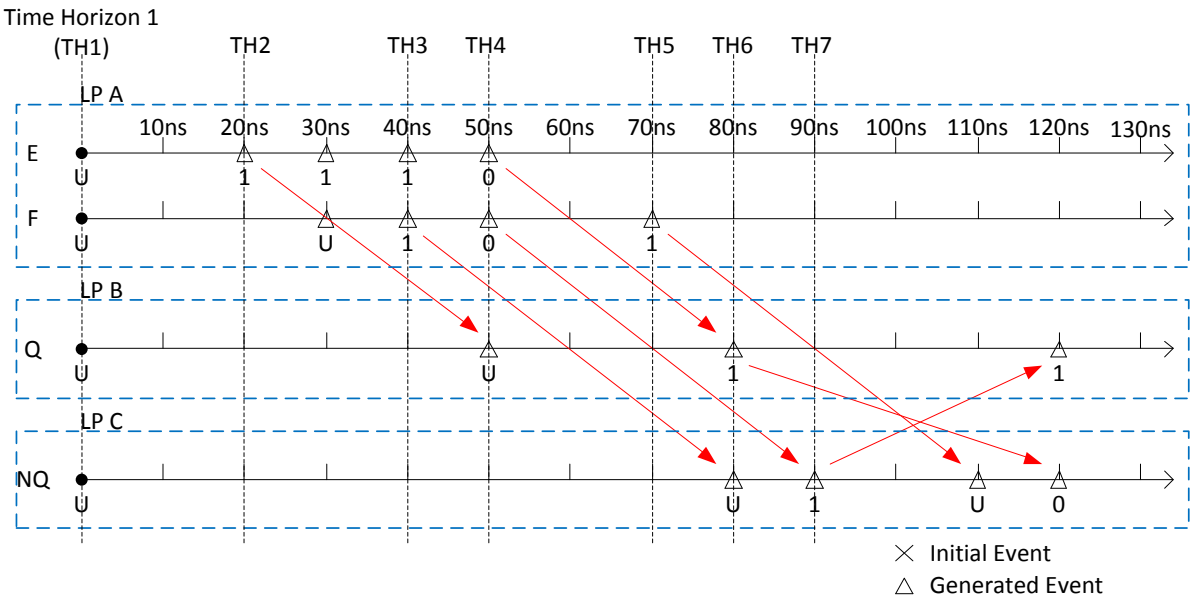



Figure 23 Time horizon distribution in time bucketing technique

Each LP has only one checkpoint which logs the system state at every time horizon. LPs execute simulation up to virtual time ($t_{ne}$) where the first local cross LP event is generated. Referring to Figure 23 at time horizon 1 (TH1), LP A will return $t_{ne}$ of 20 ns, whereas LP B and LP C will return infinity as there is no event in local event list. These $t_{ne}$ values are returned to an overseer process, and the lowest $t_{ne}$ value among all LPs is chosen to be the simulation target time for all LPs, in the case of Figure 23, 20 ns is chosen. The lowest $t_{ne}$ value is then distributed to all LPs. If local $t_{ne}$ in a LP is greater than global $t_{ne}$, the simulation rolls back to the last time horizon, TH1 in this case, proceeds simulation up to the global $t_{ne}$ (20 ns) and wait for the cross LPs events to arrive. This process iterates itself until timestamp of the last time horizon reaches the simulation end time or no event is available for further execution, whichever comes first.

Besides time bucketing technique, the lookahead may be made more efficient [54] by pre-computing the minimum delays between inputs and outputs of each LP circuit fragment. This technique is called *delay pre-computation* [3][55]. These pre-computed delays save computation time, when LPs try to determine a new next-event-time at each fan-out connection. This provides significant benefit for each LP, because processors can actually attempt determine the lower boundary without processing the events themselves (Figure 24). This exploits the parallelism by analysing the simulation

network topology [56]. Ultimately, this estimation of next-event-time only tells the fan-out LPs there will not be any events occurring before the next-event-time. It is not a guarantee that there *will* be an event *at* the next-event-time.

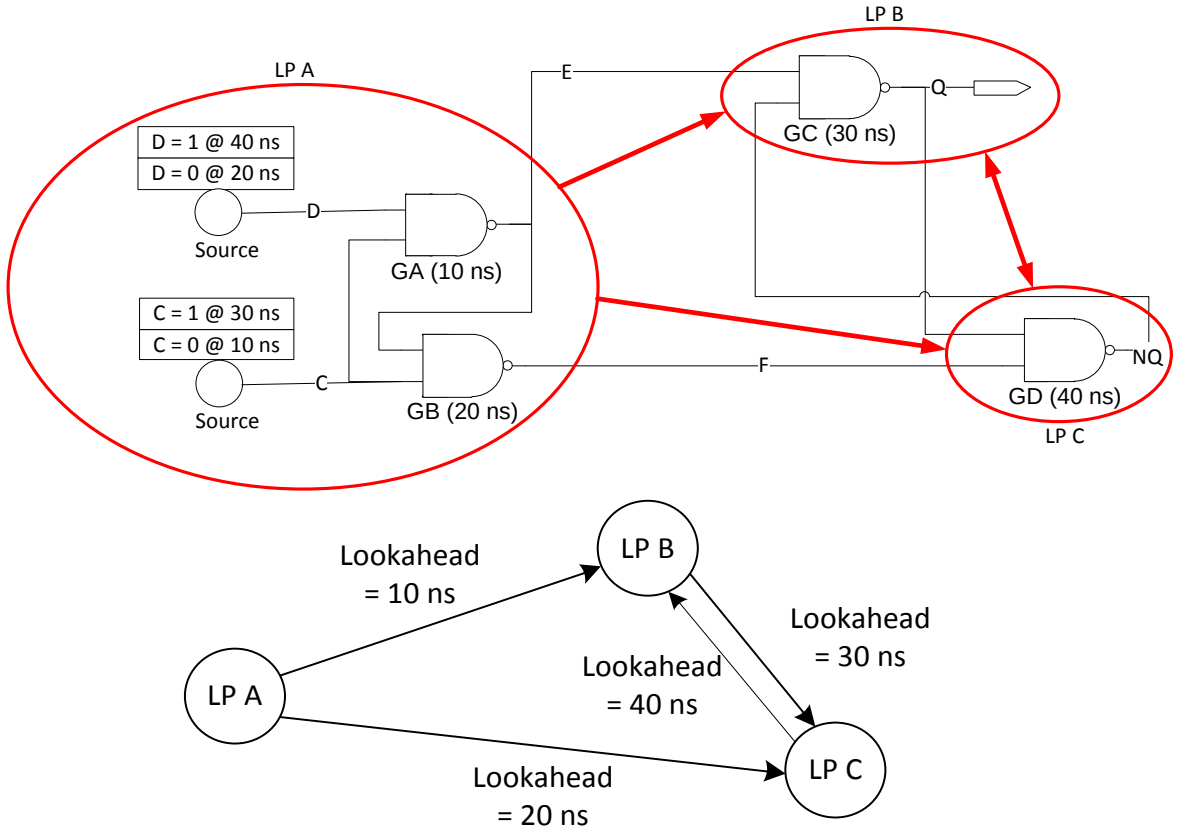



Figure 24 Lookahead value distribution

By abstracting the relationship between LPs from the latch circuit, a graph (Figure 24) can be shown. The lookahead shown on the arcs is the minimum time delay between any external input event and the next-event-time at the source of current arc. As can be seen from the original circuit mapping, the minimum delay between the external input event at LP B and the link to LP C is 30 ns. Take LP A as another example, if events in the sources are treated as external input events, the possible delay between any external inputs and the link to LP C will be 20 ns and 30 ns. The lowest value will be used as lookahead value between two LPs.

The difference between delay pre-computation and time bucketing technique is the approach to generate boundary information. In time bucketing technique, a search for time horizon is required, but unlike delay pre-computation technique, a time horizon

indicates an actual cross LPs event. In the delay pre-computation technique, the boundary is the minimum delay value between the inputs and outputs of a sub-graph. This provides a precise boundary for the process, and avoids rollbacks in the time bucketing technique. This minimum delay guarantees that no event will be available before this time, but it will not guarantee that there will be one after it.

In the deadlock handling category, all LPs are connected via First-In-First-Out (FIFO) connections, and events are passed through these FIFO connections [2][21]. The timestamp of newly inserted events cannot be smaller than any events stored in the FIFO. Therefore, at the receiving end, events coming out of the same FIFO will always be in ascending time order. LPs can compare the lowest timestamps of incoming events and determine the lowest boundary of external events. Any event, which has a smaller timestamp than this boundary, is safe to be processed within the LP. This rule makes sure the causality within the system is maintained. However, it gives rise to a deadlock problem.

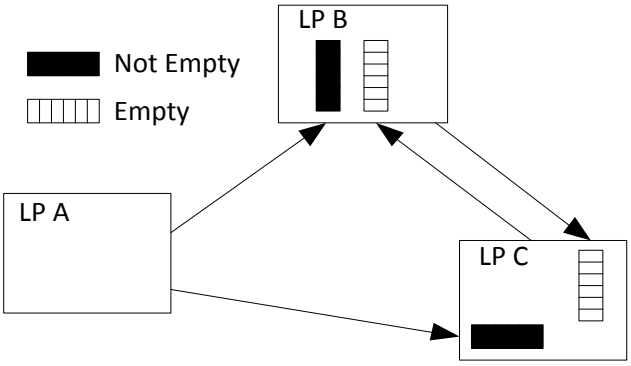



Figure 25 FIFO relationship between LPs

In a deadlocked circuit simulation, a chain of LPs, each waiting for its predecessor, will be formed (Figure 25). This formation will cause the LP chains to effectively freeze during simulation routine as each LP is trying to execute events which have time stamps lower than the earliest possible external event. However, the empty FIFO does not possess any timing information, so the LP is locked until there is another event with a higher timestamp value comes through the empty FIFO which, of course, can never happen.

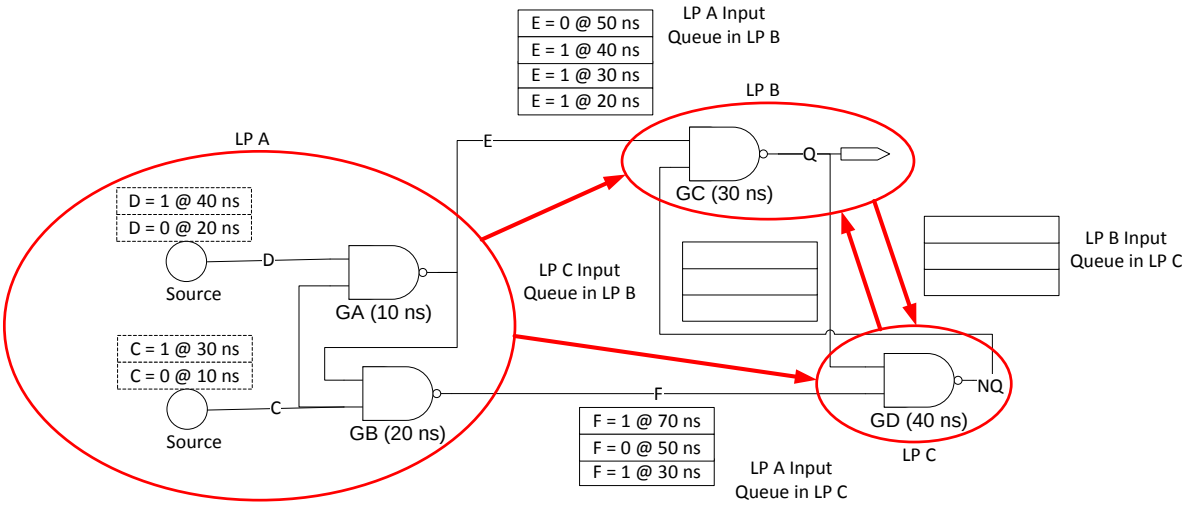



Figure 26 Deadlock case

In the case of Figure 26, the initial events at LP A are processed and generated events are stacked in the input FIFOs in LP B and C. However, a deadlock is formed due to the empty input FIFOs between LP B and C. Neither LP B nor LP C is allowed to process events on the input FIFO that holds events, because of the empty input FIFO lies between the two LPs. And the techniques discussed below will demonstrate how to tackle this deadlock problem.

In general, two ways exist to solve this problem. Firstly, *deadlock detection and recovery* method [58] allows deadlocks to form as the simulation continues. Simulation is divided into two phases, computation phase and deadlock recovery phase. The LPs will simulate until deadlocks are formed. Then LPs can detect any deadlocks either locally or globally [59], [60]. When there is an empty event queue, the deadlock can be detected locally. If the target input queue is full or not ready to receive a new event, the deadlock can be detected via a global operation. A demand-driven null message protocol is engaged. [61] The recovery phase will collect the minimum timestamps for each LP. The LPs which hold the minimum timestamp events will be forced to execute those events and hence break the deadlock. It requires less time to transmit redundant null messages compared to the deadlock avoidance technique.

The optimization of deadlock detection and recovery methods populates around event dependency, state causality, and analyze lookahead across multiple LP. The event dependency approach focuses on optimizing the critical causality path in order to reduce the number of boundaries set on it. [62], [63] A more comprehensive analysis which involves the lookahead, event distribution, physical transfer delay, and processing speed improves the performance and becomes the state causality method. [64] The last method assumes that simulation processes are clustered in physical processes and hence improve the lookahead by grouping up local lookahead information to improve overall lookahead value. [65], [66]

Simulating the circuit shown in Figure 26 using deadlock detection and recovery method will reach the same state as it is shown. And an overseer process can detect that the simulation stops as no events are being processed, the simulation target time has not reached and there are messages around the system. The overseer process can request for the lowest event timestamp in each LP. The LP with the lowest timestamp value among the returned is allowed to proceed with the event it holds. In this case, the event "E = 1 @ 20 ns" is processed in LP B, and an event "Q = U @ 50 ns" is generated in LP B and sent to LP C. Following this input event, the events that hold timestamp lower than 50 ns is allowed to proceed, because there will be no event comes from LP B that might carry a timestamp lower than 50 ns. The simulation can carry on as normal after these. If a further deadlock is formed, the recovery process is re-initiated to solve it.

Secondly, the *deadlock avoidance* method [2], [67] - as its name implies - does not allow deadlock to be formed during a simulation. LPs update the simulation boundary with their immediate successor LPs as soon as there is a change in a local status. This simulation boundary is derived from the input boundary or current simulation time, whichever is lower, plus the shortest minimum delay of local components. In the initial state of the simulation, each LP is initialized with the local current virtual time, a vector of boundaries, and a local boundary time which is derived from the vector of boundaries (see example in Figure 27, further explanation available follows the figure). A boundary in the vector consists of two parts, the predecessor ID and the boundary time. A boundary indicates the *lowest* timestamp that will be attached to a generated event

produced by a predecessor LP. Each predecessor that feeds signal to a LP *n* will leave boundary information in the LP *n*. Unlike the pre-computational technique (see page 38), deadlock avoidance technique sends null messages according to *minimal incremental step* within the circuit rather than working out the propagation delay values between I/Os. In order to maintain causality, LPs are not allowed to process any event that carries a higher timestamp than the local boundary. The local boundary derives from the lowest boundary in the vector of boundaries. However, if there is no predecessor LP for LP *n*, the vector will be empty and the local boundary is represented as infinity.

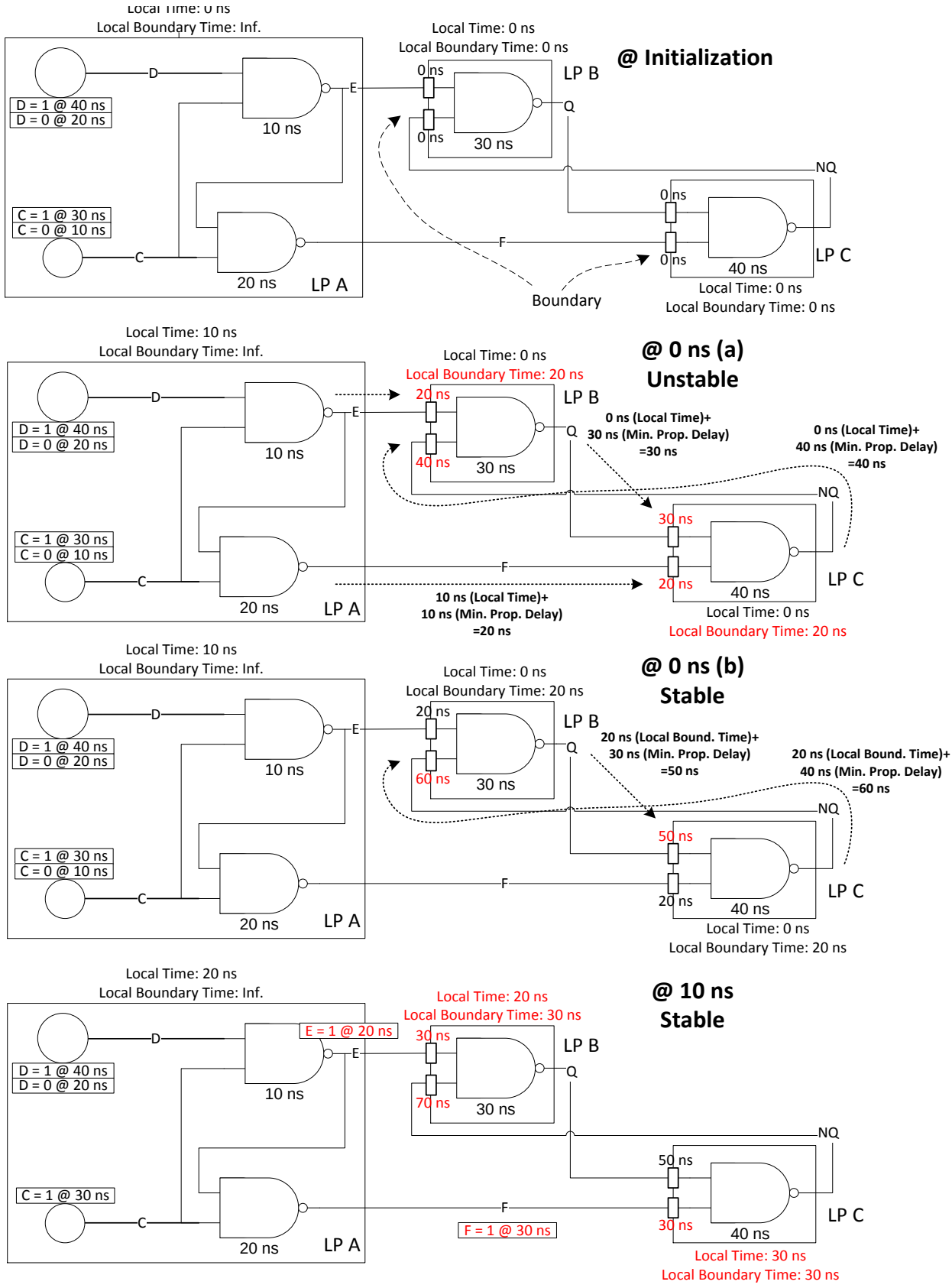



Figure 27 Deadlock avoidance example

The example shown in Figure 27 demonstrates snapshots of system states when simulating from initialization to 10 ns using deadlock avoidance technique. (The global virtual time shown on the top right hand side of each figure is not a real parameter in simulation. It is for ease of understanding only.) The circuit is the same circuit used in previous examples, with the simulation parameters appended to the graph. These parameters include local current time, local boundary time, and timestamp value attached to each boundary. There are four gates and two sources shown in Figure 27. They are mapped to three different LPs manually.

At the initialization stage, all the boundaries and current time are assigned with 0 ns, except the local boundary at LP A which is assigned with infinity. This is because there is no external input connection to LP A, therefore an event with the lowest timestamp is always safe to be processed by LP A.

Immediately after the start of the simulation, all LPs establish local current simulation time $T_{local}$ by defining it with the lowest timestamp among the events, or zero if no event exists. The lowest boundary value $T_{bound}$ among the inputs is also defined. LPs then find the lowest propagation delay $L_{delay}$ among the local components. Following these three steps, a null message time can be computed. The null message time is the lower value of $T_{local}+L_{delay}$ and $T_{bound}+L_{delay}$. Any update to both values will trigger a new null message to be generated if the new value is greater than the previous value. For example, the local time at LP A is 10 ns because of the event at signal C, and the minimum propagation delay among the components is 10 ns. Null message carrying 20 ns value can be delivered to LP B and LP C, the fan-out LPs of LP A. LP B and C carry out the same process concurrently and null messages carrying 30 ns and 40 ns are sent out to each other respectively (figure (a) @ 0 ns). Both local boundary time and local time are 0 ns, therefore the longest propagation delay within the LP will be the null message value.

When LP B and LP C receive the null messages sent by other LPs, new local boundary can be established which are 20 ns for both LPs. New null message timestamps are

available for both LP B and C after the boundary update, and hence new null messages carrying 50 ns and 60 ns are sent by them correspondingly. As these new null messages do not change the local boundary time or local current time at LP B and C, the system reached stable state at 0 ns (figure (b) @ 0 ns). The establishments of local boundary indicate the safe boundary for local LP. In this case, both LP B and C are able to process any event that holds a timestamp lower or equal to 20 ns without violating the causality rule.

At 10 ns, the event "C = 0 @ 10 ns" is safe to proceed, LP A generates events "E = 1 @ 20 ns" and "F = 1 @ 30 ns". They are sent to LP B and C respectively. After processed the first event, LP A sees no event exist in 10 ns. It updates the local current time to the next unprocessed event, which has a timestamp of 20 ns. New null message timestamp of 30 ns can be derived consequent to the update of local current time. As the new timestamp is greater than the previous null value (20 ns), null messages are sent out to LP B and C. A further propagation of this boundary update is triggered and reaches a stable state shown in the last graph in Figure 27.

In order to speed up the propagation of the boundary information, non-blocking null message was investigated in reference [68]. An advanced event scheduling looking for large blocks of ready to execute events were also investigated to increase lookahead [69]. A process migration mechanism based load balancing technique was also introduced by [70]. Other null message reduction techniques were investigated via mathematical model [71] or lookahead value optimization[72], [73]. A compiler based lookahead extraction method was presented in [74], [75]. The performance comparison studies for preceding algorithms were carried out in [76–79].

One of the key features of the deadlock avoidance technique is the absence of global control. Causality is maintained by establishing local boundaries and updating them with null messages. The ensemble of LPs does not require a central control to complete a simulation. As the related information propagating through the LPs forms a *local* minimal time, this mechanism replaces the function of any global control. In addition,

the conservative nature of deadlock avoidance does not allow LPs to perform optimistic event execution, which saves the memory cost related to optimistic event processing and the extra state saving required in optimistic approaches.

This section has provided an overview of the two principal techniques that are termed *conservative.* There also exist other techniques [55], [80–84] based on the techniques introduced in this section. In general, the conservative approach is a relatively slow but low memory cost method for implementing a PDES system.

### *Summary*

All of the approaches above are able to solve the PDES problem, but their usage of system resources and control overhead differs. In order to have a better idea of all the pros and cons of these approaches, a basic comparison between the messages density and the appropriate memory cost, is summarized in Table 3.

| | **Quantity of Null Message** | **Rollback Involved** | **Memory Consumption** | **Synchronization Requirement** |
|---|---|---|---|---|
| Deadlock Avoidance | High | No | Low | No |
| Deadlock Detection & Recovery | Medium | No | Medium | Periodic |
| Delay Pre-computation | High | No | Low | No |
| Time Bucketing Technique | No | No | Low | Regular |
| Moving Time Windowing | High | Yes | Medium | Periodic |
| Time Warp | No | Yes | High | Periodic |

Table 3 Comparison table of different techniques

Based on the SpiNNaker structure, a high memory consumption technique is the least likely to be chosen as the candidate technique, due to the high cost of memory access time for a SpiNNaker processor and limited memory space available. Furthermore, a higher communication cost would favour the SpiNNaker system due to the low communication cost asserted by the SpiNNaker specification, in comparison to the conventional cluster computers. Finally, global synchronization during simulation levies a heavy cost on the overall performance for a SpiNNaker system, as SpiNNaker is a fully distributed system. A synchronization operation can leave many individual

SpiNNaker nodes idle in order for it to be carried out. As a result, we are left with two possible candidate techniques: deadlock avoidance and delay pre-computation. Since the delay pre-computation technique is based on deadlock avoidance, it makes sense to start modelling from the deadlock avoidance technique.

Two threads can be identified in the evolution of PDES strategies. One of them is heading for more abstraction which lead to the development of High Level Architecture (HLA) [85], [86]. Instead of focusing on the underlying techniques, this develops a standardized simulation infrastructure which is capable of user-friendly general purpose parallel simulation. The second trend follows the path of optimistic simulation, which is explored in [87], [88]. They have shown that the multilevel time warp system does improve the performance over the traditional time warp system. However, the performance quickly reduces when the simulation goes towards higher levels of parallelization [89].

From the two arguments above, there is a possible gap in the spectrum of parallel simulation techniques, which might be filled by enhancing the conservative simulation technique. Although the conservative simulation technique cannot outperform its optimistic opponent in a highly distributed system, the result is unclear when dynamic load balancing function kicks in. The nature SpiNNaker seems a sensible candidate platform for this simulation technique to be carried out in a large scale distributed system. In summary, the deadlock avoidance technique is chosen to be the simulation technique which emulates the deterministic simulation capability of the SpiNNaker platform. In addition, the impact of dynamic load balancing on conservative simulation can also be seen in the final results.

## 2.3 *Partitioning Algorithms*

Network partitioning is the process of splitting a network model into several parts, while maintaining the overall functionality, prior to mapping the parts onto the simulation hardware. In single process discrete event simulation, the network inside the simulated model is a complete, fully elaborated network. The simulator process has full access to all elements inside this network. However in PDES, the network in the

simulated model is split into a number of parts and each part is located on a different processor. These processors only have full access to their own part of the circuit. This enables the processors to simulate the model concurrently, which increases the computation throughput of the simulation system. By introducing the partitioning algorithms, it enables the PDES to simulate circuits size ranges from a single component to the entire circuit with efficient connection in between different LPs.

Network partitioning plays a key role in the distribution of the *workload* and *connectivity* between the processors. Workload is the amount of computation demand located in individual partition. Assuming the processing speed in each partition is constant, then the higher the workload, the longer the overall simulation time. If the workload is not properly partitioned, the heavily loaded processors will slow the overall simulation speed. Likewise, if partitioning is performed in an inefficient way, a lot of connections are left between the processors: extra communication load is introduced and hence the performance is again degraded. Therefore, partitioning is a very important part of PDES. The circuit network partitioning problem represents a circuit as a directed graph. Figure 28 shows how a circuit may be represented as a directed graph. In this figure, signals C, D, Q and NQ are assumed to be inputs and outputs - each of them is represented using a node in the final graph. The four gates are also represented as nodes in the directed graph. The wires in the original circuit are translated into the arcs that connect different nodes.

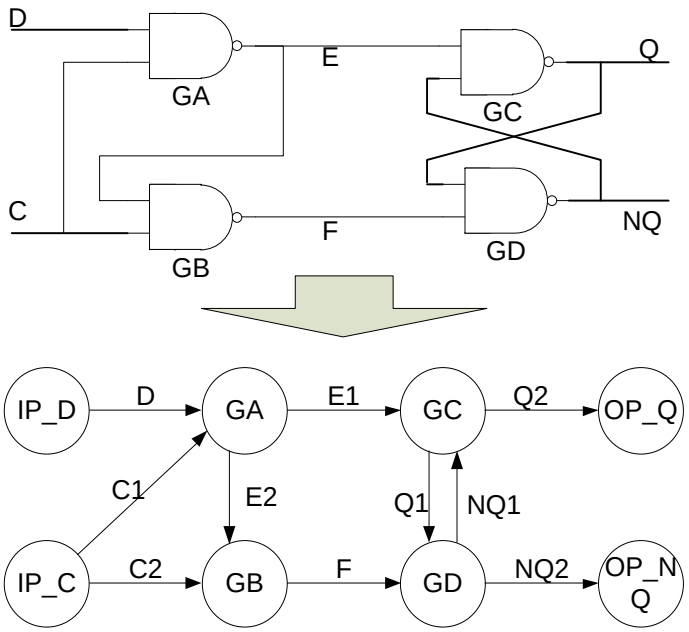



Figure 28 Representing a circuit as a directed graph

The inputs of a logic gate may or may not be interchangeable. In SDES, there are two types of circuit graph. One is a direct translation of structural VHDL which is called VHDL Translated Graph (VTG) in this thesis. The other is a further translation of the VTG using only basic logic gates which is called the Combinational Logic Graph (CLG). In a VTG, the components may be complex circuits on their own. As a result, when constructing a VTG circuit, there is a strict rule about the connection order of its inputs. This is because the VTG relies on the order to distinguish between different inputs. In contrast, CLG only consists of basic logic gates of which inputs are interchangeable. For instance, the multiplexer in VTG graph has two types of inputs, selection and data inputs. These inputs cannot be swapped. However, at the CLG level, components are combinational gates only. Their inputs can be switched without affecting the simulation result.

The aim of network partitioning is to split a network model into many parts without changing the functionality of the overall circuit. The number of wire cuts and the workload distribution between the processes represents the quality of partitioning. However, the effect on workload distribution can only be evaluated at the time of execution. Partitioning approaches can be divided into two broad classes: static and dynamic partitioning.

### 2.3.1 Static Partitioning

Static partitioning is carried out *before* the simulation begins. The aim is to reduce the number of connection cuts, i.e. the number of wires that pass between the partitions of the circuit. Static partitioning may be further sub-divided into iterative moving, annealing, and multi-level/clustering partition techniques.

The *canonical iterative algorithm* to solve the partitioning problem was developed originally by Kernighan and Lin [90]. It was then improved for bi-partitioning by Fiduccia and Mattheyses [91]. The concept was then expanded to enable multi-way partitioning by Sanchis [92], [93]. The idea of an iterative move algorithm is to find potential component movements between partitions, which are capable of reducing the number of wire cuts. Firstly, the algorithm scans the graph and tries to move each component from one partition to another, and sees if there is any wire cut reduction. Secondly, it stores wire cut reduction values in a vector, ordered on which components can produce the largest number of wire cut reduction. This wire cut reduction value can be negative, which means the movement of this single component will *increase* the number of wire cuts. Finally, the components with the most wire cut reduction value are moved, and the costs of all other nodes updated. The process is repeated until zero gain or no lower gain can be obtained from the movement of any components.

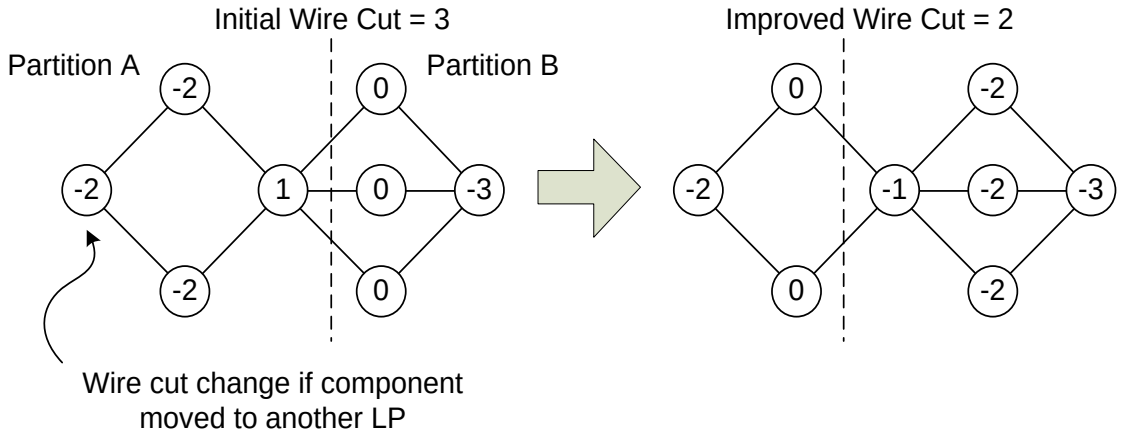



Figure 29 Iterative move example

In Figure 29, each node is labelled with the change in wire cut value that would be caused if it were to be moved to the other partition. The node with a gain of 1 in partition A is the node with largest gain available in the graph. Other components have gains less than or equal to zero. Therefore the iterative move algorithm will transfer this node from partition A to B. There is also a balancing problem in the iterative move algorithm. If most of the components in a circuit are assigned to a particular partition, components in other circuit tend to be moved to the overcrowded partition with a high wire cut reduction value. This is prevented by the upper and lower component number *boundary*. If a partition device count exceeds the upper limit of the component count, components in other partitions are banned from joining this overcrowded partition. The same principle applies to the lower limit, when the component count in a partition is below the limit, components within this partition will not be moved to other partitions. Further performance evaluation of this method can be found in [94].

In the *simulated annealing algorithm* [95], the circuit components are shifted in a way similar to that employed by the iterative move algorithm. However, the decision to move is controlled by a random number generator, so the possibility always exists that an individual transaction will actually worsen the overall quality of solution. The concept of *temperature* in the annealing algorithm represents the possibility of acceptance of a negative gain movement. If the temperature in annealing algorithm is high, a negative gain movement has a higher possibility of being accepted by the partitioner. The annealing algorithm starts at a high temperature, which allows more negative gain components to be moved. Then the temperature is gradually decreased, allowing fewer and fewer negative gain movements to happen. Finally, the partitions *freeze* by only allowing the positive gain movements. This is a good approach to partitioning, because it can avoid local minimum wire cuts by allowing some of the negative gain movements to occur. However, it does not guarantee generation of consistent results on repetition. The partitioning in PDES is carried out concurrently, and the random number generator is likely to be different between processors. This is a disaster from communication point of view, because local partition is not able to establish a fixed link to neighbouring process due to the unpredictable partitioning layout in the neighbour partitioning layout.

Unlike other partitioning techniques, the *clustering partitioning* technique groups components into non-overlapping sub-graphs. These sub-graphs are grouped to form a partition in a circuit with area or connectivity restrictions. In [96], circuits are analyzed and grouped according to *connectivity enclosure* within circuits. For example, a node that has multiple outputs will form a *fan-out node*, and a node with multiple inputs will form a *re-convergent node*. Nodes lying on the paths between these nodes are part of the circuit portion which is called *petals*. These *petals* are the basic elements of partitioning in the clustering partitioning technique. They can be grouped to form a partition according to the area size or connectivity restrictions defined by user.

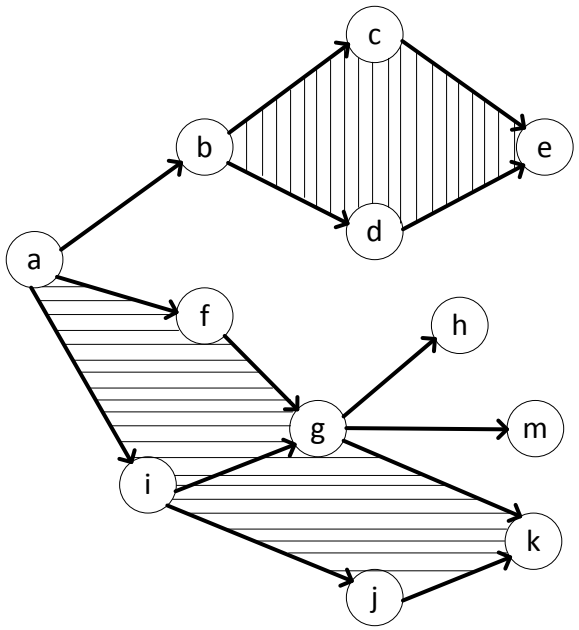



Figure 30 Cluster partitioning example

In Figure 30, the fan-out nodes are a, b, i, and g. Convergence nodes are e, h, m, and k. If a fan-out node has disjoint paths to a convergence node, then this forms a basic petal during partitioning. The example shown above has two petals, which are b → e and a → k.

Similarly in [97], nodes are grouped according to the density of connections between them. The nodes which are closely connected to each other will form a sub-graph. The connectivity between sub-graphs is further analyzed, and the tightly connected sub-

graphs will form a partition. This process repeats until the desired number of partitions is created. Experiments presented in [35][36] show that this multilevel algorithm can produce substantially better results than non-multilevel schemes.

Take the same example used in Figure 31, the partitioning technique requires a scan of the network which creates a list of nodes sorting by the number of connections connected. Based on the figure, node g has the highest number of connections. All adjacent nodes of node g are grouped to form a single petal, which includes f, i, h, m and k. These components are flagged up, along with node g, to be excluded from further partition. The connection counts of the petal adjacent components are updated to exclude the connections to the newly created petal. The node next in line with highest number of connections is node b. As node i is already included in a petal, and the connection number of node a is updated to only 1 due the exclusion of connections to petal. A new petal is hence created based on node b. The new petal includes node a, b, c and d. Node e and j are created as individual petals which can be later merged with other petals. All petals in the graph are shown in Figure 31.

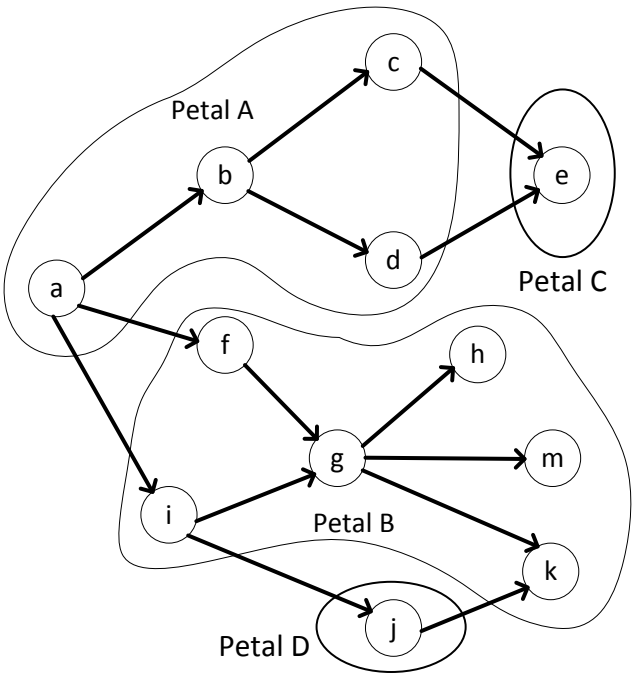



Figure 31 Cluster partitioning example 2

A similar type of partitioning algorithm, called *functional partitioning*, splits the circuit according to the function of the sub-circuits. This can greatly reduce the communication cost at run time, as these sub-circuits tend to have closer connections and hence improve overall performance. However, this partitioning technique requires the partitioner to have prior knowledge about the structure of the target circuit and the boundaries between different implemented functions. (language based [100–102], problem based [103])

Employing a good static partitioning technique can improve the overall performance, due to a more evenly distributed workload and fewer communication links among all LPs. In this project, the improved version of canonical iterative algorithm, the multi-way partitioning technique is employed. The simple concept, performance and consistency of this technique make it suitable for PDES. The partitioning performance of this technique is investigated later in chapter 4.

As outlined above, there are four main types of static partition algorithm – canonical iterative, simulated annealing, clustering and functional partitioning. The major limitations and differences between the four different types of partitioning techniques are shown in Table 4.

| | Require Structure Knowledge | Capable of Multiple Partitioning | Repeatable |
|---|---|---|---|
| Kernighan–Lin (K-L) | No | Partial | Yes |
| Fiduccia-Mattheyses | No | No | Yes |
| Multi-way | No | Yes | Yes |
| Simulated Annealing | No | Yes | No |
| Clustering | No | Yes | Yes |
| Functional Partition | Yes | Yes | Yes |

Table 4 Static Partitioning Algorithm Comparison Table

The aim of static partitioning is to cut the circuit into a number of subcircuits, minimising the inter-subcircuit wire count. The key is to find the maximum reduction of

wire cuts attainable by moving components between sub-circuits. As the multi-way partitioning is based on K-L algorithm, the K-L algorithm will be explained first.

The K-L algorithm deals with only two partitions: A and B. *Firstly*, the algorithm starts with a perfectly balanced component distribution, i.e. the number of components in the two partitions is equal. The initial partitioning is carried out arbitrarily, but it requires the components counts to be nearly identical. *Secondly*, the algorithm tries to find a set of components in a partition which can be moved to the other partition. The components are then labelled to be excluded from subsequent partitioning movements. As a result, the reduction of wire cuts for moving each of unlabelled components is calculated:

$$G_{moving} = C_{internal} - C_{external}$$

The $C_{internal}$ is the number of wires cut when the component stays at the current partition. $C_{external}$ is the number of wires cut when the component is moved to the other partition. $G_{moving}$ is the wire cut difference before and after the component movement. For example, if a component is assigned to partition A, the gain obtained by moving this component to partition B is represented as

$$G_{moving} = C_A - C_B$$

Applying this to all the components creates a vector of *wire cut reduction data*. These data can be sorted in descending order in two stacks for easy access, where each partition has its own stack of local components. *Thirdly*, the partitions of the components on top of both stacks will be switched and labelled to prevent further movement in current iteration. The gain information connected to these two components will be updated and sorted in the stacks. *Fourthly*, repeat the switching activity until the overall gain by switching the two components will result zero reduction in wire cut. This completes one iteration of moving. The partitioning process will reset all labels attached to components, and start the entire process again until the label reset does not produce any reduction in wire cut. The pseudo code for this process is written below:

```
Function KL Partition
   split nodes into two balanced partitions A and B
   find G_moving for all components
   sort the components according the G_moving value in descending order
   do {
      while ((G_max in A > 0 + G_max in B) > 0) {
         select the component C_A, C_B with G_max in A and B
         switch the partition for C_A and C_B
         update the G_moving for neighbouring components
         remove C_A and C_B from further partition switching in this iteration
         sort the components according to the G_moving value in descending order
      }
      reset the G_moving value associated with all components
      find G_moving value for all components
      sort the components according to the G_moving value in descending order
   } while (G_max > 0)
End function
```

Figure 32 Pseudo-code for the KL algorithm

The progress of the algorithm is shown in Figure 33. The graph on the left shows the wire cut reduction in a single iteration, and the one on the right shows the gain obtained through the iterative approach. The bold line indicates a component move has been performed. The dotted line shows the changes in overall wire cut if further improvement steps were carried out when the highest possible wire cut reduction is less or equal to zero. The process terminates when no improvements can be made from the initial partition.

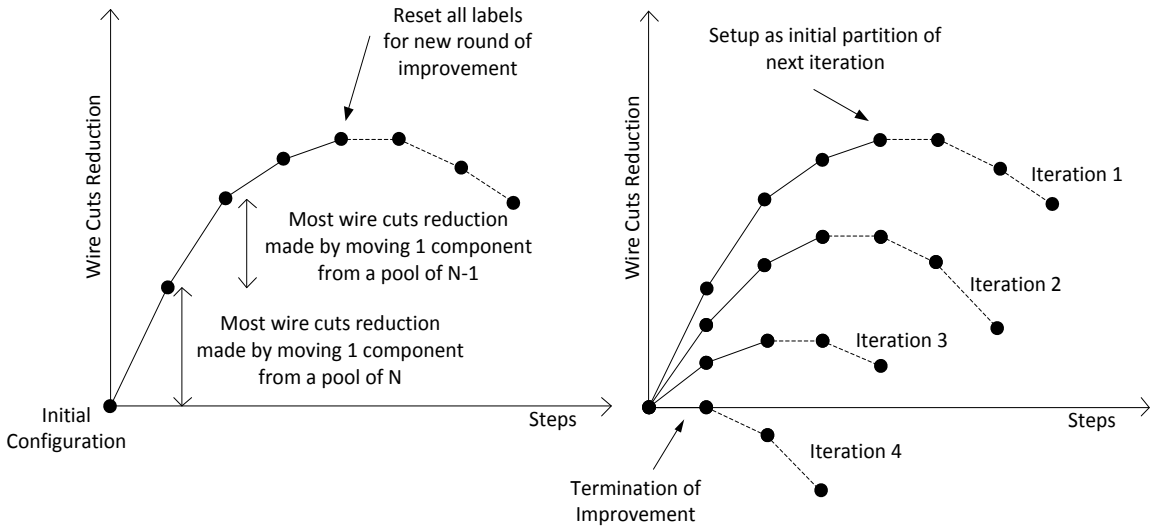



Figure 33 Wire cut reduction process

The multi-way partitioning method establishes a database of wire cut gains for each single component within the simulation object, in this case the gates within the circuit. The difference between multi-way partitioning method and K-L algorithms is the

tolerance of imbalance component count between different partitions. In the K-L algorithm, the movement of components must be paired up in order to balance the size of both circuits, whereas the multi-way partitioning allows the system to move to a target partition without its counterpart moving to the source partition. As explained earlier in this section, there is an upper and lower limit on the difference between the local and the average component count when the components are evenly distributed. The implementation and analysis of the multi-way partitioning technique are shown in section 3.2 .

### 2.3.2 Dynamic Partitioning

The aim of *dynamic* partitioning is to change the partitioning *during* a simulation in response to a fluctuating workload. This is known as dynamic load balancing (DLB). As simple wire cut reduction may be done at the static partitioning stage, DLB can pass part of the calculation workload from one LP to another which smoothed the workload for each LP. This enables LPs to finish at roughly the same time and hence increases the overall system performance.[104] There are two components that play the key roles in DLB. One is knowledge of the current workload, which enables DLB to balance the workload accurately, and the other is the control overhead required to complete the balancing. Different DLB approaches adjust the trade-off between these two factors.[105–109]

There are two types of DLB approach, *diffusional* and *hierarchical* schemes. Diffusion schemes pass the workload from one LP to its neighbouring LPs [49], [110], [111] or vice versa. LPs have upper and lower workload thresholds. Once the workload in a LP is over the upper threshold, it can pass workload to other LPs. Ideally, the cost of passing jobs to neighbours is less than the amount of extra work required to process the job. In other words, the transfer of a job only happens when the estimated gain is more than the cost of rebalancing. Because DLB will effectively stop the simulation during balancing, this by itself reduces the simulation speed. The pause is caused by the redistribution of components between different partitions. This process not only requires the movement of components, but also the events that associated with these components. Without the pause in simulation, some of the events may be processed without correctly

routing to other LPs, as the component information hasn't been updated at the time of event processing.

There are two parameters that define the workload threshold. The first is the minimum workload that is eligible for transfer. The second is the level of difference in workload that is tolerated. The first parameter filters out immature workloads to be transferred, which means the benefit of transferring workload does not outweigh the cost of DLB. For instance, if LP A processed 100 events and LP B only processed 10 events, although the level of difference is high at 90%, the absolute amount of event workload difference is negligible, since the event processing speed is around 100K events/sec. The additional time for processing the extra 90 events is less than 1ms. The second parameter defines the maximum difference in workload that the simulation can tolerate. As the simulation progresses, the workload imbalances will shift from one place to another and it is not feasible to force the workloads in different partition to be exactly the same. If this parameter is set to zero, the DLB will occupy most of the simulation time - this is not practical.

In a *hierarchical* scheme by contrast, load information is centralized and analyzed by a central controller. The workload may be balanced across the entire system. The hierarchical scheme directly transfers the most heavily loaded partition to the most lightly loaded partition regardless the geometric connection between the two. However, the control overhead required by hierarchical scheme is greater than the diffusion scheme. Thus, the rebalancing threshold in hierarchical schemes is typically greater than the threshold in diffusion scheme.

The basic idea of SpiNNaker is to perform scalable distributed simulation, which implies that it does not have a master processing unit that controls the overall simulation. For this reason, the diffusional approach is employed in this project.

Diffusional DLB may be further sub-divided: sender initiated diffusion (SID) [112][113], receiver initiated diffusion (RID) and the gradient model [114]. The gradient model requires a regional overseer in order to determine the workload distribution. Due to the distributed nature of the SpiNNaker system, only local information is available, so only SID and RID are suitable. The performance difference between SID and RID is around 10% [105]. When a process enters partitioning mode, the predecessors of this process need to hold their partition stable, otherwise the connectivity between the processes may be lost in the dynamic partitioning.

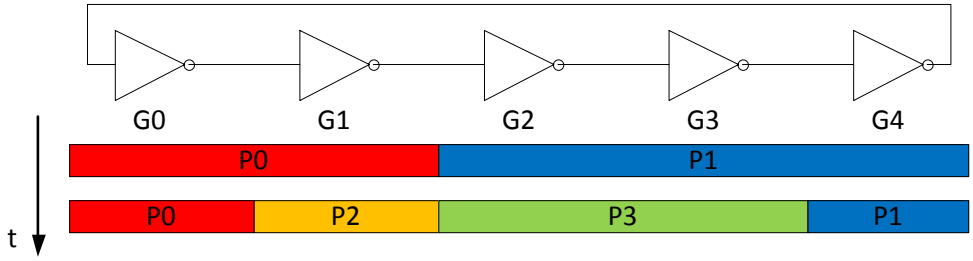



Figure 34 Locking mechanism example

Take the circuit in Figure 34 as an example. The five gates are partitioned into two parts. Assume both P0 and P1 are allowed to carry on with partitioning and G1, G2 and G3 are the gates that need to be transferred to new partition P2 and P3. If both operations are carried out concurrently, P3 will establish a link with P0, as P1 still has the old information that G1 is situated in P0. By the time P1 receives the updated partitioning information, it will only update local information without propagating further to P3. This is because the component transfer process has already ended. And same applies to P2, which will establish a link with P1 instead of P3.

To avoid this scenario, a locking mechanism is employed. The locking mechanism disables the component transfer in all predecessors of a component sender to avoid any link mismatch during the balancing act. Locking of successor processes is unnecessary, as successor is unable to initiate a component transfer if its predecessor is already locked or in component transfer mode. If this locking mechanism returns fail in one of the predecessors, the dynamic partitioning request will be dropped. Applying RID technique lengthens the locking chain, which in turn increases the control overhead and

reduces the success rate of initiating a DLB. As a result, SID is employed in the final implementation.

## 2.4 *The Message Passing Interface (MPI)*

The development trajectory for this project has the basic ideas initially realized and tested on a conventional machine. The intention has then to move to a conventional cluster machine, to explore the attributes of running the algorithm on a true multi-core machine, albeit with mature command and delay facilities. In the event, the SpiNNaker hardware has not freely available before the end of the research, so all the experiments reported here are based on a machine utilising conventional message passing technique. Here we describe the MPI (Message Passing Interface) system.

The MPI system is an API (Application Programme Interface) which enables many computers to communicate with each other. MPI is a platform independent communication library. It defines the behaviour of data types, communication types, memory placement and others. By setting standards such as these, it becomes possible for computers running different OS (Operating System) and hardware architectures to communicate with each other. The goal of MPI is to provide a high performance, scalable and portable communication capability[115][116].

The well-defined behaviour of MPI allows both heterogeneous and homogeneous computing systems to execute MPI. Heterogeneous systems are those in which computing nodes within a system have different runtime environments, both software and hardware. In a homogeneous system, the computing nodes are identical to each other, and execute identical software. During execution, processors call MPI routines to communicate with others. The routines can be called from FORTRAN, C and C++. The basic communication unit in a MPI system is the *message*. A message is a block of formatted memory. The type of data within this block of memory can either be predefined by MPI or customized by users.

The program is duplicated in all the processors of a parallel system. The differences of execution may be influenced by the processor ID within the system. The user needs to define the behaviours for different processor IDs.

MPI provides both broadcast and point-to-point communication. In the broadcast mode, a message is sent to all the LPs within the same MPI communicator. A communicator is a label representing a group of LPs. The size of the group may be anything from 0 to all the available LPs. Point-to-point communication only sends messages to a single LP within the same communicator. Communicators can be formed and regrouped dynamically.

MPI is a rich and diverse library, containing a large number of functions and capabilities, such as profiling and parallel IO. Profiling functions can be used to record the timing information during the simulation stage. They provide the performance data needed to evaluate the performance of a parallel simulation system.

In this project, LPs are designed to execute PDES. LPs are decoupled from each other, and hence point-to-point communication is used widely across the system. However, broadcast communication is employed in some of the early developments, which enables the LPs to determine the termination or current system status quickly and across the system. The performance time of a point-to-point message can be split into two components, initiation latency time and actual transfer time. The initiation latency depends on the quality of the local environment and the size of the message, while the transfer time depends on the connection quality. If large memory block messages dominate the communication, the connection bandwidth between computation nodes plays a more important role. In this project, in order to avoid the impact of latency, most of the point-to-point communications are non-blocking. This enables LPs to make a request to a communication handler, and while waiting for the request to be processed by the communication handler, other calculations can be performed. In practice, this means a MPI request made in the code will not stop the progression of execution.

## 2.5 *Summary*

This chapter introduces five main topics related to the DES and its peripheral implementation techniques.

- **Mainstream research** has been focused on pursuing standardized abstract level simulation, for which HLA was created. However, this is avoiding the complexity and moving towards functional simulation rather than improving the actual overall parallel performance. There might be a gap in low level simulation.
- **Conservative and Optimistic approaches** solve the low level discrete event simulation using two radically different methodologies. Further investigation is required to quantify their performance, in order to choose one of them as the base simulation technique for SpiNNaker Discrete Event Simulator (SDES).
- **Static partitioning** is implemented using multi-way partitioning technique which try to minimize the number of connections that exist between different partitions.
- **Dynamic partitioning** is employed as a sender-initiated diffusional technique which reduces the cost of initiating a DLB.
- **MPI** is used to implement the communication system in the emulation system.

# Chapter 3 Preliminary Work

After introducing a range of techniques available to support the implementation of PDES, this chapter focuses on the preparation work that needs to be done before implementing the SpiNNaker Simulator. This includes the test portfolio, the circuit import process, circuit partitioning work and the final selection of which simulation technique to employ.

## 3.1 *The Test Portfolio*

The test portfolio is an essential part of any software project. A set of well-considered circuit examples is essential to explore the different performance limitations of the simulator. The aim of the test portfolio is not simply to provide as many circuits as possible, but to provide circuits that will highlight different aspects of the simulator and hopefully to expose any shortcomings.

To define these circuits, some input language is necessary. The EDA community provides a wide range of languages, but most are far too rich for the purpose required here: to simply define the structure of the test circuit with as little extraneous detail as possible.

### 3.1.1 Circuits in the Portfolio

In order to choose the circuits, a set of testing criteria needs to be identified. In the final simulator, users will need to know the performance of the system in terms of event simulation speed, communication efficiency, and static and dynamic partitioning effectiveness. The circuits in the portfolio must be carefully chosen to illustrate - preferably quantitatively - these effects.

The aim of parallel simulation is to reduce the overall simulation time for all simulation scenarios. Ideally, the simulation time will be reduced as the additional computational

power kicks in. This is because in the ideal case, the event processing work *w* is evenly distributed among *n* LPs. Assuming the speed of event execution is constant and no control overhead is involved, the evenly distributed event work *w/n* will be processed in all the LPs using *1/n* of the original time in serial execution. This performance *speedup* effect can determine the control overhead involved when more and more processors are engaged in simulation. The *speedup* data provides a good indication of the scalability of a simulation technique.

To extract the best ***speedup data*** of an event simulation engine, a good way is to load the simulator with massive parallel circuits. In this case, a large number of individual 3-stage ring oscillators is an elegant possibility. Each individual ring oscillator generates events for an indefinite period; putting *n* identical ring oscillators together can generate *n* events at each discrete time. When these oscillators are partitioned in a way that no inter-partition wire exists, the cost of event communication can be eliminated, and only control messages exist during a simulation: hence the parallel control overhead can be detected. The ring oscillator circuit is shown in Figure 35. The *n* chosen in this project is 100.

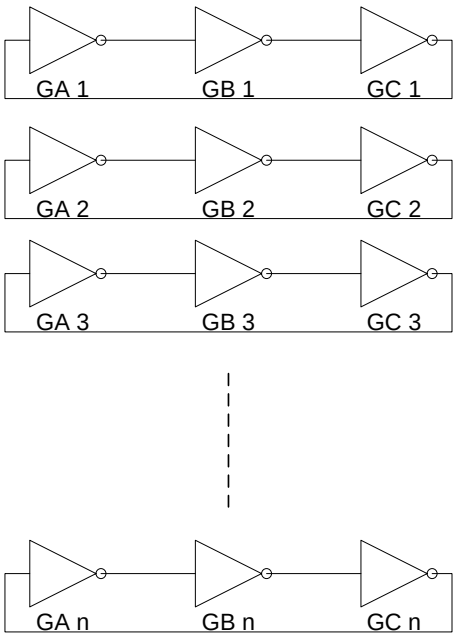



Figure 35 Ring oscillator circuit

The ring oscillator will constantly generate a large number of events at a discrete time instance after initial triggering. Assuming that the ring oscillators are distributed evenly within the simulation system, as the number of LPs employed during the simulation increases but much less than *n*, the speed of the simulation should increase accordingly. As no events will be sent between LPs, in theory, there is no communication cost involved during this simulation. As a result, a linear relationship is expected between the number of processors employed and the final speedup ratio, if the number LPs engaged in simulation is much less than *n*. The speedup ratio is measured by dividing the simulation time for a single LP by the simulation time for multiple LPs.

After evaluating the best case scenario for the simulation engine, the worst case scenario for demonstrating the scalability of the simulator should be tested in order to show the lower performance limitation existing in SDES. When a simulation does not have more than one event that can be processed in parallel, there is no parallelism within the simulation. In this case, parallel simulation will only bring an additional control overhead to the simulation without any improvement on the speed of event processing. Under these circumstances, the parallel control overhead can be evaluated, as the performance difference between serial and parallel simulation mainly consists of the additional control overhead. As an example circuit, a single chain of inverters is used and only one event fed to the source (Figure 36).

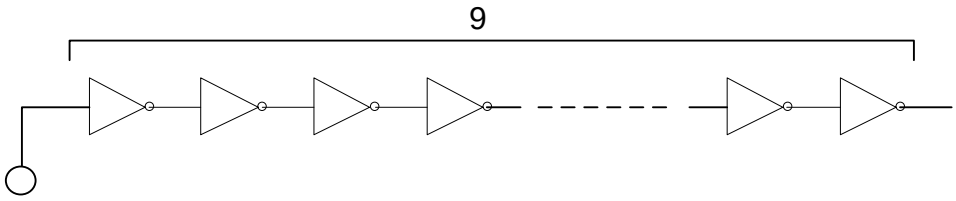



Figure 36 A series of 9 inverters

In this simulation, the best performance that can be achieved is simulating the circuit in a sequential environment. In a serial simulation, there is no communication or control overhead involved in the simulation. When porting this design to be run on parallel platform, this enables the boundary control overhead to be identified. As there are two levels of boundaries in SpiNNaker system, monitor and slave level, by mapping the same components to different setting, either mapping all the components onto a single

node using only one monitor process, or mapping each component to a node using 9 monitors, the cost of these boundaries can be identified.

The parallel overhead identified above includes the communication overhead and the parallel control overhead. As a single event will travel through the partitions sequentially, it creates events crossing LPs during the simulation. Another interesting feature about this circuit is the effect of varying the rate at which input events are pipelining through the simulated circuit. As the number of events within the circuit increases, the speedup curve will gradually recover from negative speedup to positive speedup.

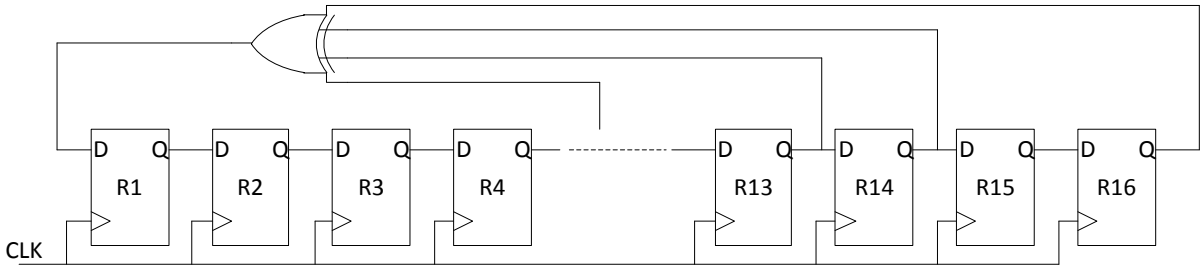



Figure 37 16-bit LFSR circuit

The performance of the 16-bit LFSR (Linear Feedback Shift Register) sits between the two circuit examples shown above. The LFSR circuit generates pseudo-random numbers by shifting the overall bit pattern in one direction within the registers and feeding selected bits back to the most significant bit of the register (MSB). When this circuit is partitioned into *n* blocks, where *n<16*, and mapped to *n* LPs, there is a constant flow of data between the LPs. However, it is very different from the series of inverters example, because the LFSR requires a limited amount of computation within each LP. It is a good example to show how the simulator speedup responds to the additional computation required in addition to the serial property of the LFSR circuit. This circuit is used to compare the performance of the three simulation techniques investigated in section 3.3 .

The ***cost of communication*** is another critical parameter of the simulator. There are two general communication layers in the final simulator. The bottom layer is the MPI communication cost for sending messages between different processes. The top layer is the simulation layer which deals with the event passing between different LPs. We need to devise a strategy to distinguish between these aspects of the simulator.

As the simulation is based on a cluster system, the cluster itself has its own overhead for processing the MPI-based communication and control overhead, which differs from the behaviour of SpiNNaker. As a result, it is necessary to measure and understand these overheads in order to correct the quantitative measurement made in the cluster environment and try to predict the performance of SDES on the SpiNNaker platform.

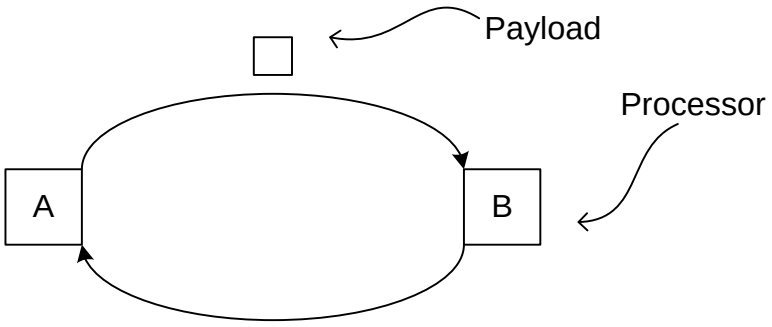



Figure 38 MPI communication cost evaluation mechanism

The most common way of obtaining the *message passing time* is to set up a Ping-Pong like message passing mechanism (Figure 38) [117]. As the time taken for a single sending and receiving action in MPI is very short and is highly variable (see section 2.4 for detail), a sensible way to measure the cost is to evaluate the time taken to send messages in batches, by letting two processes send and receive a single message in a circle and by doing this a number of times.

There is a possibility that the system may recognize this communication pattern and try to optimize the process and hence reduce this communication overhead. To avoid this mis-measurement of communication cost, the communication instructions used in this test are blocking send and receive commands, which means the program will not execute the next instruction until the current instruction is completed. The

communication cost for a single send and receive can be calculated by dividing the total time for the whole batch by the number of times the messages are sent and received. The result can then be compared with the communication cost for SpiNNaker, and we can estimate the speedup that can be produced by porting the simulator to SpiNNaker platform.

In addition to the communication cost for the basic MPI layer, the cost for event communication is also a necessary parameter to evaluate the communication performance. An event message consists of multiple MPI messages, because in SpiNNaker the communication is based on *packets* and the packet payload is 32-bit in length. Messages, which are longer than 32-bit, need to be sent over multiple packets. As a result, a communication protocol is required, as there are many different types of simulation communication signal going through these 32-bit packet payloads. Although the MPI layer can transfer messages which contain more than 32-bit information, all the test results under SDES are based on 32 bits of MPI messages. SDES is designed to enable easy porting to the SpiNNaker system. As a result, a complete message decomposition and reconstructing mechanism is designed. The detailed communication code map and message handler are discussed in the communication platform section 4.2 .

On the MPI platform, packets are very unlikely to be lost in the communication system, and it is assumed in SDES that this will be the same on SpiNNaker system. However, in reality, there is a possibility that the SpiNNaker system can drop packets due to various reasons. In this thesis, this problem has not been dealt with, and further development is required if the system is to be tolerant of transient messages dropout.

When the communication protocol is employed, all the messages are split into multiple 32-bit payloads in the sending process. When receiving them, all the payloads need to be reconstructed and the original message restored. These splitting and reconstructing actions take time and they are an essential part of the performance evaluation data. Therefore, it is necessary to examine the cost involved.

The cost of event communication *within* the system can be examined by creating circuits which focus on sending and receiving events. The following circuit tests the event communication performance of the simulator in a similar manner to that in which the message transfer speed in MPI is evaluated. The difference being the measurement unit changes from a single message to a single event in this test. A single event in the SDES consists of multiple messages, so the time to transfer them is greater than a single message by itself. And the performance of a communication in a simulation is normally measured in the number of events instead of the number of individual messages.

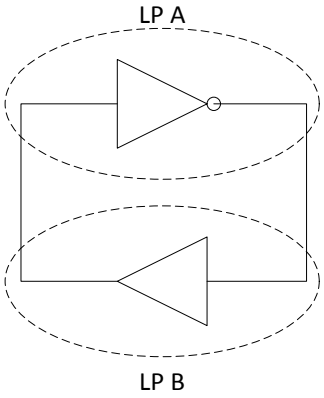



Figure 39 Event communication performance test circuit

In Figure 39, the test circuits are simulated separately. During the simulation, components are evenly distributed across two LPs. The two gates test the sending and receiving performance of the simulator simultaneously. By counting the number of events being sent over the network in the circuit and dividing it by the overall simulation time, the effective time of transferring an event can be calculated.

The ***partitioning algorithm efficiency*** is another key parameter of the simulator. The ability of partitioning algorithms to minimize the wire cut in the static model and to balance the workload in the dynamic model is essential to the final simulation performance. For the *static model*, the two quality measurements are the number of wire cuts and the balance of the component count between different partitions. The logic behind minimizing the number of wire cuts during the partitioning is to reduce the communication cost during the simulation time. Without the live workload data, the

partitioner can only assume all the wires are *equally weighted* on workload and hence the fewer wire cuts the better.

Moreover, when the partitioner tries to reduce the number of wire cuts, there is a tendency for the components to be pushed into a single partition where the number of wire cuts is naturally zero. This, however, will defeat the purpose of a partitioning algorithm. Hence, the number of components that can reside on a single partition is bounded by a minimum and a maximum limit. For this purpose, two circuits are created to test the efficacy of the static partitioner.

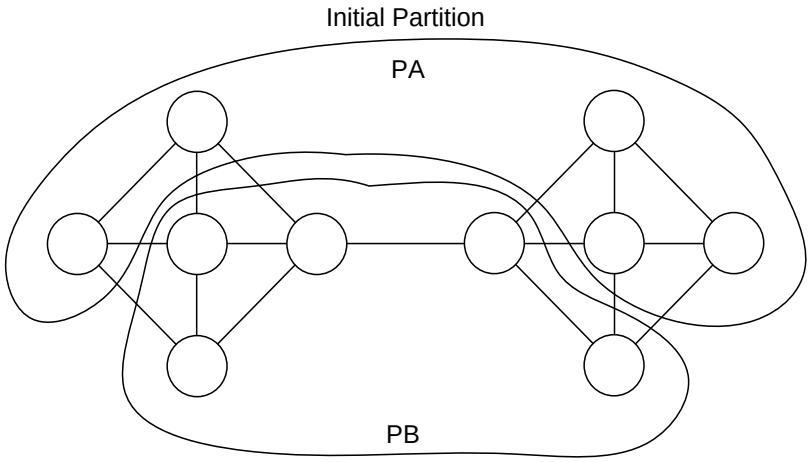



Figure 40 Static partitioning mechanism test circuit 1

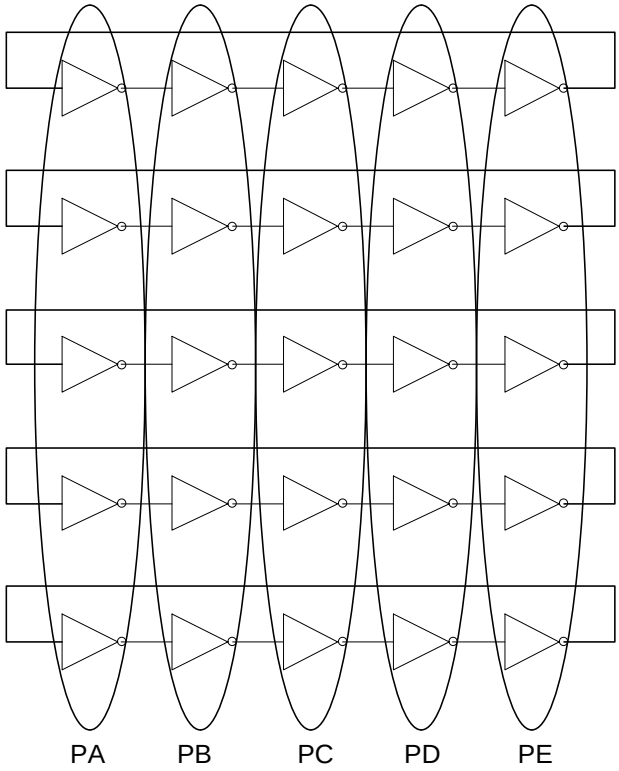



Figure 41 Static partitioning mechanism test circuit 2

The best partitioning result for both the circuits of Figure 40 and Figure 41 can easily be solved by inspection. In Figure 40, the initial partition splits the components into two sub-circuits and hides the minimum cut (min-cut) solution inside one of the partitions. Ideally, when the partitioning algorithm has finished the partitioning, there will be only one wire cut instead of eight. This simple circuit illustrates whether the partitioning algorithm has the capacity to find a min-cut solution to a network topology. In Figure 41 however, due to the maximum and minimum component count limit, the system may not be able to reach the optimum partition. For example, moving most of the components from one partition to another partition will not save any wire cuts; it is hard for the algorithm to get the big picture. It is therefore a good example of a test circuit to show how the algorithm can overcome this min-cut problem with constrains on the component count. These are good example test circuits because there are obvious best results to act as references to the actual partitioning result.

At the other end of the partitioning spectrum is the *dynamic technique*. In contrast to the static algorithm, the dynamic technique tries to balance the workload on-the-fly, and it

attempts to make the computational work evenly distributed among the LPs. The figure of merit here is not the wire cut, but the signal traffic density. In turn, the simulation speed will also increase due to the shortened calculation time in each LP.

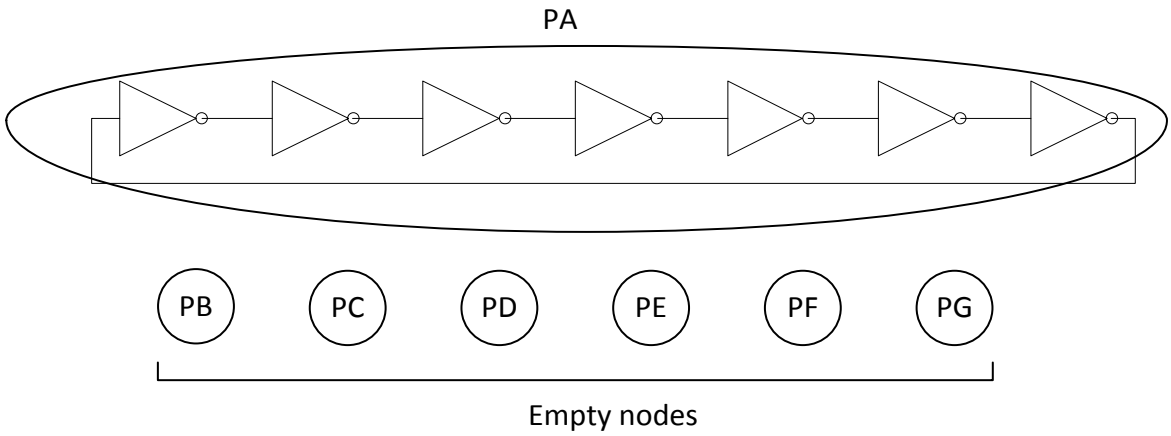



Figure 42 Seven-stage clock generation circuit for dynamic partitioning test

*First and foremost*, the dynamic algorithm must be able to demonstrate that it is able to distribute the workload evenly across all LPs available in the simulation system. An intuitive way of testing would be allocating the *n* components to a single LP and simulating this circuit using *n* LPs. This allows the dynamic partitioner to migrate the components into the empty LPs during simulation. Given a long enough time to perform the simulation, the components will become evenly distributed among the LPs, in which case each LP will hold one component. In order to show this property, a simple seven-stage clock generation circuit is used (Figure 42). This simulation starts off by putting all 7 components onto a single LP to see if they will propagate to other LPs during the simulation. If they are evenly distributed at the end, this illustrates that the dynamic partitioning mechanism is behaving correctly.

The static partitioning system can be defeated by certain pathological circuits. In other words, the minimum wire cut cannot cause even distribution within the workload: see Figure 43. Dynamic partitioning is vulnerable to similar pathologies.

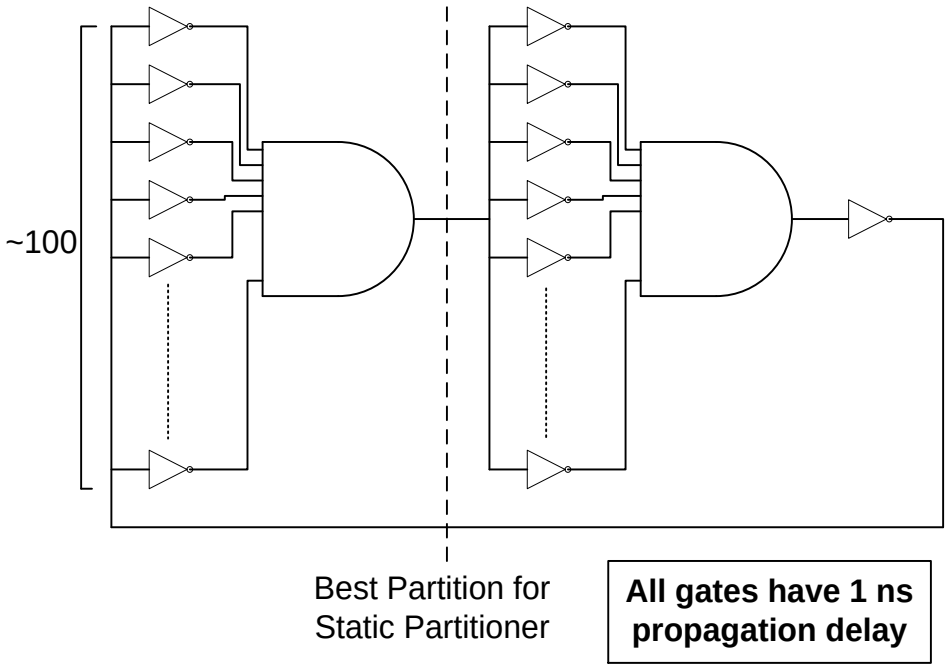



Figure 43 Dynamic partitioning test circuit, two heavily connected AND gates

In dynamic partitioning, the balancing figure of merit is different to that of the static algorithm. The aim of dynamic partitioning is to balance the *workload*, not the absolute component count. As a result, a distorted workload distribution may cause the partitioner to move away from the best partitioning solution, even though the initial partition is already the best solution. In Figure 43, the min-cut partition lies between the two AND gates. However, the workload will shift from one AND to the other temporarily, once the circuit starts oscillating. In the course of a simulation, the workload is balanced within 1% of difference between the two. However the temporarily imbalanced workload may cause the dynamic algorithm to rectify the situation by moving the inverters between the two LPs and balance the workload. This will increase the wire cuts but the computational time should be cut in half due to the exploitation of the internal parallelism within the circuit.

A further circuit illustrates the working mechanism of the dynamic partitioner. The last example was designed to establish how the partitioner will react to a short term workload imbalance in the circuit. The next example circuit will initiate activities in the circuit step by step: the dynamic algorithm must deal with the newly introduced workload and balance the workload across the system step by step accordingly. This

circuit (Figure 44) starts off the simulation with the optimum static partitioning setting, but the workload starts unbalanced and slowly moves towards equilibrium under the edges of the dynamic partitioner. The *waveforms* associated with Figure 44 are shown in Figure 45.

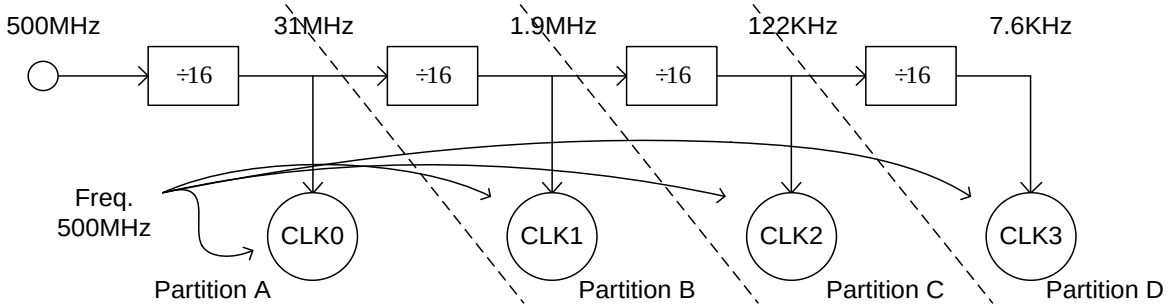



Figure 44 Delayed activation of clock generation circuit

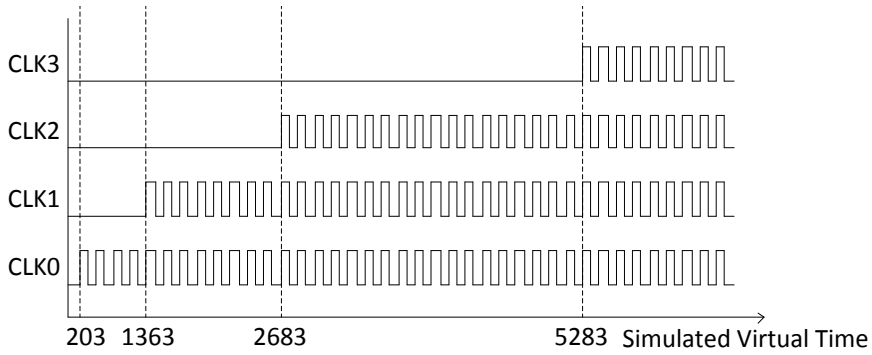



Figure 45 Waveforms for delayed activation of clock generation circuit

The clock nodes circuit within Figure 44 are identical. Each clock node (CLK0, CLK1, CLK2, and CLK3) contains ten clock oscillators. The events within the frequency dividers can be ignored due to the low activity rate compared with the clock nodes. The events that will have an effect on the dynamic partitioning are the events generated by the clock oscillators themselves. When the first clock node (CLK0) is activated, the partitioner will distribute the clock oscillators inside CLK0 into several partitions. When the second clock node is activated, the dynamic partitioner will share out workload again and the process continues. The problem is whether the partitioner can return to the original partition, which is the optimum partitioning setting, at the end of the simulation.

As well as the specifically constructed circuits, it is important to test the simulator on *real* and *complex general circuits*, proving that SDES is capable of simulating complex circuits employing the tool chain. These general circuits are defined in behavioural VHDL, where all the other purposely constructed circuits are described in hierarchical Bench file format (see detail in Appendix B.4). Two general circuits are used for this: a Data Encryption Standard (DES) circuit [118] and a Finite Impulse Response (FIR) circuit. Both circuits contain computationally intensive blocks, and as a result, are good candidates for exercising parallel simulation. The determination of the speedup for going parallel is the objective for these two tests.

The DES circuit provides an easy way to test and verify the behaviour of the simulator while maintaining a high level of complexity at the same time. The DES circuit has three inputs (Figure 46); first input is the de/encryption key which consists of a 56-bit key plus 8 parity bits. The second input is the 64-bit data, which can be encrypted or decrypted using the key provided. The third input is the decryption/encryption switch that controls the behaviour of the DES block.

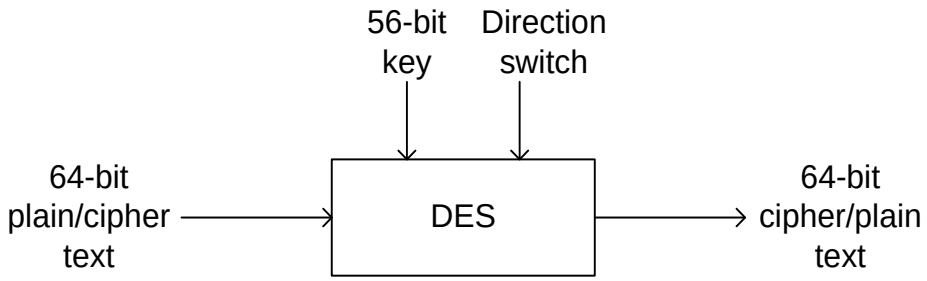



Figure 46 DES circuit package

If the DES block is set to *encryption* mode, the 64-bit data will be encrypted to cipher text which is also 64-bit and vice versa. With the same key, the cipher text can be restored to plain text. It is a simple way of verifying the functionality of the simulator.

The second general circuit is a FIR filter. This is a massive circuit employing sixteen 16-bit multipliers, which in theory will generate a large quantity of events, demonstrating the speed advantage of parallel simulation. A FIR filter performs a weighted sum of input signals. The FIR is formed from a series of *delay* components

and a *tap* attached to each of these delay components (Figure 47). Each tap multiplies the signal by a coefficient. The products after multiplication are summed to form the filtered signal output. However, the filtering feature of FIR is not the purpose of using a FIR filter. The reason to choose a FIR filter is the multiplier involved in the multiplication stage at each tap. When flattened, this creates a very large gate count.

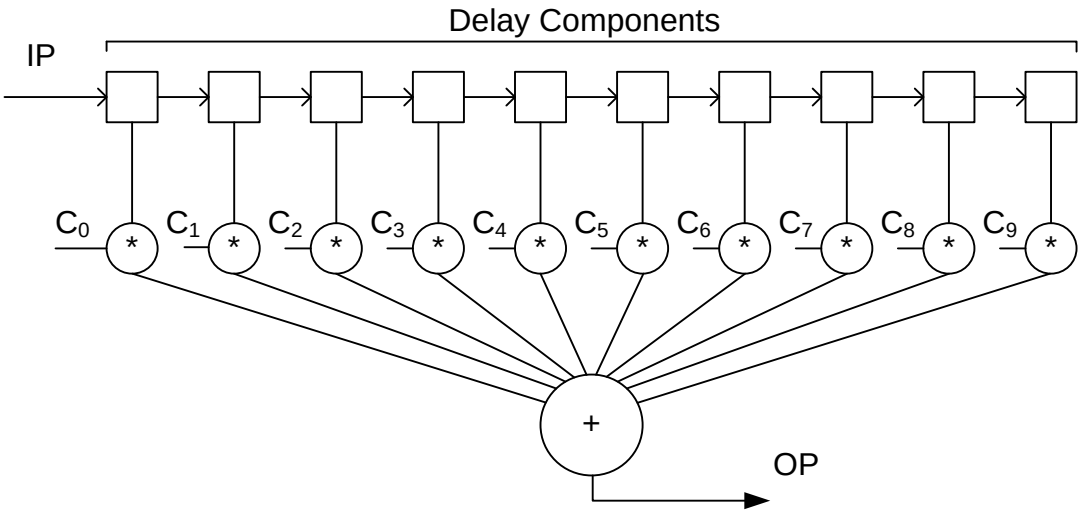



Figure 47 FIR filter circuit

The FIR filter generates a lossy transform of the input signal: the original signal cannot be reversibly reconstructed from the output. Therefore, in order to verify the correctness of the FIR simulation, the SDES simulation result has to be compared with a reference result, generated by ModelSim [119]. The same circuit is tested using the same input events, and the results are checked against each other.

These are the set of circuits that will be tested under SDES. They may be grouped as event processing performance tests, communication tests, partitioning tests and two general circuits. The results of this portfolio will be listed and analyzed in 0.

### 3.1.2 The Tool Chain

It is necessary to establish a tool chain to support PDES. Circuit topologies may be input as a subset of *behavioural VHDL* [18] or extended version of the widely accepted *Bench circuit format*. At the end of the simulation, results are stored in VCD (Value Change Dump) [120] format, which can be read by a wide range of waveform viewers.

The Bench file format is used to define ISCAS (International Symposium on Circuits and Systems) test circuits. The original format is extended to enable a Bench file to contain hierarchical information, which allows users to input complex systems from a hierarchical perspective. The behavioural VHDL widens the range of accessible input circuits.

The tool chain for SDES consists of MOODS, two parsers, a function generator, a circuit converter, a static partitioner, the simulator and a waveform viewer. MOODS is a behavioural synthesis tool created in the electronics and computer science department at the University of Southampton. It takes behavioural VHDL as input, and produces structural VHDL as an output. Details of MOODS can be found in Appendix B. The entire work flow is shown in Figure 48.

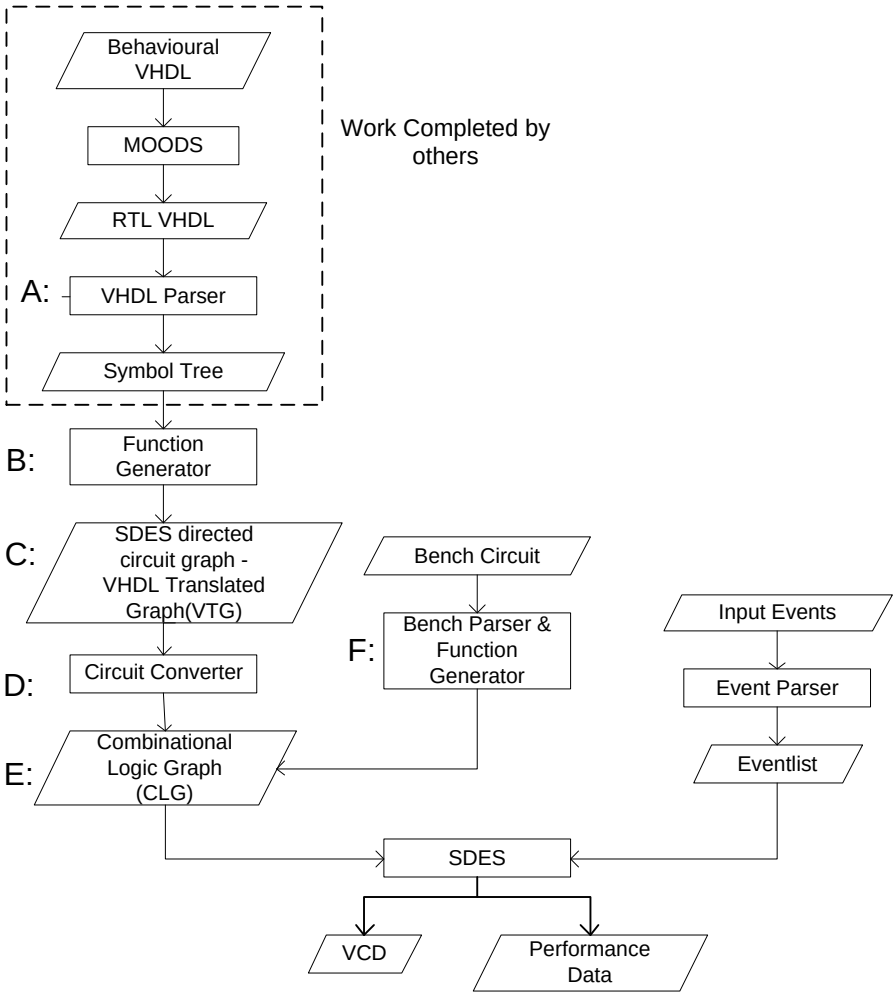



Figure 48 The tool chain of SDES

The first parser (A) takes MOODS output (structural VHDL) and converts it into a symbol tree, which contains all the information contained in the VHDL. Within the tree structure, the type of data represented in the original VHDL symbols is also stored along with the original code. Detail of the symbolic tree can be found in [121], [122].

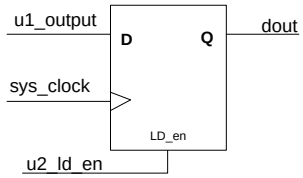
The function generator (B) processes this symbol tree and converts it into a SDES usable directed circuit graph, VTG (C). However, the structural VHDL in the MOODS output circuit uses a Xilinx cell library as its synthesis library, so there are many complex components that are represented as a single node. Each of these complex component instantiation is converted into a single node in the directed circuit graph by the function generator. These complex nodes require another conversion operation (D) to expand them into fully combinational circuits (E). The circuit converter (D) expands the complex nodes (multiplier, adder, comparator and so on.) into basic combinational gates. This, on average, raises the complexity of the circuit by an order of magnitude.

The partitioning method employed in this project does not require the structure knowledge of the simulated problem, as a result, all the components are flattened to the basic logic gates directly. If other partitioning methods were used, the partitioning might be carried out at the higher VTG level (C) in addition to the CLG level (E).

As well as complex behavioural VHDL, the system is capable of simulating simple circuits. A second parser (F) is designed to parse the *extended Bench circuit format*. This Bench file format can be used to construct simple circuits such as the circuits for estimating the cost of communication. This is very different from the general circuits in the portfolio where the behaviour is mainly focused on the parallelism within the circuits. There is another function generator connected to the end of this Bench file parser, which produces the circuit graph that can be used by the parallel simulator. The detailed definition and explanation of this Bench file format is in Appendix B.

Structural VHDL will be converted into a VHDL Translated Graph (VTG) (C), defined in this thesis. VTG represents a circuit using MOODS library components, which consists of 56 types of components, plus 6 additional assistant components. The 6 additional components help complement the functionality of the MOODS library by adding some VHDL inherited functions, such as buffer, IO, ROM Decoder and so on. At a less abstract level lies another type of circuit graph simulation format used in SDES, the Combinational Logic Graph (CLG). In order to increase the circuit complexity, these library components need to be expanded into combinational logic. CLG is a convenient way to describe the combinational logic. The *hierarchical Bench circuit format* is also converted into CLG for SDES simulation.

The VTG is a directed circuit graph representing both gates and wires as nodes. Arcs in the graph contain the width of the signal bus connecting in between. This simplifies the reconstruction of a circuit at the parsing stage. The CLG represents gates using nodes and wires using arcs. This reduces the data searching time for the simulator at runtime. Illustrations of both formats for a 16-bit register are shown in Figure 49.




**VHDL representation**

```
u2: REG_1
  generic map (n => 16)
  port map (input => u1_output, ld_en => u2_ld_en, ck =>
sys_clock, out_q => dout);
```

Transition A

**VHDL Translated Graph representation**

Node Format:
@ ID,Name,Type,Delay,Value,Partition,Parameters[3]

@ 11,sys_clock,0,0,U,0,0,0,0
@ 92,u2_ld_en,0,0,UUUUUUUUUUUUUUUUU,0,15,0,0
@ 222,u1_output,0,0,UUUUUUUUUUUUUUUUU,0,15,0,0
@ 564,u2,45,1,,0,16,0,0
@ 9,dout,0,0,UUUUUUUUUUUUUUUUU,0,15,0,0

Converted

Arc Format:
^ID, Source, Target, MSB, LSB

^564,9,15,0
^222,564,15,0
^92,564,15,0
^11,564,0,0

Transition B

**Combinational Logic Graph representation**

Gate Format: ID Type Delay Flag Partition Value MSB LSB Workload
- 3715 6 1 0 0 UUUUUUUUUUUUUUUU 15 0 1
- 3717 6 1 0 0 UUUUUUUUUUUUUUUU 15 0 1
- 3719 6 1 0 0 U 0 0 1
- 3721 13 1 0 0 UUUUUUUUUUUUUUUU 15 0 1
- 3722 7 1 0 0 UUUUUUUUUUUUUUUU 15 0 1
- 3723 0 1 0 0 UUUUUUUUUUUUUUUU 15 0 1
- 3724 0 1 0 0 UUUUUUUUUUUUUUUU 15 0 1
- 3725 2 1 0 0 UUUUUUUUUUUUUUUU 15 0 1
- 3726 1 1 0 0 UUUUUUUUUUUUUUUU 15 0 1
- 3727 1 1 0 0 UUUUUUUUUUUUUUUU 15 0 1
- 3728 1 1 0 0 UUUUUUUUUUUUUUUU 15 0 1
- 3729 1 1 0 0 UUUUUUUUUUUUUUUU 15 0 1
- 3730 1 1 0 0 UUUUUUUUUUUUUUUU 15 0 1
- 3731 1 1 0 0 UUUUUUUUUUUUUUUU 15 0 1

Wire Format: ID Delay MSB LSB Source Target
- 3732 0 0 0 3719->3721
- 3733 0 15 0 3717->3722
- 3734 0 15 0 3715->3723
- 3735 0 15 0 3717->3723
- 3736 0 15 0 3722->3724
- 3737 0 15 0 3731->3724
- 3738 0 15 0 3723->3725
- 3739 0 15 0 3724->3725
- 3740 0 15 0 3727->3726
- 3741 0 15 0 3725->3726
- 3742 0 15 0 3721->3727
- 3743 0 15 0 3726->3727
- 3744 0 15 0 3728->3727
- 3745 0 15 0 3721->3728
- 3746 0 15 0 3729->3728
- 3747 0 15 0 3726->3729
- 3748 0 15 0 3728->3729
- 3749 0 15 0 3727->3730
- 3750 0 15 0 3731->3730
- 3751 0 15 0 3728->3731
- 3752 0 15 0 3730->3731

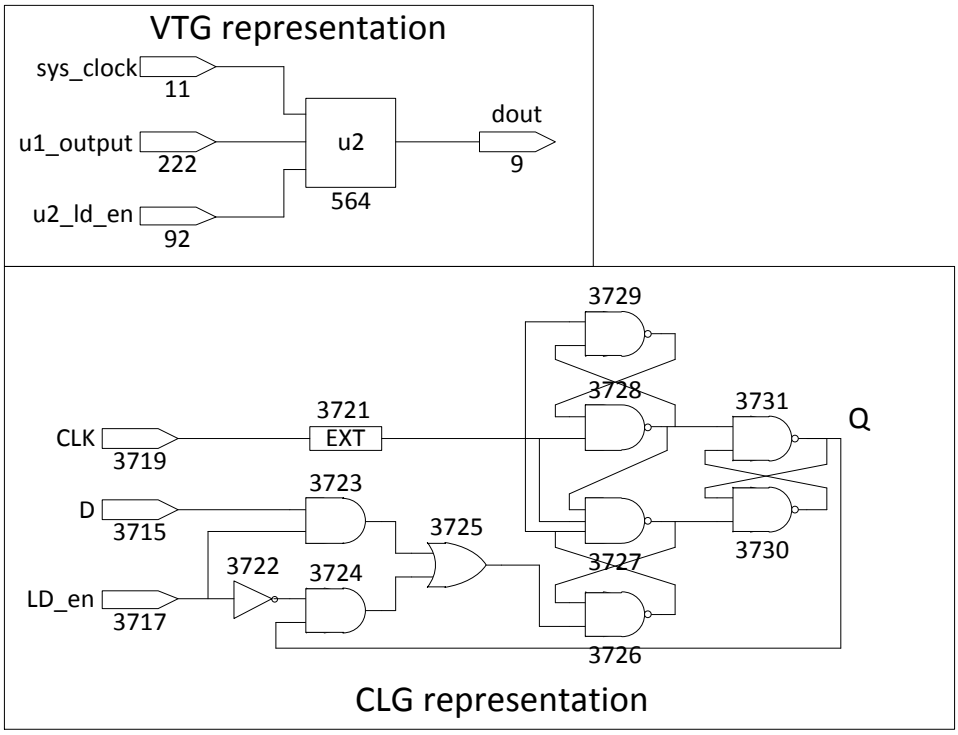



Figure 49 16- bit register in VHDL, VTG and CLG

Figure 49 shows how a single complex gate (node 564) in the VTG can be expanded into multiple gates in the CLG. Other nodes in the VTG (11, 92, 222 and 9) are wires, which are replaced by arcs in the CLG. The function generator converts a MOODS generated VHDL gate into a single node (transition A in Figure 49). The most unusual component in the VTG format is the parameter array attached to each node. This parameter array is used to store the MSB and LSB when defining a wire in a circuit, and defines the specification parameters in case of gates. In the sample complex gate, the only parameter here is the width definition of the register, which is 16 bits. This information is stored in the parameter array of the complex gate, as shown in node 564 in VTG representation. Following the VTG, a circuit converter, which performs transition B in Figure 49, expands the VTG into the CLG according to the predefined structure. The code and graph of VTG and CLG is shown in Figure 49. Further details about the circuit converter and function generator can be found in Appendix B.

The *partitioning algorithms* are required to split up the circuit for use in parallel simulation. The job of the partitioner is to split the directed graph into many parts and map them onto the available LPs. This provides an initial static partition for a simulation. Further dynamic partition is carried out on-the-fly during a simulation.

Circuit input events are parsed and formed as events, which can be used by PDES directly. This is the right hand side of Figure 48. The formats of input events are detailed in Appendix B.5. The simulation *output* is in the form of VCD files, which can be read by a wide variety of freely available waveform viewers [123–126]. Along with the simulation results, the performance data is generated while the simulation is on-going. These performance data are mostly accumulators, which are stored in memory during simulation. At the end of a simulation, these data are extracted to different log files created by each LP. This is the end of the tool chain. For a full description of the circuit graph generation process refer to Appendix B. The full details of this process relating to the main simulation are explained in the rest of this chapter.

## 3.2 ***Partitioning Techniques***

The complexity of the wire cut data increases *exponentially* with the number of partitions involved. The number of $G_{moving}$ values for each gate is equal to total number of partitions ($N_{total}$) minus 1, which gives the total number of $G_{moving}$ the value of $N_{total}(N_{total}+1)$. This is because each component has $N_{total}$-1 potential movements to make, so the total size of $G_{moving}$ in the entire graph is $N_{total}(N_{total}+1)$. By increasing the number of partitions by *x*, the total number of $G_{moving}$ will increase by

$$x^2 + (2N_{total} - 1)x$$

This is calculated by substituting the $N_{total}$ with $N_{total}$+x in the original formula and simplifying. After acquiring the $G_{moving}$ values for all the components in all the partitions, the components are sorted according to their maximum $G_{moving}$. This provides a reference for the partitioner to decide which component movement can produce the highest gain. The $G_{moving}$ information is stored as a stl::map<gain, stl::map<graph iterator, stl::vector<partition info> > > as in Figure 50.

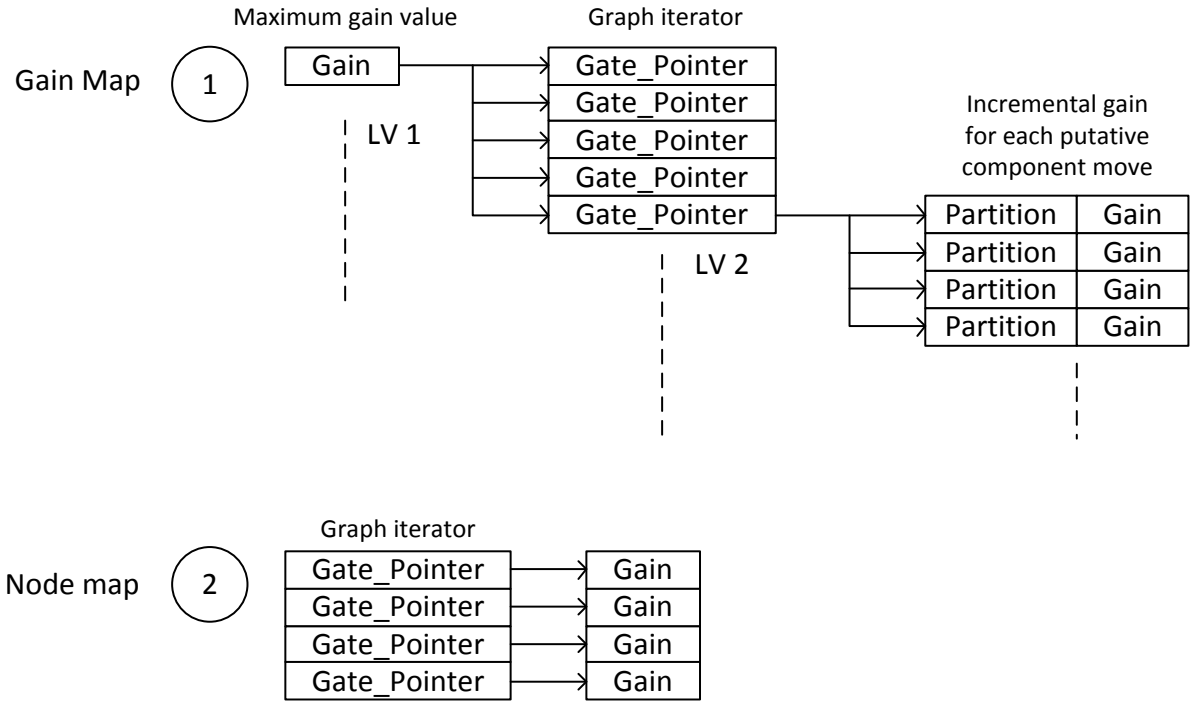



Figure 50 Partition gain map structures

There are two maps used by the static partitioner, the *gain map* and *node map*. The gain map provides a full database of gain information for each component. It contains the

maximum gain value, CLG graph iterator and the incremental gain for each putative partition move of every component in the circuit. Gates with the same maximum gain value are stored under the same gain value entry in the gain map. When the partitioner wants to update the gain value for a specific component, a second map, the node map, is created to speed up the process. The graph iterator is the index in the node map of the maximum gain value of the component; this allows the partitioner to access the component within the gain map directly. With the database of gain values ready, the system executes multi-way partitioning algorithm.

Taking Figure 51 as an example, if the network is partitioned according to the structure shown on the left hand side of the figure, the corresponding gain map for the components is shown on the right hand side of the figure. The map is constructed by calculating the gain of all the components, and sorting them into each gain bucket. Taking component “6” as an example, if it is moved from PA to PB, the total wire cut will reduce by one, so in the gain map it is represented as “PB: +1”. If component “6” is moved from PA to PC, a wire cut reduction of two can be obtained, which is represented as “PC: +2” in the gain map. The component will be stored in the gain bucket using the highest gain obtainable by this component, which makes it stored in +2 bucket of the gain map. Applying the same process to all components, and the gain map can be obtained.

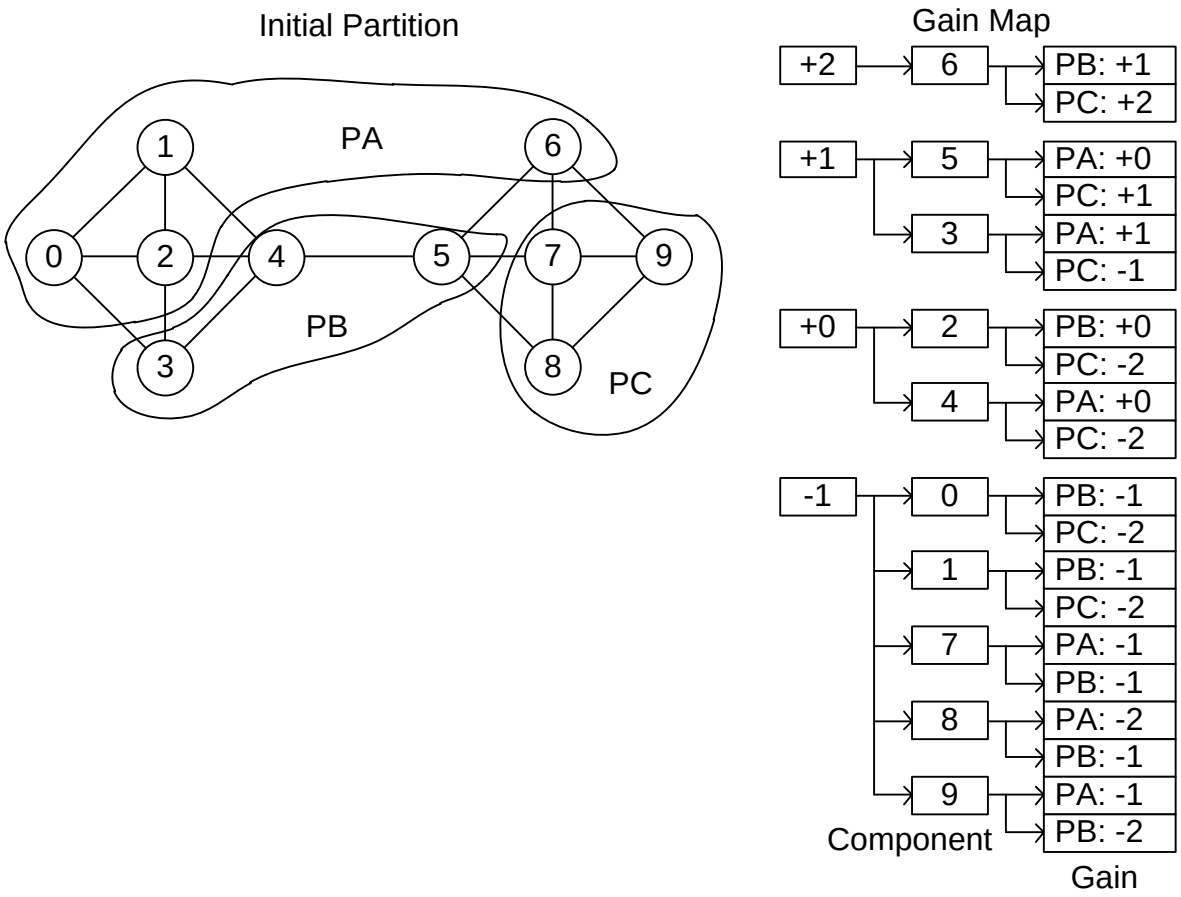



Figure 51 Partition gain map example

Similar to the labelled components in the K-L algorithm, when a component is moved from one partition to another, this shifted component will be prevented from further movement in current iteration. In practice, instead of labelling the components, the shifted components will disappear from the gain map after every movement. The improvement steps in one iteration continue until the highest gain in the current gain map is equal to or less than zero. This will trigger the gain map to be reconstructed for all the components in the left over circuit, and repeat the improvement steps iteration by iteration. When a new iteration does not introduce a reduction in wire cuts, the iterative process stops, and this final circuit is the minimum cut partitioned circuit.

The quantitative description of the behaviour of the dynamic partitioning algorithm is provided after the discussion of the simulation algorithm itself.

## 3.3 *Investigation of Simulation Techniques*

A wide range of simulation techniques that are available for parallel simulation was discussed in chapter 2.2.2 . In order to decide which simulation technique to use as the basis of the SpiNNaker simulator, three of the simulation techniques were implemented, tested and analyzed. Before jumping into how to implement different simulation techniques, we need to establish quantitatively how they will be compared. This determines the effectiveness of a simulation technique in dealing with discrete event simulation.

*First and foremost* is the simulation speedup. When all these simulation techniques are tested against the same circuit, the system with the lowest simulation time would appear most suitable for further development on the SpiNNaker platform. The *second* requirement for picking the simulation technique is the *scalability* of a simulation technique. In the final design, simulations will be carried out on a massive parallel computing platform which in turn implies that the simulation technique should be capable of fully distributed computation. The *third* measurement should take into account the computational efficiency. As data synchronization and communication become an integral part of parallel simulation, the proportion of time taken to execute these administrative tasks should also be taken into account when choosing the appropriate simulation technique.

In this preliminary test, three circuits from the portfolio are used to evaluate the performance of these simulation techniques: an array of clock generators, a 128-bit LFSR circuit, and a DES circuit. The array of clock generators evaluates the best parallel computation performance, the LFSR circuit tests the serial performance of the simulation technique, and the DES circuit mixes the two properties, which provides a general reference as to how a simulation technique copes with both aspects. Quantitative data on three of the techniques described in section 2.2.2 is presented in section 3.3.4 ; the implementation of these three techniques is described before the results.

### 3.3.1 Time Warp Rollback Simulation

The first simulation technique implemented is the time warp rollback simulation technique. As it is an optimistic simulation technique and the most complex technique of the three that were chosen, the lessons learned during implementation pave the way for the other two to be implemented. As explained in section 2.2.2 , this technique does not require centralized control over the individual simulation processes. The simulation processes execute all available events following an ASAP rule and the data synchronization problem is taken care of by the state restoration stage called time warp.

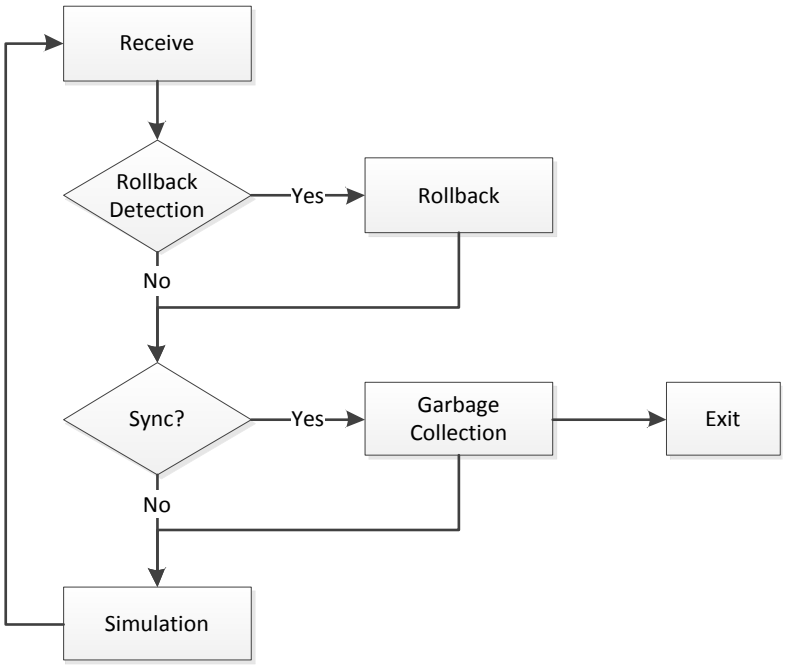



Figure 52 Implementation of Time Warp Simulation Technique

The time warp technique is shown in Figure 52. In the *receive* process, the simulator receives all types of messages from any processes. This includes messages, anti-messages and synchronization signals. Both messages and anti-messages are added either to the main event list or to the state saving module according to their timestamp relative to the current local time within the process. Whenever an anti-message meets a message, both messages will be annihilated from the system.

A time warp action will be triggered in the *rollback detection stage* if a straggler event appears in the main event list of the system. The rollback in time will restore the system state as well as the main event list, as both of them plus the current time form a

complete state description of the discrete event simulation. In this implementation of the time warp system, the state saving technique employed is incremental. This technique does not save a discrete set of absolute system state but instead logs the *changes* within the simulation process and hence has the ability to roll back to an arbitrary past time in the time scale without consuming large amounts of memory. However, the downside to this technique is that the restoration time cost increases as the distance between current and target time increases. Moreover, using the direct cancellation technique (see section 2.2.2 ) employed in this simulation, anti-messages are sent if a message is sent between the rollback target time and the current simulation time. After this rollback detection stage, the simulation environment is ready to execute, whether it has been rolled back or not.

Following the rollback detection stage, the system is ready for any *synchronization* to be carried out if necessary. In this preliminary test of the algorithm, synchronization is carried out every 5 seconds of wall-time. This limits the amount of synchronization required during a simulation. The synchronization is used to determine the GVT (see section 2.2.2 ) and release the memory involved in logging the simulation activities. It also provides a reference for the simulation processes to determine the end of the simulation.

The last stage in time warp simulation is the *simulation stage*. Event evaluation and any updates to the system state are executed at this stage. The operation of this stage is nearly identical to serial simulation (see section 2.2.1 ) apart from the communication links involved during simulation. If any event is linked to an external LP, an event message will be sent out to the target LP to notify the new state change. The external connections for these events will be searched within the circuit, and results are stored in a list which guides the MPI communicator to send out the event.

### 3.3.2 Time Bucketing Simulation

This technique relies heavily on synchronization. This algorithm is very suitable for a shared memory architecture, as large amounts of synchronization are involved during a simulation. Unlike the time warp technique this is a conservative technique which does

not allow simulation to breach event causality. The way to guarantee the work boundary is to synchronize the simulation target time before the actual simulation is carried out in each individual process.

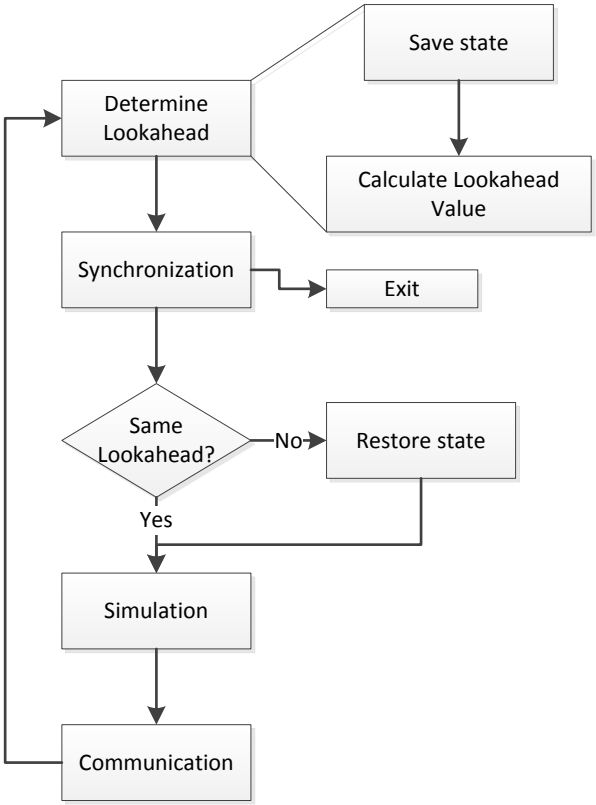



Figure 53 Simulation processes in the time bucketing technique

There are four stages involved in the time bucketing simulation: lookahead, synchronization, simulation and communication (Figure 53). In the *lookahead* stage, each logic process performs simulation of its local circuit up to the time where the first external event is generated. If no such event is found or the limit of execution is reached, the lookahead stage will pause and pass the current time to the synchronization stage.

For the time bucketing technique, *synchronization* is an essential part of the simulation. It allows the system to monitor the current simulation time of all LPs and hence determine the time for the next inter-LP communication message. LPs with the same lookahead value are allowed to continue without another round of simulation. For example, if two LPs share the same lookahead value in a three LP system, the only LP that requires rollback is the one with a different lookahead value. Although the time bucketing technique has a rollback mechanism, it is still categorized as conservative simulation. This is because the lookahead stage does not send event messages to other

LPs, it only tries to determine the *time* of the next external event rather than *distributing* the next external event. If the next external event time and lowest simulation current time across all LPs exceeds the simulation target time, the simulation is terminated.

Following the synchronization stage, *simulation* is carried out to update the system state and generate external events. Unlike the time warp technique, simulation can only proceed to the time of the next inter-LPs message time. This greatly reduces the parallel capacity of event execution, due to the global time window. After the simulation stage, the communication stage takes on the duty of updating system states via event messages.

### 3.3.3 Deadlock Avoidance Simulation

Similar to the time bucketing technique, the deadlock avoidance algorithm requires establishing a boundary to limit the simulation process in order to maintain the causality rule. However, the boundaries in the deadlock avoidance technique are local boundaries which do not share the same value unlike the time bucketing technique. This allows the simulator to better adapt the boundary fluidly for different circuit types.

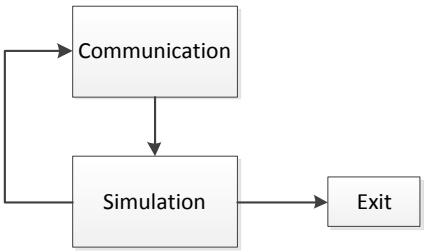



Figure 54 Simulation processes in the deadlock avoidance technique

The implementation of deadlock avoidance simulation consists of only two stages, communication and simulation. It is a fully distributed simulation system. The communication messages contain all the information that is required by data synchronization. Two types of messages flow within the communication system, *event* message and *null* messages. The event messages update the system states and null messages notify the LPs of boundary information. The communication stage updates the system state as well as creating a correct boundary for a LP to continue event processing. However, this is not carried out after processing each event, instead, the null messages after computing all pending events for a discrete time, which saves time as well as reduces communication traffic. [127]

Similar to the time bucketing technique, the simulation stage will only process the events up to a time boundary. In the time bucketing technique, this is the global lookahead value (GLV). In the deadlock avoidance technique, it is the local boundary that defines the furthest timestamp that can be safely processed locally.

### 3.3.4 Test Results

In order to decide which simulation is to be used, a quantitative set of tests across these three algorithms is presented. *First* of all, the speed of overall simulation is a critical point to examine. *Secondly*, the scalability of a simulation technique is another important reference to whether a technique is suitable for SpiNNaker platform. *Finally*, running on different circuit structures may also reveal pros and cons of a simulation technique. Simulation will be carried out on three different circuits and three sets of parameters will be examined and compared.

| | Gate Count | Aspect of Interest | Core Range |
|---|---|---|---|
| Ring Oscillator | 180 | Highly Parallel | 1 ~ 180 |
| 16-bit LFSR | 180 | Highly Sequential | 1 ~ 180 |
| DES | 5504 | Mixture | 1 ~ 180 |

Figure 55 Test circuit parameters

The purpose of these tests is to compare and choose the best simulation technique that will have the best performance when ported to a communication efficient SpiNNaker platform. As MPI on parallel cluster Iridis is slow in comparison to SpiNNaker communication (see section 5.2.3 for details), the tests carried out in this section will show the difference in performance when the MPI communication time is removed from the results. This is done by counting the time taken for each individual MPI command during a simulation. Whether the time in different cores is synchronized across all cores and the maximum precision of MPI can be extracted using MPI profiling functions. On Iridis platform, the maximum precision is 1 μs and the clocks can be synchronized.

The communication cost in real time is the only factor that can be scaled down after porting the design onto the SpiNNaker platform. It is very difficult to measure the

communication cost in real time, because the communication time might overlap with each other. The sum of the communication time in each process overestimates the real communication cost. As a result, a program is developed to detect the overlapping of communication time, and returns the estimated communication cost in terms of real time for each simulation.

The raw communication time consists of a pair of wall clock time values, with a precision of 1 µs. The difference between them is the time spent in processing the communication function. Among the three techniques, only point to point communication cost is counted as the scalable time. The synchronised broadcast and collection communication cost in time bucketing technique is not counted as suitable communication time, as the communication infrastructure that the SpiNNaker has does not support this type of communication function.

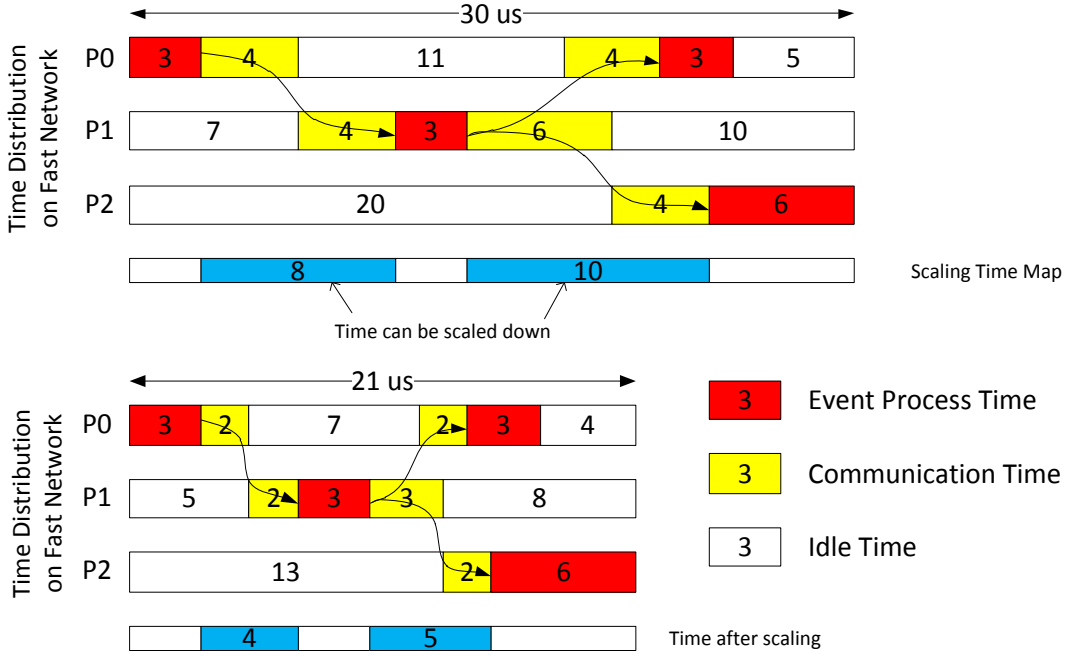



Figure 56 Communication time scale example

The raw time data provides a reference time that splits the simulation time into different states. The time is mapped to a single time map which indicates the time periods that can be scaled down using the communication time scale down factor of 34 times (see 5.2.3 for how to determine this factor) to reflect the predicted performance after porting

the system onto the SpiNNaker platform. In the case of Figure 56, assume the new communication cost is half as the old one. The corrected overall simulation time can be predicted by halving the time in scalable time map and remove it from the original overall time 30 μs, which is 21 μs.

The InfiniBand network [128] uses switched fabric topology to link up the computing nodes. The computing nodes are connected to one or more network switches via a serial connection. The switches will establish the fastest route to another target node without a well-defined structure.

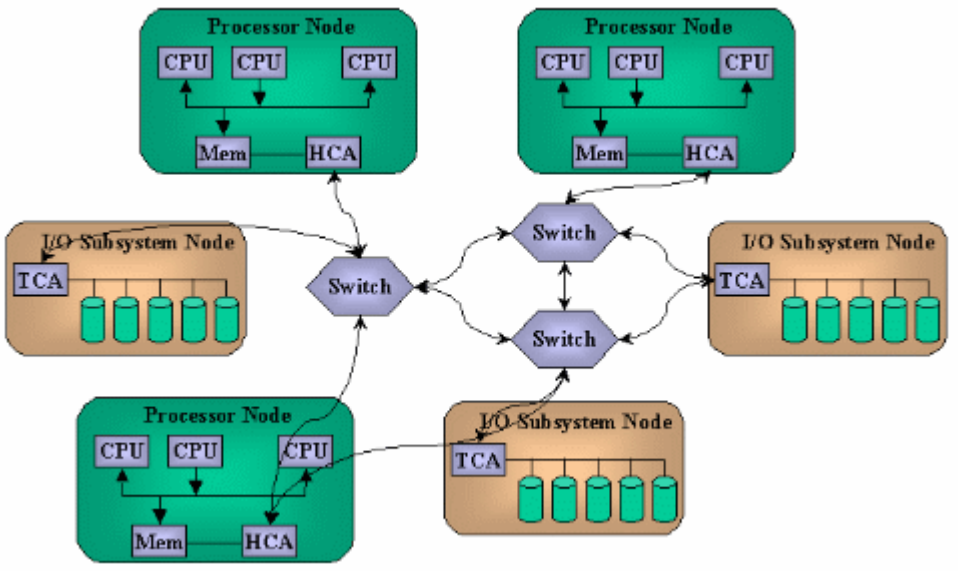



Figure 57 System Area Network based on the InfiniBand Architecture [128]

The three test circuits will be an array of 60 ring oscillators, a 16-bit LFSR random number generator and a DES encryption and decryption circuit.

### *Array of Ring Oscillators*

The array of ring oscillators will consistently generate 60 events in every discrete time of a simulation. These events can be processed in parallel and hence show how effective the system is when dealing with a circuit that does not require any synchronization during a simulation. One of the features of an array of ring oscillators test circuit is the option of parallel computation without communication, when the number of LPs is less than the number of ring oscillators. Each LP can carry out simulation individually

without notifying other LPs, as there is no connections exist between different sub-circuits.

The overall execution time of these three algorithms is shown in Figure 58. For each ring oscillator in the array, the clock rate is 16.67MHz and simulation end time 20μs. As a result, each oscillator generates $2\times10^4$ events throughout the simulation and the *total* event count sums to $1.2\times10^6$ events.

The performance results in Figure 58 are produced on the Iridis parallel cluster platform. The vertical axis of Figure 58 is the overall simulation wall clock time. The horizontal axis shows the number of physical cores involved in simulation. As each LP is mapped to a separate physical core in the system, the use of LP and core is interchangeable.

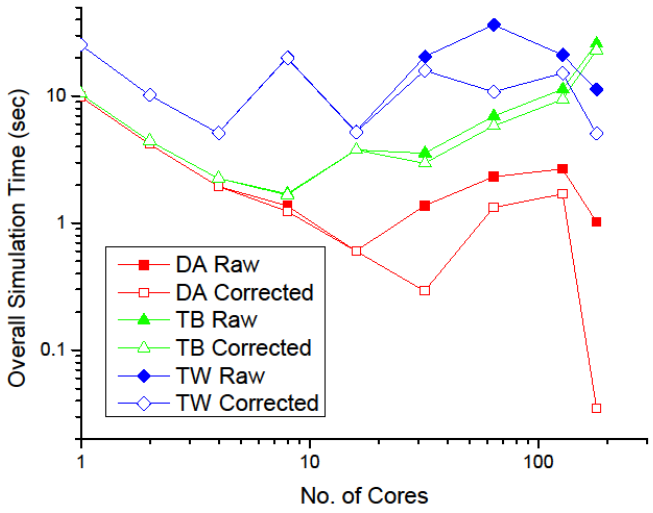



Figure 58 Overall simulation time for an array of ring oscillators

Based on the data in Figure 58, the deadlock avoidance technique has a clear advantage over the other two techniques. The large difference in overall simulation time between the time warp technique and the other techniques is caused by the extra incremental state saving action required by the time warp technique. Since the interest here is to find out the scalability of each technique, the percentage of performance improvement is the focus of this test. The time difference between the time bucketing technique and the

deadlock avoidance technique is caused by the increasing number of LPs getting involved in the synchronization stage, i.e. one process will have to wait for all other processes to reach the synchronization stage in order to perform lookahead value calculation.

Another observation is the differences in the responses of these techniques to the additional computational power. As stated before, the ring oscillator circuit only has 180 gates. The initial response of additional computational power is clear, all three technique use less simulation time than its sequential peer. However, when the number of cores gradually approaches the number of gates in a system, speedup gradually breaks down. The time bucketing technique performs worse than its performance in sequential mode, when the number of cores approaches the number of gates in the circuit under simulation (CuS). Although time warp technique manages to obtain performance gain, the magnitude is moderate in comparison to the gain obtains by the deadlock avoidance technique.

The last observation is the effect of corrected communication cost has in a simulation performance. All three techniques have corrected performance shown in the same graph. As can be seen from Figure 58, when the number of cores are above 32, the communication cost starts to have clear effect on the overall performance. By scaling down the communication time in a simulation, it is clear that deadlock avoidance technique has the greatest advantage. When the number of cores is equal to the number of gates in the CuS, most of the simulation time is communication time. This effect is clearer in the speedup graph shown in Figure 59.

The scalability of an algorithm can be derived from the overall simulation time result alone. If an algorithm scales well, the speedup increases proportionally to the additional computational power.

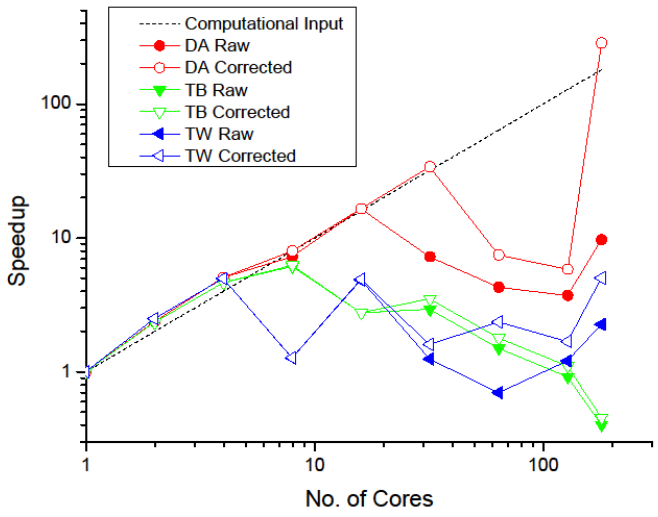



Figure 59 Speedup for algorithms when simulating an array of ring oscillators

Based on Figure 59, all three algorithms outperformed the linear speedup curve when the number of cores are below or equal to four. A speedup is defined as the simulation time in sequential mode divided by simulation time in parallel mode.

$$Speedup = \frac{T_{sequential}}{T_{parallel}}$$

A simulation running in sequential mode only uses one physical core to perform simulation. The simulation program is identical to its parallel peers. If a simulation takes 10 seconds to execute in sequential mode, and it only takes 4 seconds to complete in parallel mode, a speedup of 2.5 times has been obtained.

The simulation time is closely in track with the computational input. This is due to the reduction in event list size which decreases event access time. Hence this phenomenon cancels out the parallel overhead in a parallel simulation. When the simulation reaches the state where one core only holds a single gate, after scaling down the communication cost, the speedup obtained by deadlock avoidance technique is on track with the input of computational power. In other words, a linear speedup relationship is obtained by the deadlock avoidance technique after the communication cost scaling.

### *LFSR Circuit*

The second test circuit used is a 16-bit LFSR random number generator (see section 3.1.1 ). The reduced parallelism in the simulated circuit forces LPs to communicate with each other more often. With deadlock avoidance this means much more time is spent on null messages which establish temporal boundaries for individual LPs. In the time warp technique this means the chance of a rollback increases dramatically. In the time bucketing case, the additional communication may not have much effect on simulation speed, since the synchronization involved in simulation is already high. The speedup curve for simulating the 16-bit LFSR circuit is shown in Figure 60. The clock frequency for this LFSR circuit is 10MHz, i.e. a new random number is generated every 10ns. The simulation end time is 20μs, which generates 2000 16-bit random numbers.

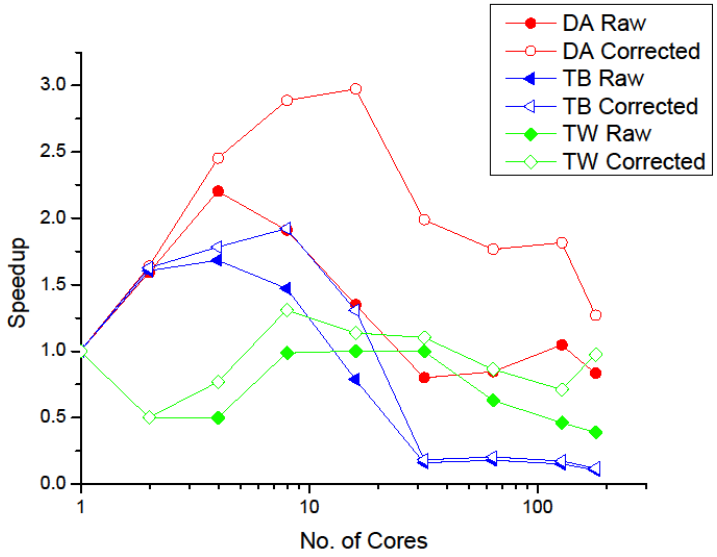



Figure 60 Simulation speedup for 16-bit LFSR

Scaling the communication time in this test made a huge difference for deadlock avoidance technique, but only moderate effect on the other two techniques. This shows that in a circuit with a highly sequential property. By eliminating communication time, the deadlock avoidance technique can outperform the other two techniques. Based on Figure 60, deadlock has a moderately better performance against the two other techniques. Its simulation speedup increases proportionally with available core count until it reaches 16 cores, which then drops as the number of cores continue to increase. The time warp technique shows a similar performance as a simulation using only one

core. The time bucketing technique as before has shown a bad scalability property, as the number of cores increases.

### *DES Circuit*

Finally, a DES circuit, which has 5503gates, is simulated using these three different algorithms. DES contains 16 rounds of encryption to generate a cipher text. Although DES is a highly computationally intensive circuit, it is a sequential processing block at its core. In this test, two DES blocks are connected to form an encryption and decryption pipeline. By using the same encryption key, the circuit output of these two chained DES blocks will be identical to the plain text that was fed into the system in the first place. The clock frequency of this simulation is 5MHz. The target simulation time for this circuit is 20 µs and the overall simulation generates 658,000 events. The simulation speedup is shown in Figure 61.

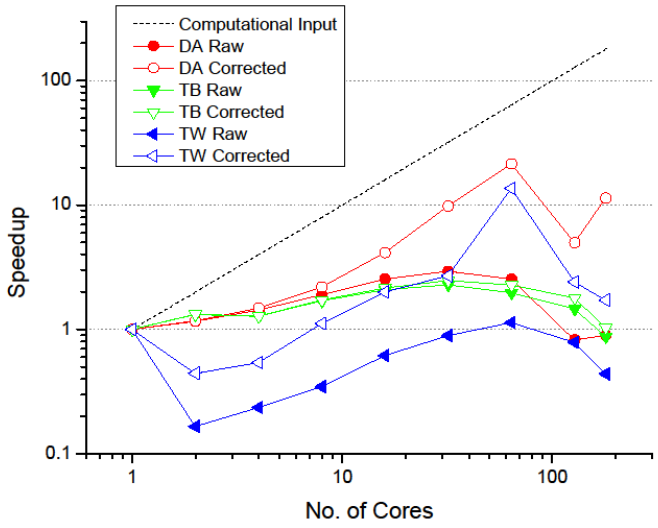



Figure 61 Simulation speedup for two DES blocks

From Figure 61, without scaling the communication cost, all three techniques fail to produce any performance gain when there are 180 cores. The scaling down of communication cost made a large difference in performance for both time warp and deadlock avoidance technique. This is because the synchronized communication in the time bucketing technique is not suitable for scaled down, as only the P2P event communication is scaled down. The deadlock avoidance technique is the winner again in this case, due to its light parallel control in comparison to time warp technique.

In summary, these techniques show a wide range of performance on the same simulation target sets. From every perspective of the three tests carried out on these techniques, the advantages of the deadlock avoidance technique make it a clear winner; despite the fact that other papers [20], [129] report that the time warp technique has a better performance over the conservative deadlock avoidance technique. This is due to the special properties implicit in in low level circuit simulation, which includes the strict causality rule and low lookahead value [130]. The strict causality rule limits the amount of events that can be processed by each LP. The low lookahead is due to the identical propagation delays associated with each logic gate within the test circuits.

The regular synchronization and pre-computation required by the time bucketing technique suppresses the event processing power of this technique. The technique shows promise when simulating the array of ring oscillators, but there is clearly a trend that shows the lack of scalability for this technique and the event processing efficiency is the lowest among the three algorithms.

The time warp technique, on the other hand, suffers from the same problems faced by the time bucketing technique. Although the low lookahead problem is overcome by the optimistic simulation algorithm, the strict causality rule punished the performance badly as shown in the 16-bit LFSR circuit. As the simulation needed to roll back the system state whenever a local LP violated the causality rule between the logic events, additional time is required to correct the faulty states that were previously updated. When the simulated circuit has a high sequential component, most of the optimistic calculation will have to be reversed and reprocessed.

Out of these three techniques, the deadlock avoidance technique shows a good overall performance, an excellent event processing efficiency and most importantly, good scalability in comparison to the other two techniques. As a result, this technique is chosen to be the base simulation algorithm for SDES simulator.

## 3.4 *Summary*

This chapter shows the preliminary performance test results and its related work.

- A set of **12 test scenarios** were created to test three aspects of the final SDES simulator, event simulation, communication, static and dynamic partitioning. The results are presented in chapter 5
- A **tool chain** that covers language parsing, function generation, function synthesis, and test bench import was created to support the SDES simulator.
- Based on preliminary implementation of the three techniques, time warp, time bucketing, and deadlock avoidance, it was shown that deadlock avoidance technique has the best scalability among all three tested scenarios. Therefore the **deadlock avoidance technique was chosen** to be the foundation of the final SDES simulator.

# Chapter 4 The SpiNNaker Discrete Event Simulator

## 4.1 *Basis and Goals*

The aim of the SpiNNaker Discrete Event Simulation (SDES) system is to address the issue of ever increasing simulation *problem size* with *scalable* computing power that matches the increase in problem size. Although a speedup in simulation time can never approach perfect scalability, the technique provides a way to significantly reduce the time required for performing a simulation.

As the size of a computing system increases, the cost of synchronization becomes a significant factor during a simulation. During synchronization, processes need to wait for other processes to complete their current work. This creates gaps in the simulation, which increase the overall execution time. A fully distributed simulation that requires *less synchronization* is therefore preferable. As discussed in section 2.2.2 and based on the preliminary tests of section 3.3.4 , the deadlock avoidance technique is most suited to be the basis for SDES. The implementation of this technique is discussed in section 4.4 .

However, as outlined in the preliminary work section, the deadlock avoidance simulation technique cannot fully utilize the computation power of the system. This is due to the fact that the workload migrates to different parts of the system and leaves certain LPs to rest and wait for further workload to arrive. The traditional way to tackle this is to move to optimistic simulation, but doing this creates two problems. One being the memory cost caused by the checkpointing required to perform rollback operations; the other is the synchronization. In this project, *dynamic load balancing (DLB)* is brought in to solve this particular problem. The idea is that DLB can maintain the advantage of conservative simulation, which preserves causality between events. As a

consequence, it does not need to save checkpoints on a regular basis, thus saving on memory. In addition, it smoothes out the workload among the LPs, which helps to reduce the overall simulation time. The technique chosen is the sender-initiated diffusional technique, discussed in section 2.3.2 The problems and implementation of a dynamic load balancing technique is discussed in section 4.6 .

The SpiNNaker hardware has been under development throughout the process of this project, so in order to continue with the project an emulation of the SpiNNaker hardware is required. However, the SpiNNaker system is a hugely complicated beast, with numerous capabilities. Creating a full-scale SpiNNaker emulator would be a massive task, and is unnecessary for achieving the goal of this project: designing a deterministic simulation system targeting distributed hardware with the ability to scale up the performance using the high core count of the SpiNNaker hardware. Therefore, a software emulation subset of the SpiNNaker system was built on an Iridis parallel cluster system, so as to emulate the environment for the purpose of this project, as described in section 4.2 .

## 4.2 *SpiNNaker Subset Emulator*

### 4.2.1 Emulation Target

The main purpose behind implementing this emulator is to replicate the necessary capabilities of the hardware structure in the SpiNNaker system. The emulator also provides the environment necessary for SDES to evaluate the platform's simulation performance, which in turn demonstrates the effect that can be brought about by the SpiNNaker hardware as a result of scaling down the relevant timing data. Finally, the emulator forms an intermediate layer between SDES and the SpiNNaker system. Future developments, and porting the simulation system to the SpiNNaker system can be simplified by removing this intermediate layer, without any major modification to the algorithm itself being needed.

### 4.2.2 Parts of SpiNNaker Not Modelled in the Emulator

As the emulator is simulation for a subset of SpiNNaker, there are certain elements featured in SpiNNaker that are assumed to be functioning when the SDES is loaded and

executed. Therefore, these elements, which are discussed in the section below, are not modelled in the subset emulator.

### *Boot Sequence*

The initialization of the SpiNNaker system includes the establishment of the connectivity map, live processor tree and routing mechanism, as well as loading a simulation problem. A description of the full process is described in section 2.1.2 . These processes are processors to perform execution on SpiNNaker. However, as the hardware system is not fixed, the software boot sequence evolves over time. In addition, the boot sequence does not have any effect on the actual simulation performance. As a result, it is ignored when designing the emulator.

Because the boot sequence is not implemented within the emulator, the assumption must be held that the boot sequence has been completed: each emulated core must be given and must retain all the knowledge of the SpiNNaker system which would have been generated by the boot sequence.

- **Layout of processor cores**

The structure of Iridis is such that it has 1008 nodes, with each node providing two 4-core processors. As a result, they can be divided naturally into a two-tier system, similar to the SpiNNaker system. However, because the two systems are very different, their connectivity must of necessity also differ. A discussion of how the virtual SpiNNaker cores are mapped to the Iridis cores is presented in section 4.3 . The implementation details are in section 4.2.4 .

- **Identification of processor cores**

An identification number is assigned to each processor during the boot sequence. This is done by passing a token using nearest neighbour packets in the SpiNNaker system. As the boot sequence is not modelled in the emulator, the emulator will assume that the identification has been assigned to each processor in the system, which leads to the *mapping problem*, as explained in section 4.3 .

- **Routing of P2P messages**

There are four types of packets in SpiNNaker (see section 2.1.1 ), and three of them need to be established at the initialization of simulation: Multicast (MC); point-to-point (P2P); and fixed route (FR). Two of them, P2P and FR, can be established during the boot sequence. Both of them are derived based on the hardware structure rather than the simulated problem. The FR provides a quick exit for any packet that needs to escape from the SpiNNaker. The MC packet is closely related to the simulated problem. The mapping of the problem graph defines the topology of the network. The MC packet is a function of the P2P table and the problem graph. However, the emulation platform does not have the visibility limitation that SpiNNaker has, this being discussed in the next part of this section. The P2P communication is the only packet modelled in the final SDES system.

- **Ready-loaded programs in monitor and slave processor cores**

The emulator is also ignoring the fact that programs must be loaded to each individual core through communication packets. In Iridis, this process is carried out by the operating system infrastructure, so the emulator assumes that the programs are in place and all routing tables initialized, ready for execution.

### *Data Visibility and I/O*

*Data visibility* in the SpiNNaker system is a massive problem, considering the size (~ 1 million cores) and the relatively few available external connections to the outside world (a handful of Ethernet ports). At the end of a simulation, not only the simulation results need to be transferred out of the SpiNNaker system, but also the performance data. This creates a vast amount of data that needs to be transferred at the end of a simulation.

Furthermore, each of the SpiNNaker nodes has only six external connections to its nearest neighbour in a toroidal structure. This poses a limitation on the *connectivity* between the SpiNNaker nodes. Packets sent to and from nodes at the side opposite an external connection have to travel a long distance, and must also travel through the crowded communication network in the SpiNNaker system. On a full-blown 1M core system, the maximum hop length $\sqrt{\frac{1000000}{18}} \div 2 = 117.8$, multiplying it by 0.1 μs per hop.

The maximum communication cost is around 12 µs. The number 18 is the number of cores per node, dividing the total number of cores by it gives the total number of nodes in the system. The maximum communication length is a half of the edge of the square matrix.

In this emulator, such data visibility and I/O limitations are ignored. Although they are crucial problems to be solved in the final implementation, the focus of this project is to produce an SDES that targets the SpiNNaker architecture and produces an evaluation of the simulation technique, which in itself has little to do with the limitations faced by the final implementation. In addition, the implementation tries to emulate the SpiNNaker platform to be as true as possible, so that if anything goes wrong, the problem can be reflected on the underlying causes.

#### *Memory Allocation*

The memory on the SpiNNaker system is very limited: 64Kbytes data memory and 32Kbytes program memory per core (for details see section 2.1 ) – which is clearly not enough for a large complex problem. When solving a complex problem, the program must be able to handle not only this slow but large SDRAM (128 MB), at the same time competing with other peer on-chip cores. The reading and writing access time of internal register is 3x and 1.5x faster than they are for the SDRAM. This is clearly a complicated issue for the emulator. As the emulated cores are spread over a fully distributed cluster network, the memory space for each process is protected from its peer emulated cores. If the emulator tries to replicate the same behaviour as the SpiNNaker cores, a virtual memory space that is distributed among the cluster network must be implemented. This memory space must be synchronized across the emulated cores belonging to an emulated node. This limitation is not taken into account when building the emulator.

### 4.2.3 Parts of SpiNNaker Modelled in the Emulator

#### *Interrupts*

Interrupts are the driving force behind the SpiNNaker software system. Applications for the SpiNNaker system are modelled based on the assumption that interrupts will trigger the processing during a simulation. Each SpiNNaker cores has its own interrupt

controller. There are 32 different interrupt sources, ranging from RAM access to communication control, as shown in Table 5. However, in this project, the only type of interrupt that really matters to the emulation is the communication control interrupt.

| # | Name Function | |
|---|---|---|
| 0 | Watchdog | Watchdog timer interrupt |
| 1 | Software int | used only for local software interrupt generation |
| 2 | Comms Rx | the debug communications receiver interrupt |
| 3 | Comms Tx | the debug communications transmitter interrupt |
| 4 | Timer 1 | Local counter/timer interrupt 1 |
| 5 | Timer 2 | Local counter/timer interrupt 2 |
| 6 | CC Rx ready | Local comms controller packet received |
| 7 | CC Rx parity error | Local comms controller received packet parity error |
| 8 | CC Rx framing error | Local comms controller received packet framing error |
| 9 | CC Tx full | Local comms controller transmit buffer full |
| 10 | CC Tx overflow | Local comms controller transmit buffer overflow |
| 11 | CC Tx empty | Local comms controller transmit buffer empty |
| 12 | DMA done | Local DMA controller transfer complete |
| 13 | DMA error | Local DMA controller error |
| 14 | DMA timeout | Local DMA controller transfer timed out |
| 15 | Router diagnostics | Router diagnostic counter event has occurred |
| 16 | Router dump | Router packet dumped - indicates failed delivery |
| 17 | Router error | Router error - packet parity |
| 18 | Sys Ctl int | System Controller interrupt bit set for this processor |
| 19 | Ethernet Tx | Ethernet transmit frame interrupt |
| 20 | Ethernet Rx | Ethernet receive frame interrupt |
| 21 | Ethernet PHY | Ethernet PHY/external interrupt |
| 22 | Slow Timer | System-wide slow (nominally 32 KHz) timer interrupt |
| 23 | CC Tx not full | Local comms controller can accept new Tx packet |
| 24 | CC MC Rx int | Local comms controller multicast packet received |
| 25 | CC P2P Rx int | Local comms controller point-to-point packet received |
| 26 | CC NN Rx int | Local comms controller nearest neighbour packet received |
| 27 | CC FR Rx int | Local comms controller fixed route packet received |
| 28 | GPIO[0] | Signal on GPIO[0] |
| 29 | GPIO[1] | Signal on GPIO[1] |
| 30 | GPIO[6] | Signal on GPIO[6] |
| 31 | GPIO[7] | Signal on GPIO[7] |

Table 5 Interrupt Sources

A simplified version of an interrupt controller is implemented, one which deals only with a P2P receiving interrupts. The justification for only using P2P packets is given in the next part of this section. The controller dedicates a 32-bit address register, a 32-bit data register and two control flags for interfacing are dedicated. In *receiving* mode, if a packet is waiting to be read, the receive flag will be raised and the registers will be filled

with the received packet address and payload. The simulator can read the information in both registers after the receive flag is up. In *sending* mode, once both address and data registers are filled with information, the contents in the registers will be automatically sent via an MPI message. Both the source and target monitor addresses must be written in the 32-bit address register, and the 32-bit payload needs to be written in the data register. After both these registers are written, the information inside both registers will be formatted as an MPI message and sent to the destination monitor process specified in the address register. In this project, all messages are sent in the form of this emulated 32-bit packet format. The address data in the original SpiNNaker packet is transferred using the flag of the MPI message; the 32-bit data payload of the original SpiNNaker packet is then sent using four 8-bit character bytes (See Figure 62).

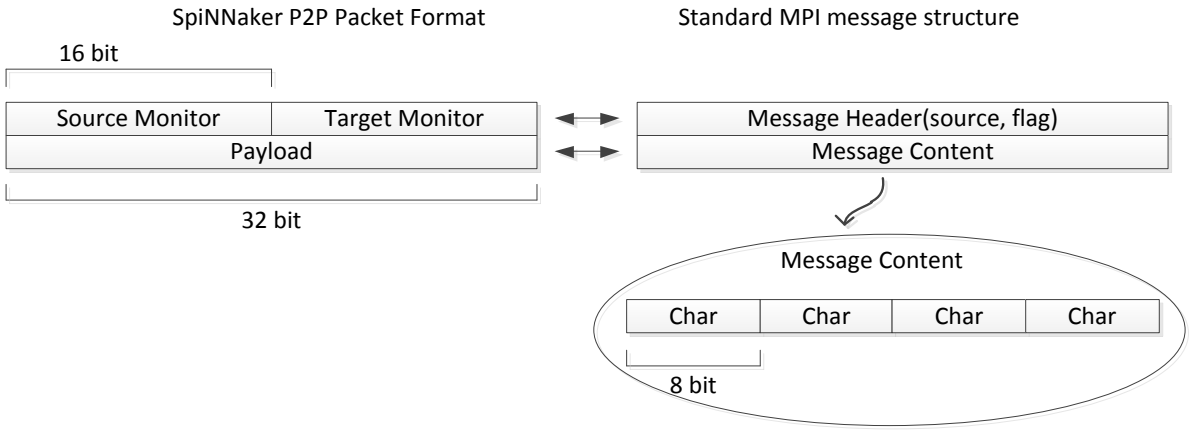



Figure 62 SpiNNaker packet representation in MPI messages

The emulation of this packet-passing operation is implemented using MPI functions that perform the MPI message-passing work between the Iridis cores. In a receiving operation, an incoming packet must be received before raising the receiving flag. In addition, if there is no incoming packet available, this operation must not prevent the program from continuing to operate. *MPI_Iprobe()* is a non-blocking MPI message probing function, which allows incoming MPI messages to be checked for, but without actually receiving them. The arguments returned from the *MPI_Iprobe()* function will indicate whether or not an MPI message is available to transfer; if so, the source and flag of the message will be returned. When *MPI_Iprobe()* shows that a message is waiting to be transferred, the receiving operation receives this message using a blocking *MPI_Recv()* function to store the content of the message, which is four 8-bit character

data. After receiving a message, the receiving flag is up and the content of the dedicated 32-bit data register is ready to be read by the simulator. After a reading operation of the content of the data register, the receiving flag is cleared without clearing the content in the data register, so the data register can be read regardless of the status of the receiving flag.

In a sending operation, the address and data registers are available for access when the *send ready* flag is up. Once these registers are filled with fresh data, the *send ready* flag is pulled down and the contents of both registers are formed as an MPI message. This message will be sent to the designated target monitor core using an *MPI_Send()* function. After the send operation is complete, the flag is raised up again, indicating that the communication is ready for the next packet.

### *Packets*

In the SpiNNaker system, there are 4 types of packet: Multicast (MC); Point-to-Point (P2P); Nearest-Neighbour (NN); and Fixed Route (FR) (See 2.1.1 for details). The FR packet type is not intended to be involved in general packet-passing operations. The NN packet type is designed for communication between physically adjacent processors, and is therefore not a candidate for long distance packet-passing operations. This leaves only two options, the MC and P2P packet types.

Initially, the **MC packet** was used as the packet-passing mechanism. The MC packet type has two major **advantages**. *Firstly*, an MC packet can be routed directly between different processes, as well as monitor-to-monitor, monitor-to-slave, and slave-to-slave. A detailed description of the structure of the process layout can be found in section 4.3 . This reduces the cost of communication; a packet does not need to be processed at the monitor process level in order to arrive at slave processes. *Secondly*, an MC packet can be replicated automatically through the communication network without any interference from the simulation program. In other words, an MC packet can reach multiple targets simply by sending the packet once at the initiation of the process. However, the MC packet-passing mechanism is incompatible with dynamic load

balancing, where the mapping between the LPs and the components is changed on-the-fly in a simulation.

The MC packet-passing mechanism relies on the MC routing table to transmit the packets to appropriate communication links. The routing table is generated according to the mapping between the LPs and the simulated problem. If the mapping is changed, the routing table must be changed correspondingly. There are two major **problems** associated with altering the routing table on-the-fly. If the mapping of a component is changed when going from one process to another, the routing table in all the SpiNNaker nodes along the old path must be updated. In addition, new routing entries must be established in all the nodes along the path between the source LP and new target LP. To make matters even worse, a further **problem** is associated with updating the old routing path. A single routing entry in a routing table is capable of routing a packet to multiple targets, so when a component is moved from one process to another, the old path may still be valid for another component in the same process. An example will help to clarify these problems, as shown in the following figure.

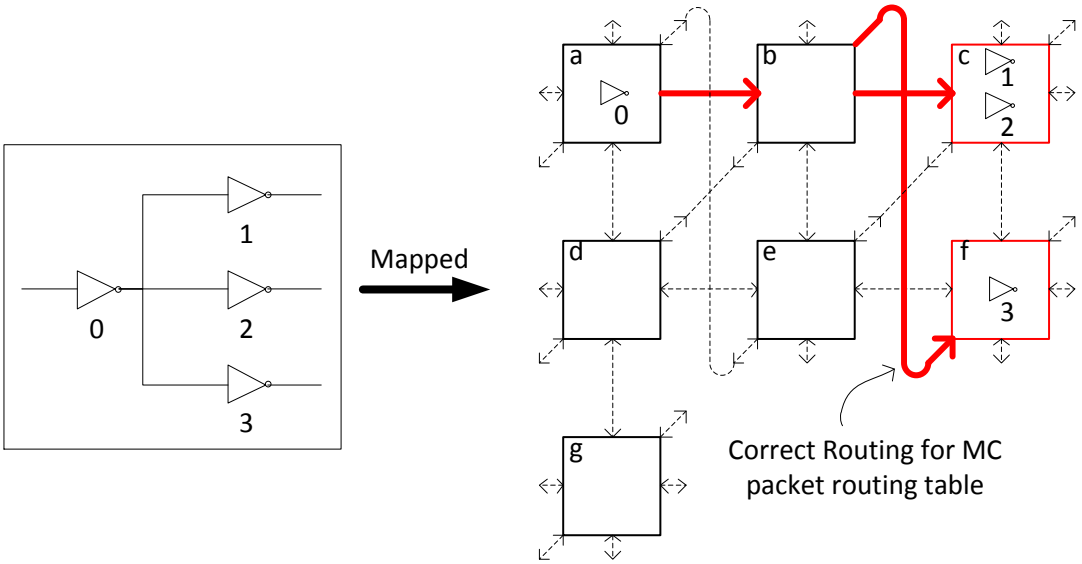



Figure 63 Example of mapping components to SpiNNaker nodes

In the example shown in Figure 63, four components are mapped to three nodes. Gates 1, 2, and 3 are the fan-out gates of gate 0. In this example, gates 1 and 2 are mapped to the same node *c*. The physical connections between these nodes are shown by the dotted

lines. For demonstration purposes, the links in the nodes are only partially connected. The MC routing of a packet sent by gate 0 is shown in bold red arrowed lines. Whenever a packet is generated by gate 0, it will be routed along the red lines. When an MC packet reaches its target, the router will route the packet to local cores, as highlighted by the red boxes.

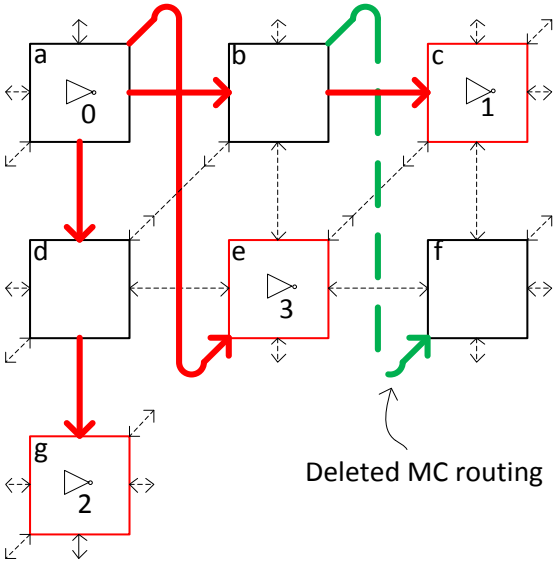



Figure 64 Correct routing after a change in component mapping

When changes occur in the mapping of the components, the routing in the MC tables must be adapted. Figure 64 shows two components have been moved in comparison to their original mapping in Figure 63. Components 2 and 3 are moved to nodes *g* and *e* respectively, and new connections need to be established in order to deliver the MC packets from gate 0 correctly.

Firstly, the routing change associated with component 3 is analyzed. In the old routing map, the shortest link between component 0 and component 3 is *a*-*b*-*f*. After the movement, the shortest link becomes *a*-*e*, so the routing map in nodes *a* and *b* must be updated. It is very difficult for the intermediate nodes to update the routing information, because the MC routing entry only holds the source ID as the routing key (as shown in Figure 4), and they only retain the relative position of where the target of a packet is.

In this case, node *b* can remove the routing entry for gate 0 from the MC routing table, which removes the old link between gate 0 and 3, but this will cause an error in routing, as part of the old link (*a*-*b*) is still valid for the link to gate 1 (*a*-*b*-*c*) in node *c*. This puts node *b* in a very difficult position, because whenever it tries to remove an entry from the MC routing table, it needs to confirm that no fan-out component of this entry exists along all subsequent routing tables throughout the entire system.

Although the update along the path of a link can be achieved with the help of the P2P packets, in two cases this presents a huge task. The *first case* is when a single link is spanning across a large number of nodes (65,536 in the worst case scenario, the maximum number of nodes supported by a P2P address), so the number of nodes that need to be updated for a single entry is a tall order. Another case is when the fan-outs of a component are located in a large number of nodes. In an extreme case, where a component has fan-out components located in all the nodes across the entire system, an iteration of the routing table update along the paths may make no difference at all. Think of the case for a routing table update for gate 2 after the component movement.

When updating the routing table for gate 2, the link between gate 0 and gate 1 validates the existence of the old link, and a brand new routing entry is created in both node *d* and *g*. In this update process, the routing tables in five nodes were updated simply for moving a single component around the system. This shows the scale of work required after updating the MC routing table when performing DLB.

Apart from the difficulties created when introducing DLB, the MC routing mechanism imposes other **limitations**. For instance, the size of the MC routing table is hardware limited to 1024 entries, and the routing table in each node must be configured to reflect the initial mapping of the simulated problem. The *first* limitation can be partially relaxed by optimising the routing keys and masking part of the address to allow entries to share the same piece of routing information, but this optimization can only go so far. If this happens, the adjacent nodes must take over the extra routing load. The *second* limitation requires a separate external program to handle the creation of routing table in

all the nodes. This requires an extra level of effort in order to allow a problem to be simulated on the SpiNNaker platform. Both of these limitations are non-critical problems, but as a result of the above discussions, the MC routing table generator was not used in the final design of the SDES system.

Because of these drawbacks of using a MC routing table, the MC routing table is not employed in SDES, instead, by using P2P packets only, the packet routing problem can be solved with complement from the simulator itself. As a P2P packet requires the node ID of a message, the simulator needs to provide component mapping information in order for the P2P packet to be routed.

The **P2P packet** is the only packet type used by the emulator, as there are benefits to using the P2P packets as the message intermediary between cores. The *first advantage* the P2P has is the initialized routing table in the boot stage without reference to the problem graph. As long as the physical structure of the system is left unchanged, the P2P routing table is always valid. The *second* advantage is the well-defined routing behaviour of a P2P packet. Unlike the MC packet, the P2P packet is not duplicated when travelling through the communication network. This provides a communication mechanism that is similar to generic computing clusters, and is thus an easy way of porting a generic problem onto the SpiNNaker system. This opens up the SpiNNaker system to many generic applications, and for this project it paves the way for employing the dynamic load balancing technique in a discrete event simulation.

These benefits are not without consequences. Due to the limitations of the address length in a packet, P2P packets can only reach the monitor core of a node. Further routing of a packet to a slave core must be controlled by the monitor core, which leads to a slowdown in data transfer speed. Experiment in section 5.2.3 shows a 2.8μs slow down for the additional monitor process. From the point of view of a slave core, it may never receive an interrupt from the router, because no packet can be delivered to a slave core without passing through the monitor core. As a result, an extra interrupt is created, one generated by the monitor process. This interrupt replaces the function of the

receiving flag interrupt, which is generated from the router. The detailed structure of the architecture of the system can be found in section 4.3 .

### 4.2.4 Extra Functions Modelled in the Emulator

The SpiNNaker system only provides a way to execute or process a *problem*, and its main objective is to create an open platform. As a result, the performance evaluation task is left to the users. In order to evaluate the performance of the simulator created, a full set of tools must be created.

#### *Instrumentation*

The SpiNNaker system is a fully distributed asynchronous system. The processor clock is frequency locked in the system, but not phase locked throughout the system. There is also a 32KHz clock in the system. This clock is sufficiently slow that phase lag is not a problem; it is used to indicate the passage of real time.

Performance measurements in this project are taken on the parallel cluster Iridis, and one of the aims of this emulator is to provide guidance as to how the simulator may perform when ported to the actual SpiNNaker system. This measurement is an important part of the project. The Iridis system is a generic parallel cluster, provides the emulator with a synchronized clock among different computing nodes. However, measuring the raw performance data is only the first step in achieving the goal of providing performance behaviour for this newly designed simulation system. The data requires significant post-processing to remove artefacts introduced by Iridis. This is described in detail in section 5.2.3 .

The major performance difference when running a program on the Iridis cluster and the SpiNNaker is the communication speed, and this leads us to an analysis of the difference, which is discussed later in section 5.2.3. After this analysis, it can be seen that the conversion rate between the communication time in the Iridis and that in the SpiNNaker is around a fixed factor of 34, i.e. the communication speed in SpiNNaker is 34x faster than it is in Iridis.

### *Inputs*

Section 3.1.2 explained how the SDES prepares the input data, including the digital circuit and the corresponding input events. There is still a gap between importing the circuit and then distributing it across the system. The emulator only provides a communication and performance measurement platform for the simulator. *Distribution* of the simulated problem, that is, the digital circuit simulation problem in this project, is required in order to initialize the system.

Although the SpiNNaker system does not have a central control, the emulation assigns one of the cores as the system overseer to manage the simulation environment. It performs the initialization and finalization of a simulation, but does not engage in event processing activities once the initialization is finished. The distribution of the SDES simulated problems is carried out in three phases, *system parameters*, *digital circuits* and *input events*. All the information is passed into the cores in the SpiNNaker packet format. The data involved is complex; the 32-bit packet payload is not sufficient, and the implementation of how the necessary data structures encoded and restored is discussed in section 4.4 .

In the *system parameters phase*, the identification, layout, and simulation technique configuration of the LPs are initialized. These are the fundamental parameters for the simulation to be carried out on this emulation platform.

Although the emulator does not perform the same boot sequence (see section 2.1.2 ) as in the SpiNNaker system, *unique IDs* of LPs are required in order to distinguish one from another. As the SDES is currently only running on Iridis, the identifications are already assigned by the MPI platform.

In practice, the SDES is running based on cluster computers, and the virtual layout of these computing cores is flexible. The SDES assigns an LP or a virtual monitor to each Iridis core. The SDES can define the number of processes that a node can hold, and the

simulator will then work out the number of nodes that are involved in the simulation process. Each node will have a monitor process and at least one slave process. In a regular cluster-based simulation, the *computation power* involved is proportional to the number of processors within the cluster. However, unlike other simulation systems, it is the number of slave processes within the SDES that defines the computation power involved during simulation. In a SpiNNaker node, there are two types of processes, one is monitor process and the other is slave process. The monitor process performs administration tasks for a node, while the slave processes are the ones that carry out computations. (See section 4.4.2 for detail)

The *layout assumption* made in this project is that SpiNNaker nodes form a square lattice. When the number of nodes does not form a perfect square, SDES will find the smallest square dimension and fill those rows with nodes first. SDES will fill and connect 7 nodes within a 3x3 matrix, as shown in Figure 65.

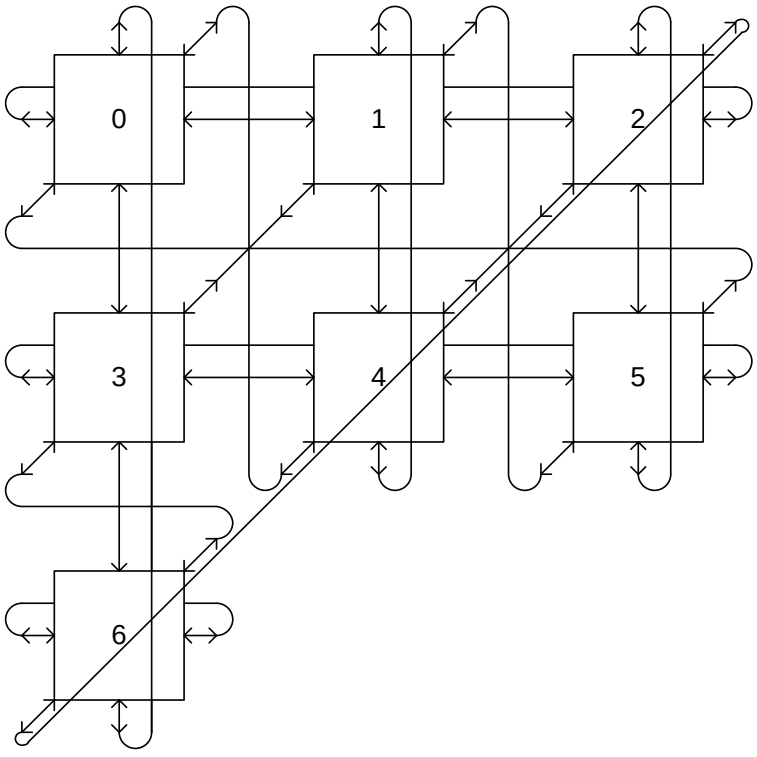



Figure 65 Partially filled square SpiNNaker node matrix

Mapping knowledge is not a key element in the communication system, but it is the key to dynamic load balancing. DLB employs a diffusion technique, and therefore needs to

know where the various neighbours are so as to operate the balancing mechanism. As a result, it is necessary to set up the mesh at the beginning of the simulation.

The *configuration* of the simulation system is distributed after the identification and layout of the system has been initialized, and includes both static and dynamic partitioning settings. A complete list of the parameters is shown in Table 6, and this data is passed on to each individual core within the SDES system.

| Simulation Option | Value Range | Details |
|---|---|---|
| Static partition enabler | On/off | |
| DLB enabler | On/off | |
| Static partitioning technique | 0/1/2 | 0: Block Partition 1: Random 2: Multi-way cut |
| Garbage collection interval | 1 ~ 10 | In seconds |
| Maximum workload difference % | 0~100 | Workload difference between LPs |
| Minimum workload to initiate DLB | 1K ~ 1M | Workload in an LP measured in number of events |

Table 6 Simulation configuration options

Once the system parameters phase is finished, the *digital circuit* distribution phase starts. In this stage, the circuit is loaded from the input circuits, and partitioned according to the configuration specified in the last phase of initialization. The circuit is imported from the file system using the overseer, which performs static partitioning to the whole circuit. Then the various components and the linkage between them are distributed to all the nodes by this overseer core. Once the transfer is complete, a further static partition is carried out at the node level. The monitor core splits the components belonging to the current node further according to the number of slave cores contained in the node. The full data distribution is explained in section 4.4 .

Following the digital circuit phase, the *input events* are distributed to the nodes. As the gates are distributed and assigned to a particular partition, the overseer is able to deliver the input events to the nodes that own the gates that are directly driven by the events. As

the packet used in SDES is a P2P packet, the packet can only arrive at the monitor core. The monitor core will then find the sub-partition of the gate and forward the event on to the correct slave core. When this initialization phases complete, a final message specifying the simulation end time is sent to all the nodes. This final message triggers the start of the SDES simulation.

### *Outputs*

As stated in section 4.2.2 , the I/O limitations of the SpiNNaker system are not modelled in the emulator. The simulation results and performance data have two ways of exiting the system; one is through the overseer core in the emulator system, and the other is through direct file access. The first technique is the standard way of exporting information. However, this process is painful, and often takes much longer than the simulation process itself. The second technique is a faster way of extracting all the information. Both monitor and slave cores produce their own output files, and the overseer process reads and sorts the data in these files and produce reports on the simulation results and performance data.

## 4.3 *SDES – the Architecture*

Before going into detail about how the simulation technique is implemented, this section introduces the distribution of data and how it is emulated on the parallel cluster platform Iridis.

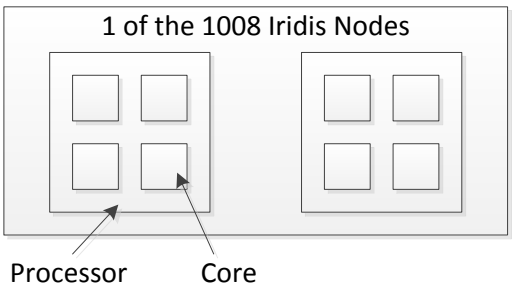



Figure 66 Iridis node layout

As introduced in section 3.3.4 , Iridis is a parallel cluster consisting of 1008 computing nodes, each node having two 4-core processors. All the nodes are connected to an InfiniBand network for inter-process communication. From the user's point of view, these processor cores can be seen as an array of processors having low latency connections to each other through a cloud of switches. Each of the SDES LPs will be

mapped to an Iridis core when performing the simulation, and although Iridis does not have a fixed structure, the SpiNNaker system has.

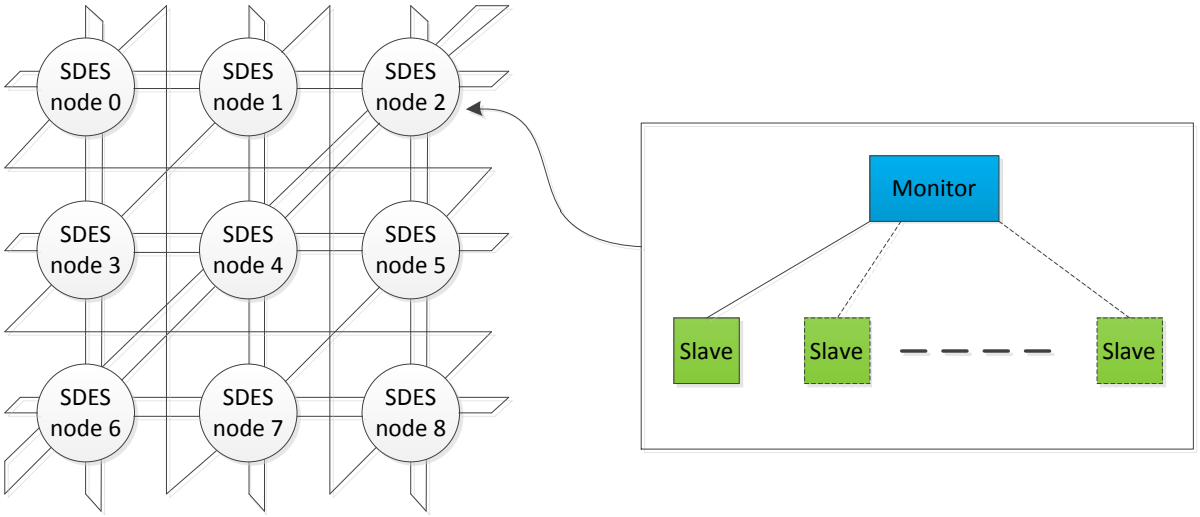



Figure 67 The SDES node structure

The structure of SDES can be represented as a mesh of processing nodes, where each node is equipped with 1 or 15 LPs plus a monitor process (Figure 67), a master-slave structure similar to works carried out in Grid computing [131][132][133]. As the number of LPs in a single node is represented as partitioning information using 4-bit data inside each gate, this limits the number of processors in a node to 16, however this can be increased to an arbitrary number by modifying the communication protocol (see section 4.5.3 for details) related to component transfer activities during the dynamic load balancing process. When the SDES LPs are mapped to the Iridis system, the hierarchical structure of SDES will be flattened.

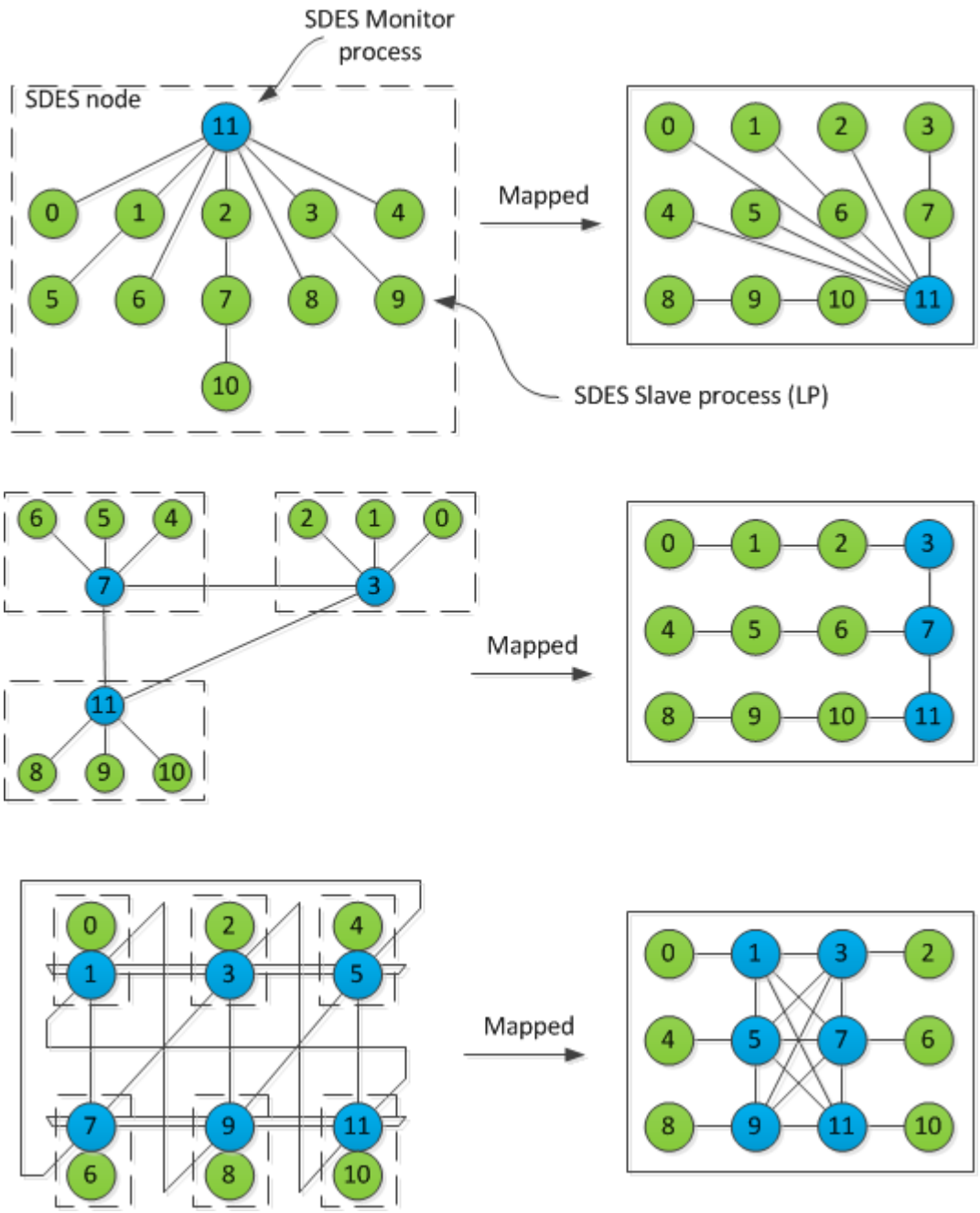



Figure 68 Flattened SDES structure after mapping to Iridis

The left hand side of Figure 68 is the designed structure of SDES and the connections between the nodes. The right hand side shows the distribution of the SDES processes that are mapped to Iridis cores. Although the hierarchical structure of the SDES processes has been flattened, the connectivity between the nodes is maintained.

When the system is configured, two parameters define the overall structure. The first parameter is the total number of cores to invoke in Iridis. The second parameter defines the number of cores in a virtual SDES node. Other parameters, such as the number of SDES nodes, the layout of the SDES nodes and the number of slave processes, can be derived from these two parameters. The SDES nodes follow the same system hierarchy as the SpiNNaker system: the same structure assumption is made that all nodes are filled in a square shape, i.e. three SDES nodes will fill a 2x2 square, and six SDES nodes will fill a 3x3 square. In addition, the process having the greatest ID value in an SDES node is defined as being the monitor process within the SDES node. Both concepts are illustrated in Figure 67. By replicating the hierarchical structure of the SpiNNaker system in the Iridis environment, the virtual SDES nodes can be directly mapped to its Iridis counterparts with the connectivity already taken care of.

## 4.4 *Implementation of the Deadlock Avoidance Technique*

A description of the raw implementation of the deadlock avoidance technique is given in section 2.2.2 , but this section focuses on the SDES implementation of it. The distribution of data, the packet-based communication system, boundary control in SDES and the functions of both monitor and slave processes are also explained in this section.

### 4.4.1 Data Landscape

The hierarchy within SpiNNaker naturally divides the computational work into monitor and slave processes. The monitor process controls the communication system due to the limitation of P2P packets. Slave processes, though, are focused on the event processing work, processing eligible events and generating response events which are sent to the monitor process for further communication routing.

The circuit digraph data is stored in every process in the system, within both monitor and slave. This reduces the DLB execution time, because detailed data about the circuit structure is already in every process. Only new partitioning data, rather than the entire description of a sub-circuit, is required to be transferred during DLB. During the initialization process, static partitioning is performed in both the overseer and monitor processes. The overseer process partitions the initial circuit according to the number of

nodes involved. This overseer-partitioned circuit is distributed to the monitor process in each node. Another level of partitioning is carried out by the monitor process to further split the local components in the overseer-partitioned circuit according to the number of slave processes involved in the local node. The SDES stores the overseer-partitioned and monitor-partitioned information as two separate variables in the component description.

Another essential part of a discrete event simulation, the events, is also distributed to each of the LPs in the SDES system. An LP can only be mapped to a slave process, as the monitor process does not engage in event processing activities. Each LP has a main eventlist and a log of executed events, which forms the simulation result. The monitor process also has an eventlist, but this is used for communication purposes only.

In addition to these, there is another set of data structure that are specifically designed to support the packet-based communication system. This is termed “message reconstructing”, and it is described in detail in section 4.5 .

The performance data collected throughout the simulation consists of time counter values for different simulation stages. This is stored in each LP along with the simulation results.

### 4.4.2 Processes

In the SDES system, there are three types of processes: overseer, monitor and slave. An SDES system has only one overseer process, this being responsible for initialization and finalization of a simulation, so once a simulation is initiated, the overseer process does not engage in the simulation until the end. The whole simulation is carried out by the latter two types of processes, and a more detailed division of tasks is explained in this section.

#### *Monitor Process*

The tasks performed by the monitor process can be split into three different types: a *reinforcing routing mechanism*; a *simulation boundary control*; and *workload monitoring*. These are the unique problems that the SDES system faces when implementing the conventional deadlock avoidance technique.

As outlined in section 4.2.3 , the type of communication packet used in the emulator is the P2P packet type. The P2P packet can only reach the monitor process of the target node specified in the packet address, so the packet then has to be routed through the monitor process in order to reach a slave process. As a result, the monitor process must handle all the incoming packets and direct them to their correct destinations. The details about incoming message handling process are discussed in message reconstructing part of section 4.5.3 .

However, unlike the MC packet, where the target slave core ID is specified in the packet address, the target slave core ID is hidden inside the payload of a packet. Since the payloads are encoded using a communication protocol (see section 4.5.3 ), the monitor process has to decode the entire message in order to figure out the destination of a message, which is represented in a series of packets.

The *second* task of a monitor process is related to event simulation. Although the monitor process does not perform event processing activities, it does share part of the burden that was devolved to the LPs in the conventional deadlock avoidance technique. Because the monitor process has knowledge of all the messages passing through to the local slave processes, it can derive a boundary (see section 2.2.2 ) that is shared by the local LPs. This additional layer of boundary control reduces the amount of null messages that have to be delivered if the boundary is set at the LP level. A comparison between conventional and SDES boundary layouts is shown in Figure 69.

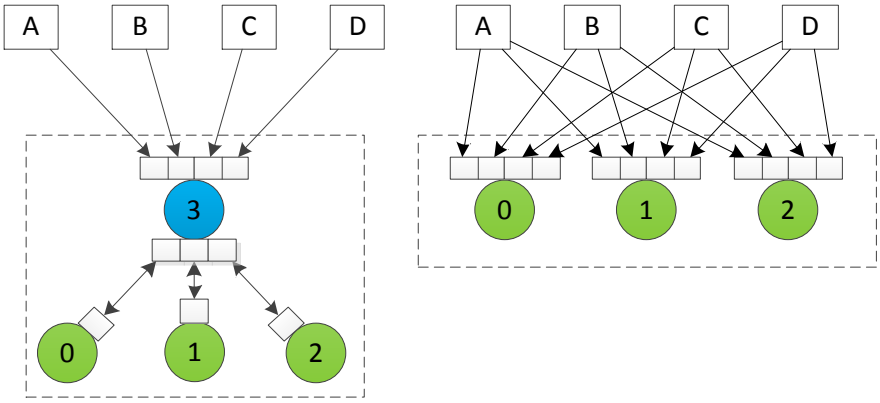



Figure 69 Boundary comparison

The case in Figure 69 assumes that all four external nodes try to establish access to LP 0, 1, and 2. In the SDES side on the left, the monitor process takes over the boundary control and only feeds a single boundary value to its LPs, which requires the system to manage 10 boundaries instead of the 12 in the conventional deadlock avoidance system. The lowest values amongst the 7 slots in the monitor process will become the boundary for the local SDES node. When it comes to establishing the null value for generating null messages for external SDES nodes, the lowest value in the local LP array, which has 3 values in Figure 69, is chosen to be the null value for the SDES node.

The boundary in a conventional deadlock avoidance technique is a fixed structure. However, the introduction of DLB in the SDES causes the system to vary the location of the boundaries in a system. As a result, the entire boundary system needs to be re-established by analysing the connectivity between different nodes. Further details about the impact that DLB on the boundary conditions is explained in section 4.6.3 .

The monitor process is not solely devoted to this task. If there is only one node in the entire SDES system, the boundary establishment is disabled in the monitor process, and the slave process or LP takes over the task.

The *third* task is to monitor the workload balance between local and adjacent SDES nodes in the lattice array of nodes. The monitor process periodically collects the

workload data from local LPs and broadcasts the sum of all local workloads, which is the aggregate number of events executed, to all six adjacent nodes. If the difference in workload between two adjacent nodes has passed a certain threshold, DLB is triggered and the two monitors cooperate with each other and begin the DLB process (see section 4.6.3 ).

### *Slave Process*

The slave process or LP in the SDES is very similar to an LP in the conventional deadlock avoidance technique, apart from the simplified boundary control and the additional DLB capability. The slave process receives events and null messages, processes all eligible events, and sends response events and local LP null messages to the monitor process. The deadlock avoidance technique background can be found in section 2.2.2 . A detailed description of the event processing mechanism is explained in the next section. Once a DLB is initiated, a slave process stops simulation and collects local components and associated events ready for transfer. When simulation stops, the slave processes dump both simulation results and performance data to the dedicated files, which are then further reorganized by the overseer process.

The timestamp in a null message is the current LP boundary value plus the *lookahead* value. The lookahead value is the minimum delay between any pair of IO (input-output) ports. The simulator analyses the delay between all possible combinations of IOs if the structure of the current circuit changes, which gives a minimum delay value that is the lookahead value of the current LP. The default delay for a gate in SDES is 1 ns. Without this evaluation of the lookahead value, the default lookahead value is a mere 1 ns, which limits the parallelism during processing.

The null message has two possible values. If there is no eligible event, which means it has a time stamp smaller than the local boundary, the null event will be the input boundary value plus the minimum delay value, which is 1 ns by default. If eligible events do exist in a local LP, the null value becomes the current simulation time plus the minimum delay value.

As described in the last section, the slave process has to take over the boundary control task when there is only one SDES node in the entire SDES system. Therefore, an optional boundary analysis system has to be in place to deal with this problem. It analyzes the monitor-partitioned information, and adds a slot in the boundary array for each fan-in LP detected. Similarly to the monitor process, the lowest boundary value in the array becomes the boundary for the local LP.

### 4.4.3 Event Processing Mechanism

At the core of a logic process (LP) is the event processing system, which evaluates the value asserted by a gate and produces new events, if any exist. There are three main components of a simulation: system state, event list and current time, as explained in chapter 2.1 . This section explains the difficulties faced when applying the discrete event simulation technique to discrete digital circuit simulation.

#### *System States*

A digital circuit consists of gates, ports and wires in between. They can be mapped naturally onto a directed graph data structure, but not without problems. The data structure that is used to store the digital circuit information is a directed graph structure [22]. The directed graph consists of *nodes* and *arcs*, both of which can hold data structures of their own. The arcs connect the nodes together, along with direction information. After mapping the circuit to a directed graph, the information on *nodes* are: gate ID, type, delay, partition, most significant bit (MSB), least significant bit (LSB) and value. On the *arcs*, there are only three components: wire delay, MSB and LSB. The wire delay can emulate the propagation delay on the wire if it is required in the simulation, but in SDES it is set to zero. The reason there are MSB and LSB on wires as well as gates is the possibility that only a part of the bus is connected between two gates; for example, only a single bit of an 8-bit value gate may be connected to a fan-out gate, then the MSB and LSB on the arc between these two gates will be different from the fan-in gate; any of the bits within the 8-bit gate value could be selected.

When mapping digital circuits to directed graphs, the first problem is the connectivity between gates. In logic simulation, the *order of input* wires matters. The digital circuits with which SDES will deal is capable of representing a multiple bit bus, which is a

group representation of wires. If the order of input wires changes, the result could then be different. Take the concatenation operator as an example; if the inputs are single bit wires, and the order is A-B-C-D, the concatenated output should be ABCD. If by accident the input sequence changes to A-C-D-B, the output will then be ACDB, as shown in Figure 70.

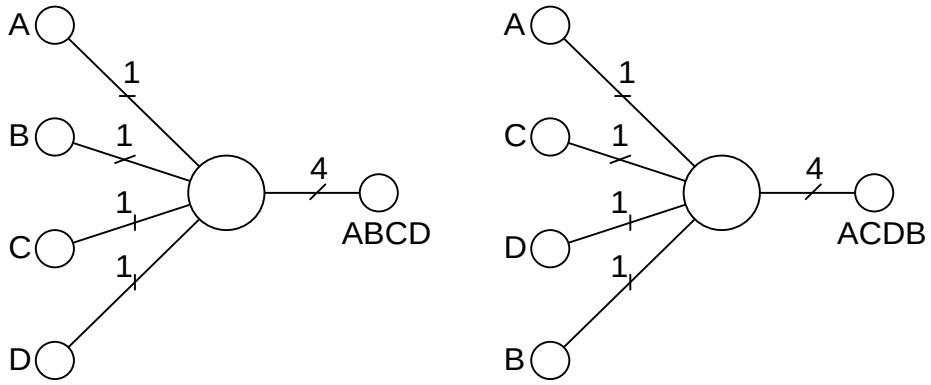



Figure 70 Concatenation of multiple wires

In order to ensure that the order of wires is correctly lined up, the wires have unique Ids as well as gates and ports. The order of inputs is determined by their ID values, and all inputs are sorted in an ascending order. As a result, provided the circuit has been constructed correctly at the start, it will stay consistent throughout the simulation process.

The second problem is the *state transition method*. There are four logic states for a gate in SDES, ‘0’, ‘1’, ‘X’ and ‘U’. When a gate switches from ‘0’ to ‘1’, the standard way of implementing this is to introduce an intermediate state ‘X’ between the initial and final states. This emulates logic transition from low to high. As shown in Figure 71, the logic level during the transition is seen as an unknown state in digital representation, because the details in this process are not of interest in terms of digital simulation.

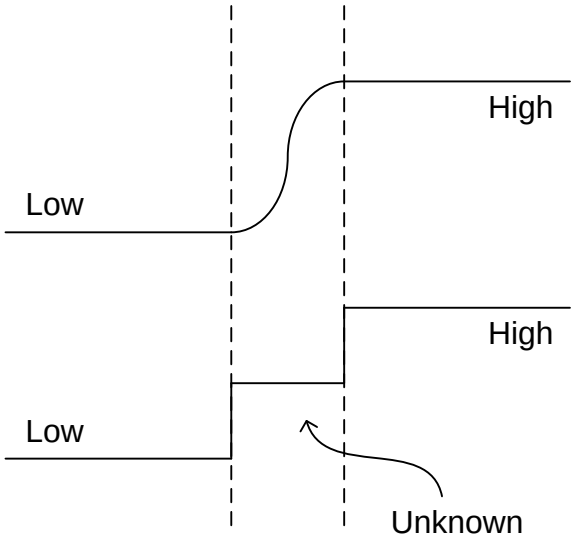



Figure 71 Logic transitions in the analogue domain and their digital translation

In the implementation phase, there is a hazard associated with the introduction of in introducing this intermediate state. In theory, the transition state should occur at an infinitely small time after the triggering event. However, in practice there is always a limitation on the resolution of time. If the resolution is high, a lot of time will be wasted in defining this transition operation. For instance, for a discrete time range of 0 to 100 ns, when the resolution is 1 point per ns (ppn), requires only 100 points to represent the range. When the resolution is increased to 100 ppn for the purpose of introducing this "infinitely small step", the number of points in the same discrete time range increases by 100 times. Only a few of these 100 points will be occupied by transition events. On the other hand, if the resolution is low, it might introduce a hazardous situation. The conventional setup has a wide gap between the simulation resolution and the minimum component delay in a system. However, when these two parameters come close to each other, a racing and unpredictable behaviour will arise.

In the case of an $\overline{SR}$ NAND latch circuit, which has a very short feedback loop as shown in Figure 72, very unpredictable simulation results can occur.

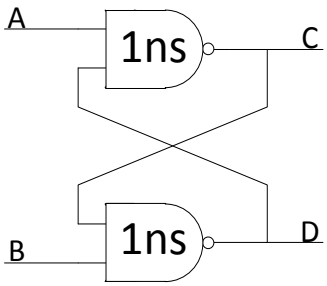



Figure 72 Latch circuit gate level representation

In the following graphs the states are changed, using both 0 → 1 and 0 → X → 1 transitions. The 0 → X → 1 transition caused a problem when the propagation delay was equal to the transition time. i.e. 0 → X → 1 takes 2 ns to complete the transition, and 0 → 1 takes 1 ns to complete the transition. If signal A is trying to switch from 0 to 1 at 100 ns, events "A = X @ 100 ns" and "A = 1 @ 101 ns" will be generated.

System States

| Time | A | B | C | D |
|---|---|---|---|---|
| 0 | U | U | U | U |
| 10 | 0 | U | U | U |
| 11 | 0 | U | 1 | U |
| 20 | 0 | 0 | 1 | U |
| 21 | 0 | 0 | 1 | 1 |
| 30 | X | 0 | 1 | 1 |
| 31 | 1 | X | X | 1 |
| 32 | 1 | 1 | 0 | X |
| 33 | 1 | 1 | X | 1 |
| 34 | 1 | 1 | 0 | X |

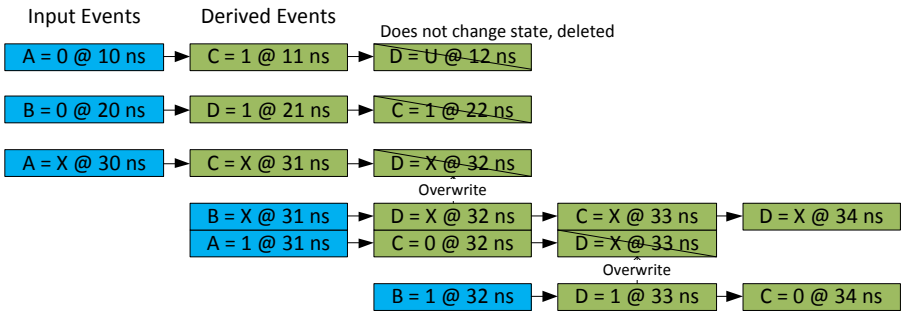



Figure 73 System state log for latch circuit

In Figure 73, there are six input events, which are shown highlighted in blue. Signal A and B are the input signals which follow the transition rule of 0 → X → 1. The transition time and the propagation delay are assumed to be the same. The green events are the derived events that are triggered by the input events. In this case the conflict starts when the switching speed is equal to the propagation delay of a logic gate. At 31 ns, the derived event of A=X@30 ns has the same timestamp as input event B=X@31 ns, so this unstable state caused gates C and D to repeatedly enter unstable states that were created by the 0 → X → 1 transition method. This shows that the intermediate unknown state could not even once solve the unpredictability of simulation, and would double the amount of total events of the overall simulation, which would slow down the

simulation speed. As a result, it was decided that a simple state transition would be implemented in the SDES.

### *Events*

In this last section, the problems faced when designing the data representation of a digital circuit are listed; and an investigation into the state of changing events has also been carried out in this section. There are three stages to processing an event: the collection of relevant data; the processing of the events; and possibly creating messages when dealing with a gate located on a different partition.

In SDES, the event ID is first extracted from an event and *searched* against the digraph iterators' ID. If the value in the event is equal to the value of the current gate then no action is required, and this input event will disappear from the simulation system. If the new event does represent a change in value, the simulator will pick out all the gates that to which this updated gate is connected. All of them will be re-evaluated with this new input event.

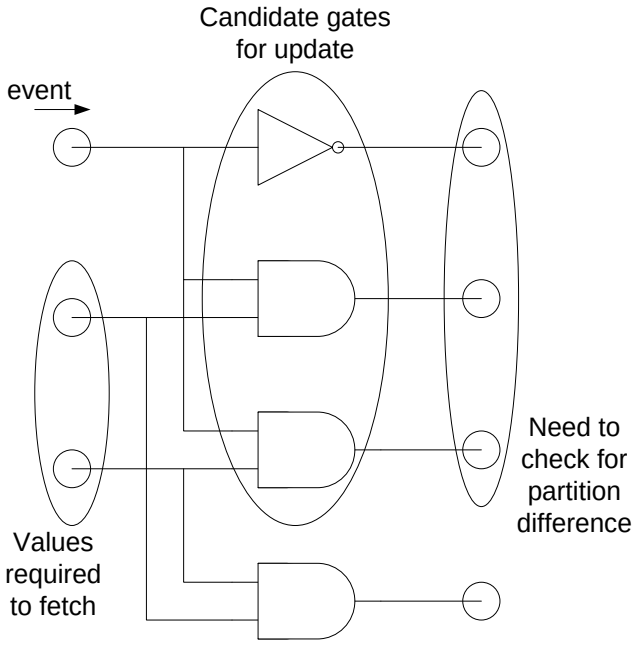



Figure 74 Value fetching operation

The relationship between gates when updating an event is shown in Figure 74. For each of these update candidate gates, the values of all fan-in gates will be obtained from the

circuit digraph, which will be *cropped* to the range stated in the connecting arc. Furthermore, they will be *stored* in a vector in ascending order of their arc ID values. Subsequent to the value fetching operation, the values will be *evaluated* according to the user-defined truth table of the gate. If a gate has multiple inputs, input values will be processed sequentially through the truth table. The output of the truth table is the new value of a new event. The time of this new event is calculated from the timestamp of the input event plus the delay in the evaluated gate. Combining this together, a new event is created. This new event will be fed back to the main event list. If any of these candidate gates is located on a foreign partition, this new event will be forwarded to the monitor via messages. The event processing flow is shown in Figure 75.

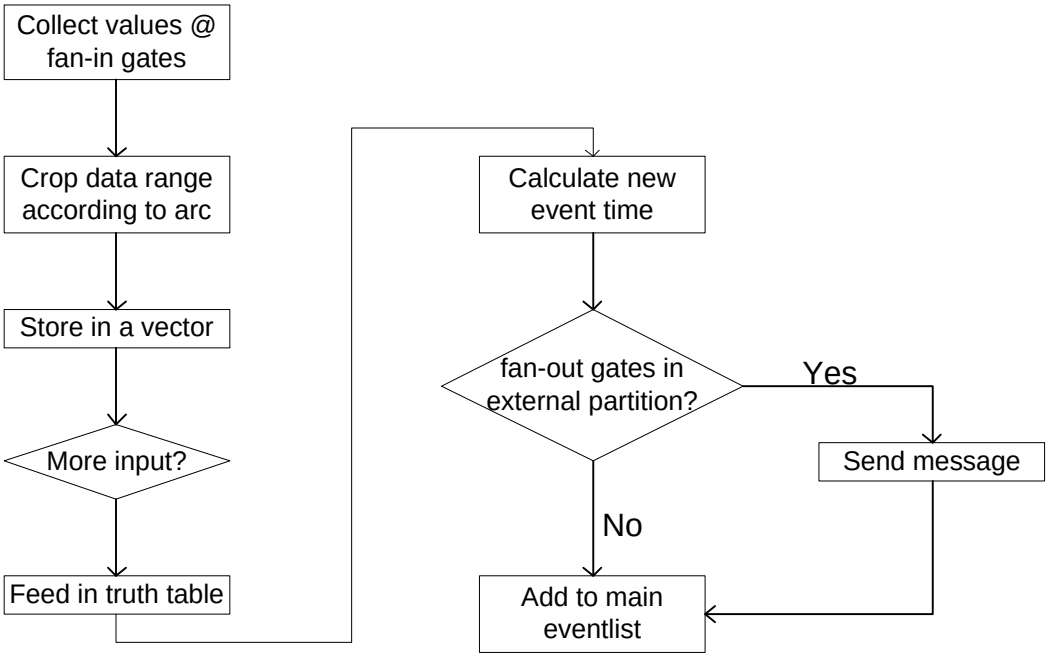



Figure 75 Gate evaluation operation

## 4.5 *Communication Platform*

In order to emulate the communication environment in SpiNNaker, a conversion between simulation messages and SpiNNaker packets is required. This conversion enables the simulation to be carried out on the SpiNNaker platform.

### 4.5.1 SpiNNaker Communication Model

The SpiNNaker system has two layers of processes – monitor and slave. The task of communication is therefore divided into two types. The monitor process controls the

routing of messages, but it is not involved in the simulation. The slave processes do not deal with the routing of messages; it only processes the inputs, and may respond to the input, depending on the simulation. The messages sent out by slave processes are routed correctly by the monitor processes.

The slaves are not allowed to communicate with other on-node or off-node slaves under any conditions, so all communication messages are routed via the monitor process. This greatly simplifies the work of slave processes, and reduces the task of synchronizing data throughout the system. If a slave process needs to work out the route of a message, this will absorb the computation power of slaves.

In addition to the limitations imposed on the tasks of monitor and slave processes, the size of a communication message is limited to 32-bits, which is very different from any other parallel platform. This is due to the nature of the SpiNNaker system, which is designed to deal with neural networks.

In a neural network, the connectivity itself tells most of the story. The spikes, which are the transmitted packets in a neural network, fire a sequence of neurons. A spike does not hold any data, but SpiNNaker was designed to be able to carry an optional 32-bit payload on each packet, and this 32-bit payload is the basis of communication in SDES.

### 4.5.2 Initialization Stage

Although the emulator does not perform the same boot sequence in the SpiNNaker system, unique IDs are required in order to distinguish between LPs. The structure of the SpiNNaker system is a matrix of nodes, and the *unique ID* helps the LP to find its position within the matrix. This ID is generated at the start of the program. The node will send out a single token passing the mesh of nodes sequentially, and the number assigned with the token will count up after passing each node. A node that already has an ID will pass the token to the node in the next connection down. If the token value is equal to the size of the mesh, then the ID assignment is complete.

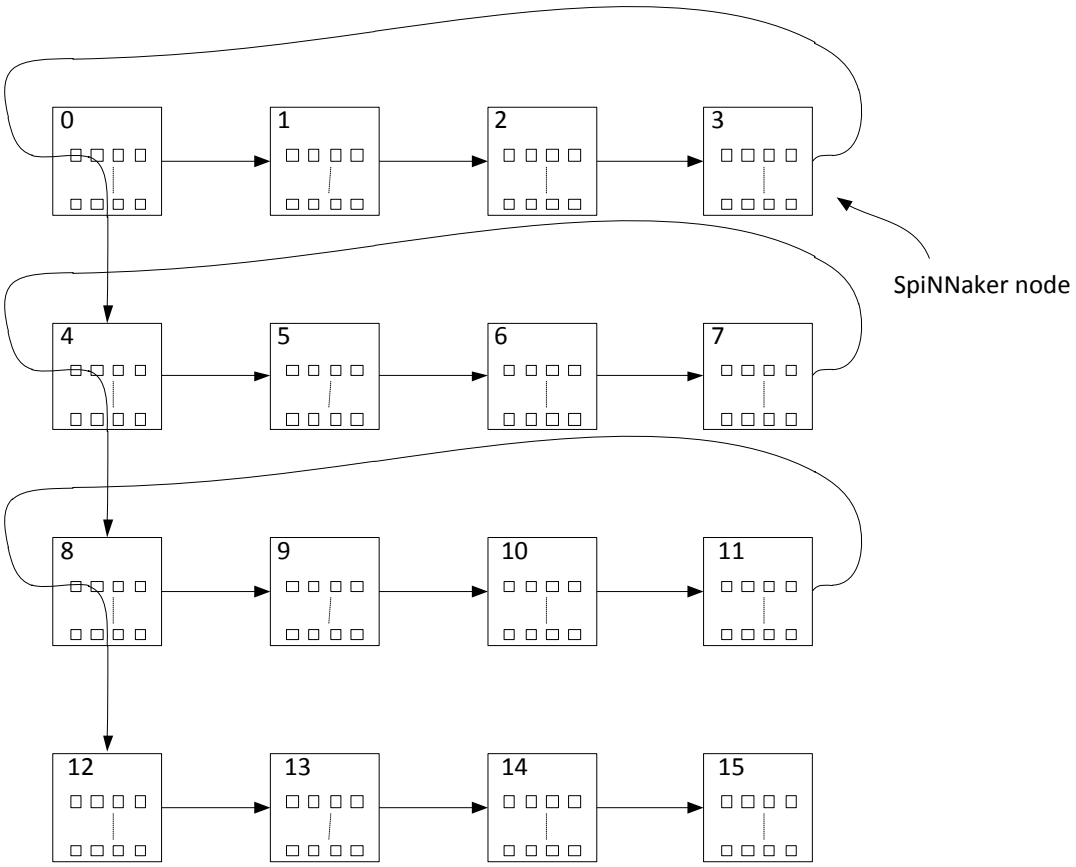



Figure 76 ID assignment sequence

In practice, SDES is running on a conventional cluster. There is no inherent SpiNNaker structure within the system. As a result, SDES can define the number of processes that a node can hold, and the simulator will work out the number of nodes that are involved in the simulation process. Each virtual node will have a monitor process and at least one slave process. In a regular cluster based simulation, the *computation power* involved is proportional to the number of processors within the cluster. However, unlike other simulation systems, the number of slave processes within the SDES defines the computation power involved during simulation, because the monitor processes only deal with communication packets and do not engage in the actual simulation directly.

The *layout assumption* made in this project is that SpiNNaker nodes form a square lattice. When the number of nodes does not form a perfect square, SDES will find the

smallest square dimension and fill the rows with nodes first. SDES will fill and connect 7 nodes within a 3x3 matrix, as shown in Figure 77.

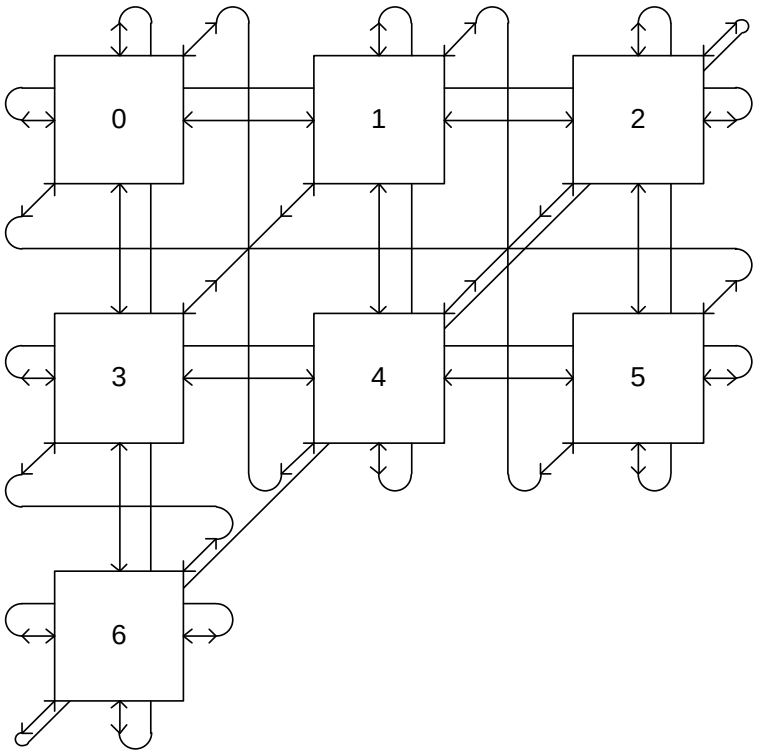



Figure 77 Partially-filled square SpiNNaker node matrix

Position knowledge is not a key element in the communication system, but it is the key to dynamic load balancing. DLB employs a diffusion technique, which needs to know where the neighbours are so as to operate the balancing mechanism. As a result, it is necessary to set up the mesh at the beginning of the simulation.

### 4.5.3 Simulation Stage

During simulation, the communication system is mainly focused on two things: *routing* of messages; and message *decomposition and reconstructing*. There are four types of packets in the SpiNNaker system: multicast, P2P, NN, and FR packets (details can be found in Appendix C). All of them are capable of carrying a 32-bit payload, but the only type of packet that is used in this project is the P2P packet.

### *Message Routing Mechanism*

P2P routing is initialized at the start of the SDES execution, and as a result the SDES does not need to initialize the routing of these messages. In practice, the P2P packet is replaced by MPI messages, but they can only carry 32-bit data in each message.

In an SDES simulation, *event messages* regularly update the circuit states across processes. Whenever an event message is generated, the target of this event needs to be extracted from the partition information within the fan-out gate. In order to speed up the routing process, this connectivity information is extracted to a map, which uses the ID of a gate as an index, and all the partition information of fan-out gates are stored as the data. During the simulation, whenever the monitor detects the arrival of an event, the event will be routed to all the nodes listed in the map, which saves time looking up individual fan-out component partitions in the large circuit digraph. This routing map is shown in Figure 78.

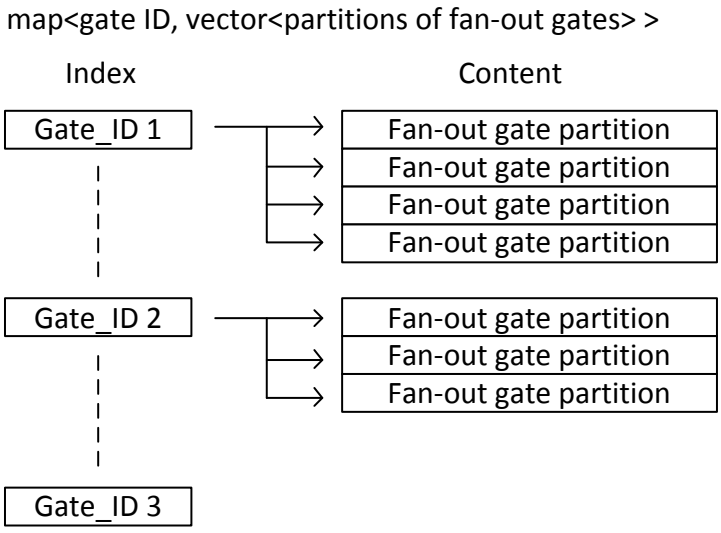



Figure 78 Event routing map

In addition to event messages, there are also *null messages*, which require routing on a regular basis. The routing map for null messages is much simpler and smaller than its event routing counterpart. A null message must be sent to all LPs that have communication links established with their source LP. Hence, the routing map for null messages is much smaller than the event routing map, because a much smaller number of LPs exist in the system than do simulated gates. The routing problem becomes very

complicated after DLB, and a discussion of this is included in the DLB section of this chapter.

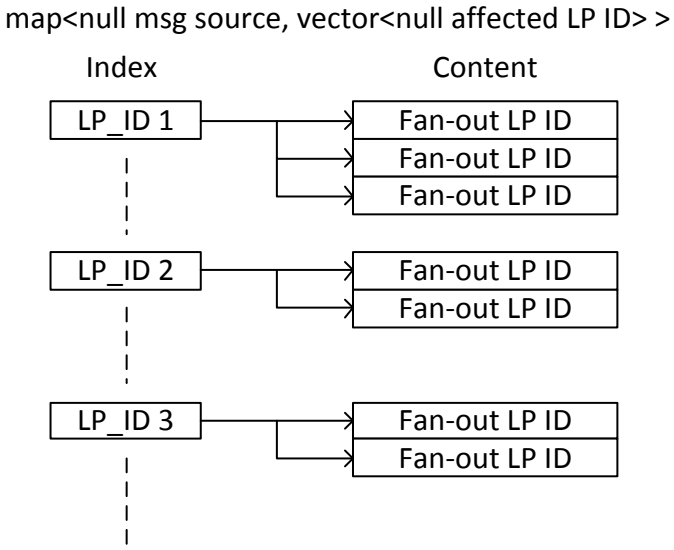



Figure 79 Null message routing map

### *Communication Protocol*

The limitation on message size requires the message to be broken down into smaller chunks in order to be transmitted, so a communication protocol is needed to allow both sender and receiver understand the fragmented messages.

A process receives many different types of messages. During the initialization stage, the circuit is imported and analysed by the master process and distributed throughout the system. The gate and wire connection information are the most common type of message at this stage. There is also information about the simulation specifications that is shared during initialization, such as simulation end time, partitioning and load balancing options.

Following initialization, the simulation operation starts and the event messages and null messages dominate the communication network. Synchronization and load balancing signals make up some of the few exceptional signals during a simulation.

However, this changes when DLB occurs. DLB effectively pauses the simulation in related LPs to start analysing; then it moves components in order to balance the workload within the system. During this process, certain components, along with the state and event associated with them, will also be transferred. At the end of a simulation, both the simulation and performance results need to be exported, and this also is reliant on the communication system to provide it.

In summary, communication is at the heart of SDES. All these different types of data need to be packed into the 32-bit payload in order to allow the SpiNNaker system to work with them, and therefore a communication protocol is designed to deal with all these various messages.

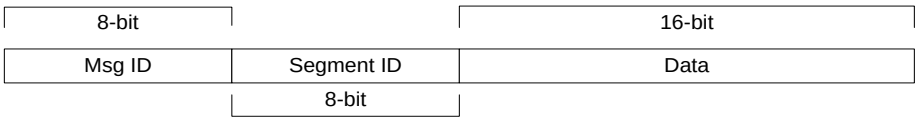



Figure 80 Packet payload format under the SDES protocol

The protocol splits a 32-bit packet into three parts, 8-bit message ID, 8-bit segment ID and 16-bit message. This creates a capacity of 256 message IDs with 4Kb content (256 segment IDs x 16-bit in each segment) in each message. The 8-bit message ID is generated by a rolling counter that counts up after each message-sending operation. The message ID is required due to a single LP can send out multiple events at the same time. And at the receiving LP, messages needs to be reconstructed to restore the original message.

For example, there are two messages “ABC” and “XYZ” need to be transferred between two nodes. If message string “ABC” is sent by a sequence of three character messages which are M1|1(“A”), M1|2(“B”), and M1|3(“C”), assuming M1 is the message ID, and the number after the vertical bar is segment ID. Applying the same to message “XYZ”, another set of messages, M2|1(“X”),M2|2(“Y”), and M2|3(“Z”), can be generated. The receiving LP can restore both messages regardless the incoming order, as shown in Figure 81. If the message ID is not available, the two messages may be reconstructed into random combination as long as they obey the segment ID, such as “AYC”, “XBC”.

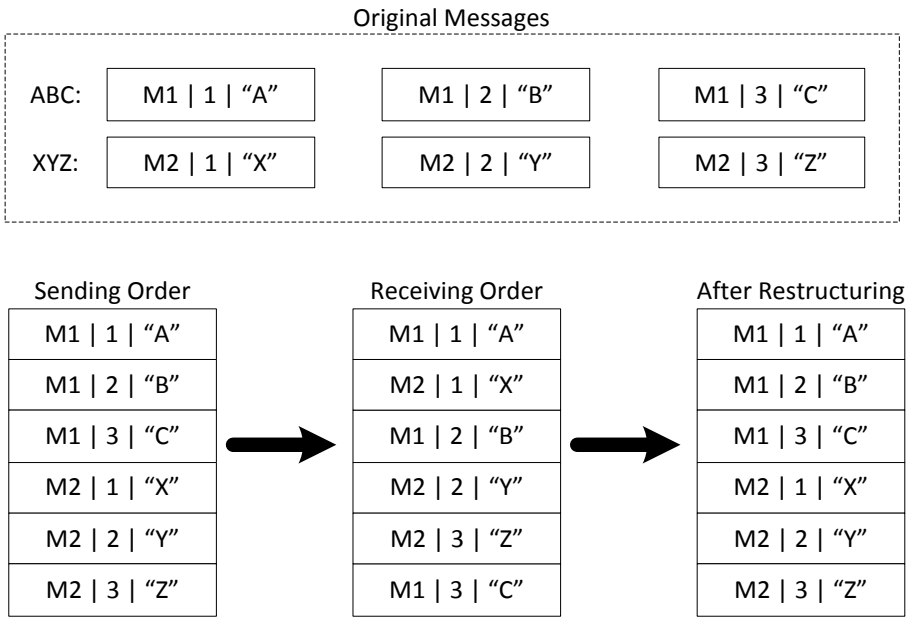



Figure 81 Message reconstructing process example

The message ID gives the message reconstructing process at the receiving end a gap of 256 messages to reconstruct the incoming message. Combined with the node ID in the packet header, these two IDs create a gap for the receiver to reconstruct the message correctly, but if two different messages are sent from a single node that is sharing the same message ID after sending 256 messages, it will cause trouble when reconstructing the message.

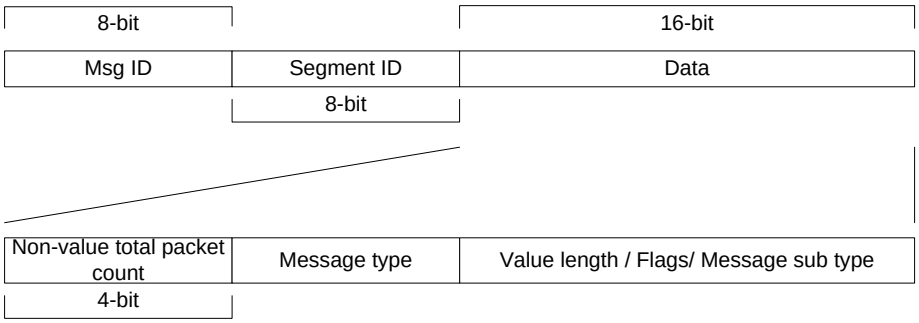



Figure 82 Header format

From a broader prospective, the protocol has two types of packets, namely the header packet and the body packet. A header packet holds information about the message type, size and length of a variable-size value, and the body packet holds detailed information about the message. The content of the body packet depends on the type of message

defined in the header. A full list of message types and their formats can be found in Appendix C.

### *Message Decomposition*

The decomposition of a message is relatively easy compared with the reconstruction work that needs to be done at the receiving end. This is because the sender has complete knowledge about what the real message is, as well as the order of packets prior to the start of a communication process. As a result, the decomposition can be implemented by a wrapper around the MPI calls. If a LP wants to send an event, all it needs to do is to call the event sending function and pass the event itself to the function. The rest will be done automatically.

The MPI calls within this wrapper will only carry 32-bits of data in each message. This emulates the communication environment in the SpiNNaker system. A sample code to send an event is shown in Figure 83.

```
void send_event(logic_event ie,int value_size,int target,int tag) {
   payload[0] = cnt;                    // Message ID
   payload[1] = msg_cnt++;              // Segment ID
   payload[2] = 0x50;                   // No. of non-value segments
   payload[3] = value_size;             // Width of value in this event
   //===Sending this 32-bit payload through the network using MPI===
   MPI_Send(&payload,4,MPI_CHAR,target,tag,MPI_COMM_WORLD);
   payload[0] = cnt;
   payload[1] = msg_cnt++;
   payload[2] = ie.time >> 24;          // Splitting 32-bit time integer
   payload[3] = ie.time >> 16;          // into four 8-bit data
   MPI_Send(&payload,4,MPI_CHAR,target,tag,MPI_COMM_WORLD);
   payload[0] = cnt;
   payload[1] = msg_cnt++;
   payload[2] = ie.time >> 8;
   payload[3] = ie.time;
   MPI_Send(&payload,4,MPI_CHAR,target,tag,MPI_COMM_WORLD);
   payload[0] = cnt;
   payload[1] = msg_cnt++;
   payload[2] = ie.id >> 24;            // Splitting 32-bit ID value
   payload[3] = ie.id >> 16;            // into four 8-bit data
   MPI_Send(&payload,4,MPI_CHAR,target,tag,MPI_COMM_WORLD);
   payload[0] = cnt;
   payload[1] = msg_cnt++;
   payload[2] = ie.id >> 8;
   payload[3] = ie.id;
   MPI_Send(&payload,4,MPI_CHAR,target,tag,MPI_COMM_WORLD);
   string str = encode_value(ie.value,value_size);  // Encoding the logic states
   if (str.size()%2==1)                             // Filling odd number bus
      str+='U';                                     // with an extra dummy bit
   for (int i=0;i<(int)str.size()/2;i++) {          // Repeat the sending process
      payload[0] = cnt;                             // until all values are sent
      payload[1] = msg_cnt++;
      payload[2] = str.at(i*2);
      payload[3] = str.at(i*2+1);
      MPI_Send(&payload,4,MPI_CHAR,target,tag,MPI_COMM_WORLD);
   }
   incre();                  // Increase message ID and reset segment ID
}
```

Figure 83 MPI wrapper for event sending operation

The aim of a wrapper is to hide the communication layer away from the simulation code. The wrapper takes logic events as its input, which consists of component ID, event time and component value. The size of the value is also an input to the wrapper. There are four logic states in this simulation system, which can be encoded to 2-bit for each bit in the component value. The encoding is desirable, because it can reduce the number of communication packets during simulation, and hence drives down the overall simulation time. The target and tag variables represent the source and target node IDs in the SpiNNaker packet (details about the SpiNNaker packet format can be found in Appendix C). The first 8-bits of all payloads is always the message ID, which increases after each message sending, and the second 8-bit is the segment ID, which also increases after each packet sending.

In the header packet, *0x50* means there are five non-value packets in this message and the message type is 0. The *value_size* shows the length of logic value in the event, which gives an idea as to how many packets are required for delivering this message. For the following four packets, they transfer the component ID and event timestamp, which are 32-bit integers, to the target. These are the five non-value packets. The number of packets for the event value portion of the message is solely decided by the variable *value_size*. Four logic values will be packed as a character, and these characters are joined together to form a string, which is transferred over the network. All other communication wrappers are implemented based on this same principle.

### *Message Reconstructing*

The reconstructing process is much more complex than message decomposition. When decomposing a message, knowledge of order of packet sending and size of the message are fully understood prior to sending. In contrast, the reconstructing process needs to establish the specification of each message from scattered packets.

There are three problems involved when reconstructing a message. *Firstly*, when dealing with messages collisions, which are caused by message sent from different sources, the packets of a single message are no longer contiguously received at the target. Therefore a buffer to store any unprocessed packet fragments is required. *Secondly*, the communication delay for each packet is different, and problems may occur when there is a very long delay waiting for one of the message packets to arrive. To improve the reconstructing performance, the reconstructing process must be capable of processing a message that has already arrived in full while still waiting for a specific packet to arrive. *Thirdly*, when reconstructing a message, a program call to other functions after reconstructing a message is inevitable. Thus the reconstructing process should be able to perform nested reconstructing operations. Based on the description above, a general picture can be established of all the problems that may occur.

With the implementation of this reconstructing process, three storage components are involved. The *first* component is the main buffer, which temporarily holds all unprocessed or unrecognized packet fragments in the system. The *second* is a

reconstructing buffer, which processes the packet fragments in their received order. For example, if the first packet received is from node 2 with a message ID of 10, it will wait indefinitely for the rest of this message to arrive. This gives an extra layer of protection to the causality rule in the simulation. However, as described above, there are some cases where there will be a long delay before the system can receive the rest of a message. To tackle this problem, a *third* reconstructing buffer is introduced. This buffer will try to find any messages that have arrived and stored completely in the main buffer. This buffer clears the backlog of transferred messages in the main buffer. Initially, the third buffer was also a reconstructing buffer that processes incoming messages according to their receiving order, but it could not cope with the overwhelming load of incoming messages. The processing flow is shown in Figure 84.

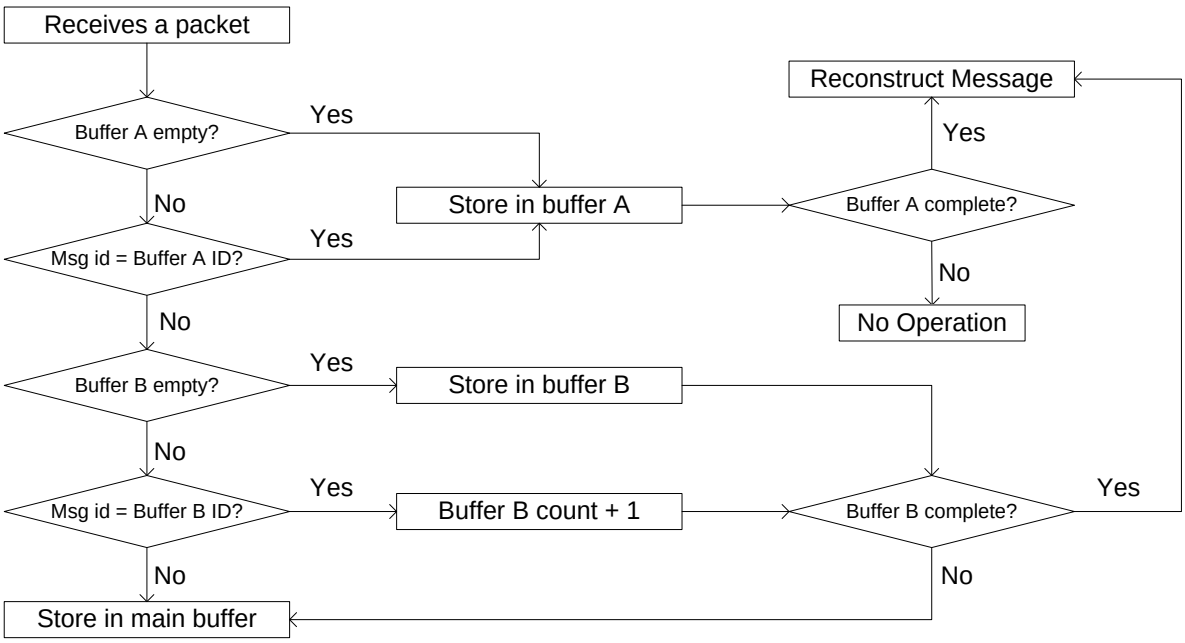



Figure 84 Message reconstructing process

After collecting together all the packets in a message, the actual reconstructing of the packets is required, and the communication protocol is the key to reconstruct a message. The header packet contains all the information needed for reconstructing, from the message type to the total number of packets involved. Part of the message reconstruction code is shown in Figure 85.

```
case 7:         // Null message
            loc_e.time = ((*in)[1][1]&255) << 24;
            loc_e.time += ((*in)[1][0]&255) << 16;
            loc_e.time += ((*in)[2][1]&255) << 8;
            loc_e.time += (*in)[2][0]&255;
            loc_e.id = ((*in)[3][1]&255) << 24;
            loc_e.id += ((*in)[3][0]&255) << 16;
            loc_e.id += ((*in)[4][1]&255) << 8;
            loc_e.id += (*in)[4][0]&255;
            clear_space(in,source,id,cnt,total);
            update_boundary(loc_e);
            break;
```

Figure 85 Message reconstructing code (partial)

The reconstructing process will first of all select the correct message type and then fetch the data held within one of the two reconstructing buffers. After the data is stored as local variables, the reconstructing buffer will be cleared to make way for other messages. Once the buffer is cleared, the functional call will begin. In the sample code case, the event is added to the local event list for future processing, and once the messages are reconstructed, it will call on other functions to update the system data in the local LP; then the simulation process will continue to simulate the system in its updated state.

## 4.6 *Dynamic Load Balancing*

There is a problem when applying the deadlock avoidance technique to a scalable system, as deadlock avoidance technique cannot balance the workload caused by the initial partitioning. In order to compensate for this, the dynamic load balancing technique is introduced. This dynamic partitioning technique can shift this workload from a heavily loaded LP to a lightly loaded LP on-the-fly.

In general, there are three types of workload during a simulation, computation, communication and synchronization. By definition, dynamic load balancing is a mechanism that distributes these three workloads across the platform on-the-fly. The synchronization cost in SDES is the cost of passing null messages that update the timing boundaries between LPs. Null messages are generated only if **at least** one event message is created. As a result, the cost of synchronization is a function of the communication cost in SDES.

Since the communication speed of the SpiNNaker platform has an edge over the conventional parallel cluster, communication cost that incurred during the simulation can be put aside at this part of the research. As shown in the results chapter (section 5.2.3 ), the communication speed on SpiNNaker platform can be at least 34x faster than its conventional cluster opponent. Based on these insights, the only type of workload that might have an adverse effect on the final performance of the SDES is the computational workload.

The logic behind optimistic simulation is that using *excessive computation power* to fully exploit the parallelism of a simulated circuit, as the clusters on which these simulations are carried out on consists of powerful CPUs, but ones that have a mediocre communication system. However, the SpiNNaker system is the exact opposite to this, as the node communication based SpiNNaker system has *fast communication links* between any two cores, although the CPUs employed are not as powerful as they are in conventional clusters; as a result, SDES shifts the parallel exploitation method from computation power focused to communication focused.

The costs of sending a 32-bit message between two LPs are 0.1 µs per hop, and 11 µs for the SpiNNaker and Iridis clusters, respectively. The 0.1 µs is the time taken for a packet to travel through a SpiNNaker node. The Iridis cluster is the parallel cluster used in emulating the SpiNNaker platform, and details of this performance test are in the communication speed section in the results chapter. Communication speed in the SpiNNaker is approximately 34x faster than it is in the Iridis cluster. As a result, the key to scalability in an SDES is to balance the computation workload evenly among LPs without considering the communication cost associated with it. This is the reason why both static and dynamic load balancing techniques are employed to ensure an even distribution of workload. The static partitioning maps the circuit to SpiNNaker nodes with minimized communication cut between them and the dynamic partitioning balances the component processing workload on-the-fly.

### 4.6.1 Load Balancing Mechanism

There are two stages in the diffusional load balancing: *measurement* and *execution*. DLB needs to establish a workload distribution status prior to performing a load balancing action. The workload in a DLB is the number of events being executed, because this is linked directly to computational workload distribution. This load information is broadcast to all node neighbours, so each simulation process will hold a copy of the workload status of its neighbouring simulation processes. Along with the workload's status, the simulated device ID that is most active within a simulation process is also passed on to the neighbours, which gives a reference point for DLB.

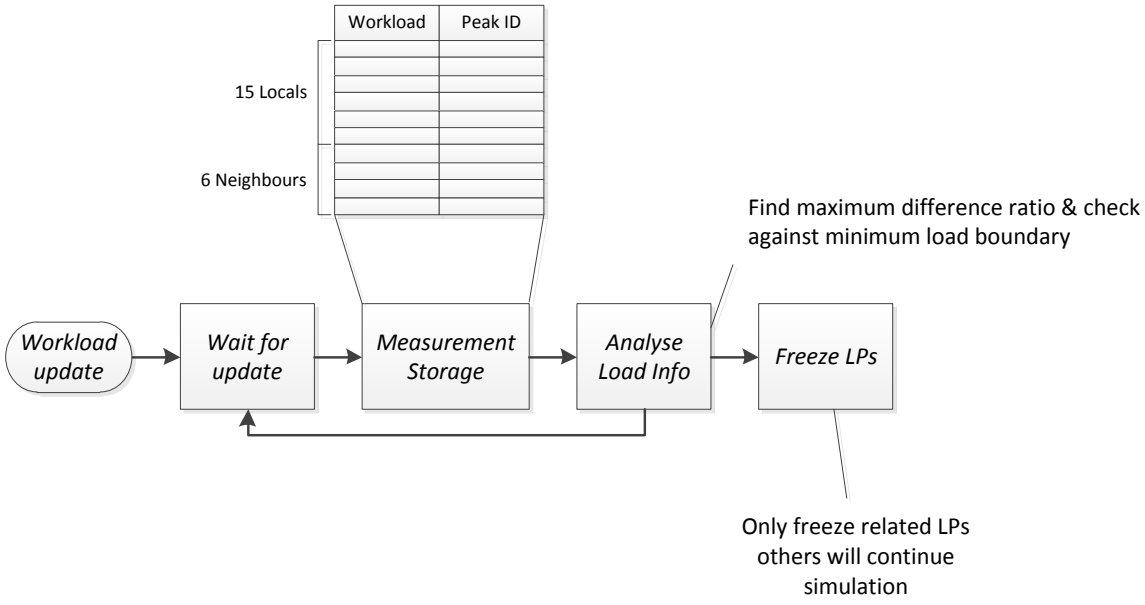



Figure 86 Dynamic load balancing process

Once the workload information is collected, DLB needs to decide whether to perform a load balance. This is determined by two factors, *minimum workload boundary* (MWB) and *maximum workload difference ratio* (MWD). The boundary prevents premature load balancing activity due to insufficient workload data: for example, if a process processed only one event, but its neighbour did not have a single event, there would be little point in changing the status quo. Without the MWB, DLB would be carried out, due to the 100% workload difference. However, this is clearly inappropriate and must be avoided.

After DLB is carried out, the workload level is logged, and unless the local workload surpasses the previous level plus the MWB again, no DLB is allowed. For instance, if the MWB is set to 100, an LP reaches 1000 events at the first DLB, and transfers 40% of its workload to another LP, then the local workload will drop to 600 and the next DLB level will be set to 1100. However, this inhibits the DLB source LP from quickly initiating another transfer. The default value of MWD is 10%, but it can be changed through amending program arguments.

The *workload difference ratio* permits the simulator to tolerate workload differences up to a specified limit, and if this ratio is set to zero, the simulator will always try to balance the system rather than to simulate the system. The process is demonstrated in Figure 86. When it comes to deciding how much *workload* should be transferred, this is normally defined dynamically by the SDES itself in accordance with the MWD, as defined by the user. If, for example, the workload difference between two processes is 30%, at least half of this workload difference, 15% in this example, is transferred to a target LP. This is the guiding transfer level, meaning that the source LP must collect enough workload to surpass this level. The default setting is 10,000 events, the same as the MWD, but this can be changed by the user.

There are cases, though, where the load difference is 100%, meaning that 50% of the workload should be transferred in order to balance the workload. However, in practice this often leads to a reverse transfer of components in later stages, because source LPs always transfer more workload than the guiding transfer level, meaning that they transfer much more workload than they actually should. Source LPs collect components until the collected accumulation of workload is greater than the guiding transfer level. As a result, an upper limit of 40% is set so as to reduce the frequency of workload backflow.

Apart from these two parameters, another parameter known as *DLB frequency* controls the wall clock time gap between two DLB events. As LPs in a SDES are event driven processes, whenever workload information is updated, the analysis function is called on.

This analysis operation not only takes time, but also obstructs other messages that try to pass through the monitor process. The DLB frequency control tackles this problem by setting a minimum wall clock time gap between DLB operations. The default value is set to 3 seconds, but it can be changed through program arguments.

**Local Hierarchical and Global Diffusional Structure**

In a conventional DLB, the simulation problem is divided into many sub-sets and these subsets are subsequently partitioned and mapped to physical processors. In effect, this creates a two tiered structure where hardware and software forms its own rigid structure. During a DLB process, although each subset is able to be moved between processors, the size of each subset is fixed throughout the simulation. This might have drawbacks, since the size of these subsets are determined by static partitioning which lacks the knowledge of live workload distribution.

Furthermore, the hardware hierarchy provided by the SpiNNaker platform naturally divides the processing power into nodes, where each consists of 18 physical cores. The hardware itself already forms a two tiered communication infrastructure. The idea behind DLB in SDES is to relieve the rigid structure imposed by the conventional DLB technique, since moving a sub-set problem can be computational demanding and not necessary. Considering the massive core count in SpiNNaker, the size of each subset will be small in comparison to a conventional environment. Therefore, instead of each subset having its own groups of data, the DLB in SDES decomposes the static partitioned structure so that each individual component can move freely around the system.

The SpiNNaker hardware requires the DLB to have two transfer layers, on-node and off-node balancing. The on-node balancing moves components between LPs within a same node, whereas off-node balancing moves components between neighbouring nodes. Since the cost of moving components within a node will be less than its off-node

opponent, due to shorter path and less processors involved, the on-node transfer will have a higher priority than the off-node transfer.

If DLB is ever required by the simulation, LPs that are subject to any component movement will be frozen. Once an LP is frozen, it cannot simulate until it has been "defrosted" by the monitor process. The on-node balancing function freezes the LPs with the maximum and minimum workload within a node, leaving the rest of the LPs to carry on with the simulation. In the case of off-node balancing, workload balancing will freeze the local LP in the node that has the maximum workload, and in the node with the lightest workload the least loaded LP in the balance target node will be frozen. In both cases, the monitor will enter balancing mode and reject any other requests for DLB from other sources.

### 4.6.2 Implementation Obstacles

There are difficulties involved when merging DLB with conservative simulation. The *first* of the problems is the need to correctly handle messages while performing a balancing action. When a monitor enters balancing mode, messages do not stop arriving at the monitor: some of them even target the LPs involved in its balancing action. If they are routed to the balancing source LP, the message might be ignored later in the simulation process, because the balancing source LP may have already moved the component at which the arriving message is targeted. *Hence*, during the balancing mode, a monitor process will store any messages for balancing related LPs and process them in batches after balancing has been completed.

The *second* difficulty faced by DLB is the distributed nature of the data around the system. In both conservative and optimistic simulation, the simulated circuit is partitioned at the start of a simulation, and this partitioning is not altered during the simulation. Therefore, a full history of events of all local gates can be accessed without any difficulty. However, after DLB is introduced, components are moved on a regular basis, so future events, as well as the most recent logic values associated with moved components, need to be sent to the balancing target LPs. Furthermore, some of these values may not be available locally. *As a result*, a function that fetches historic values of

a component in foreign LPs is required. This guarantees that the system states are correctly set up in the balancing target LP prior to moving on from the balancing stage.

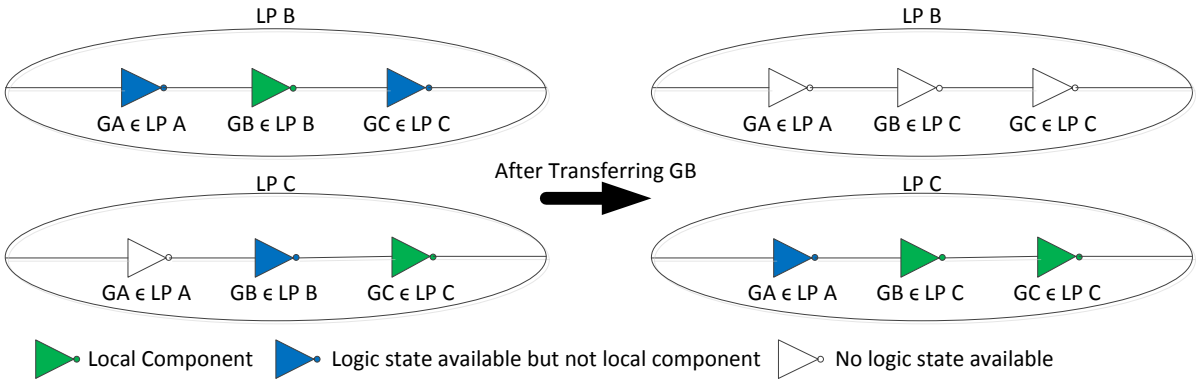



Figure 87 Distributed data handling at DLB

In Figure 87, a sample circuit fragment is shown. Gate A (GA) is situated in a foreign LP for both LP B and LP C. When gate B is being moved from LP B to LP C, the input signal of gate B, which is LP A, is also required to be transferred to LP C, and this prevents the balancing target from processing events with out of date or missing circuit states. As initially LP A is not a direct input to the local component gate C (GC) in LP C, its logic state is hence not updated.

The *third* problem is collecting enough gates to balance the workload between LPs or nodes. There are two references enabling a slave process to gather its workload: component with peak activity count; and the difference in workload between processes by percentage. It was decided to have three stages to the data acquisition activity. The first round collects the fan-in and fan-out gates of this peak component. If the first round fails to gather enough workload, the gates surrounding the peak component will be gathered during the second round, and yet another round of component collection may be carried out if there are still not enough components to meet the workload transfer ratio. This will choose the most active component that was not included in the first two phases, and will restart the process again. A sample workload gathering illustration is shown in Figure 88.

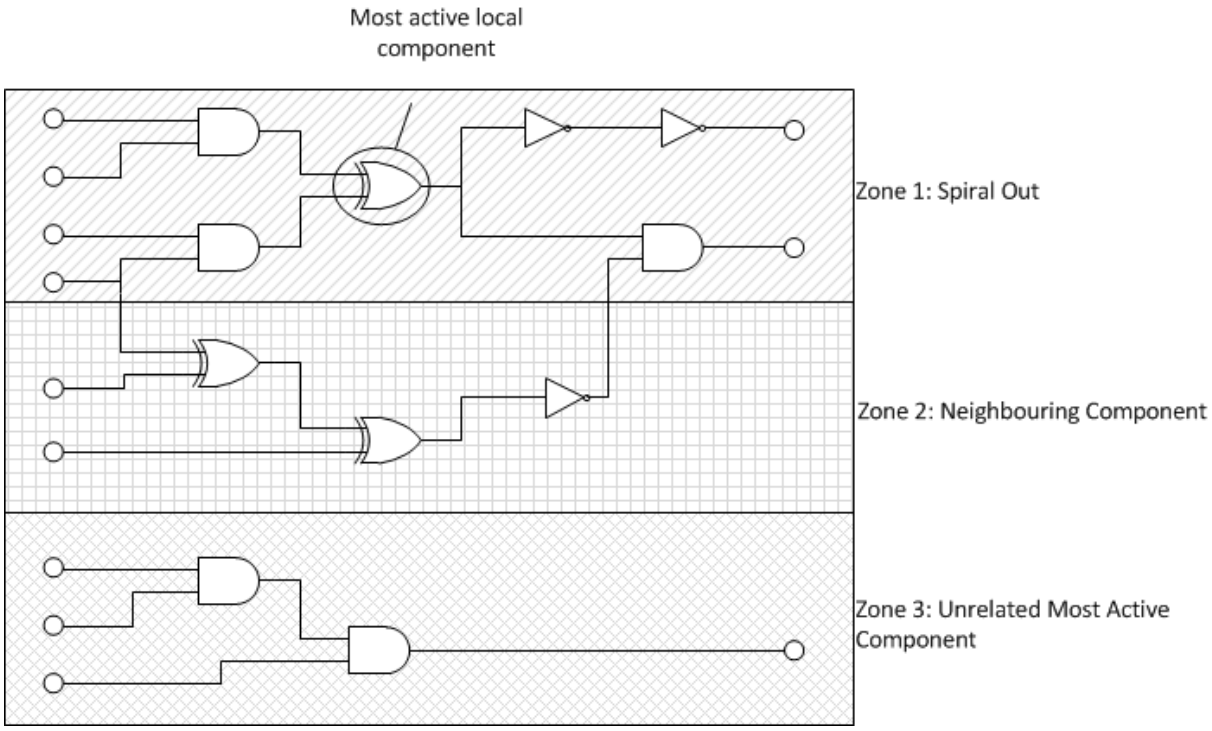



Figure 88 Workload gathering rule

This workload collection mechanism is built around the idea that the most active component always sits in the busiest area of activity in a local partition. Collecting the gates directly or indirectly linked to it will generate the maximum workload package whilst at the same time including the least number of gates, which helps reduce the component transfer time during DLB.

The workload collection mechanism accumulates components that belong to zone 1, which has direct I/O relationship with the most active local component, one by one until it collected all components in zone 1. Then the mechanism search for components in zone 2, which has indirect I/O relationship with the most active local component. If all components in zone 1 and 2 are collected but still have not gathered enough workload, then the process repeats again on the most active local component outside zone 1 and 2. The process repeats until it has collected enough workload.

### 4.6.3 Impact of DLB on processes

DLB has an impact on both slave and monitor processes. The slave process has to add extra functions in order to fulfil the component collection function required by the SDES system, and the monitor process has to manage the DLB process and collaborate with adjacent nodes.

#### *Slave Process*

There are two major components added to the slave process by the introduction of DLB; the global virtual time service and the DLB component and event transfer mechanism.

***Global Virtual Time (GVT)***

GVT requires synchronization, which in conventional simulators there are three ways of approaching the problem (a grid-based, a tree-based, and a centralized)[134], [135]. However, synchronization in SDES does not stop any of the LPs from continuing the simulation. Instead, a token with two integers is passed on by the LPs. The first integer is the *real* GVT, which LPs can use as a reference to determine the end of a simulation. The second integer is the *potential* GVT, which tries out a new GVT ready for the next real GVT. LPs cannot modify the real GVT, but they can assign the local time to GVT if the local time is lower than the potential GVT in the original token. After travelling through all LPs once, the potential GVT is assigned as real GVT and the next synchronization is ready to be launched. This synchronization process cannot collect the real-time GVT in the system, but it has the benefit of allowing the simulation to keep flowing whilst at the same time determining the GVT.

This is different from the GVT introduced in time warp technique. The original GVT sends out local virtual time requests to all LPs concurrently, and collects the local virtual time feedback from all LPs gradually. This can potentially generate a great number of packets during a simulation. The special property of the communication system on SpiNNaker platform requires data to be transferred by hopping all the nodes along the path between two distant nodes. The packets can flood the communication system around the GVT initiating node, as local virtual time feedback from all the LPs in the system try to reach a single core. Moreover, this process is required to carry out

on a regular basis, which can potentially cause problems. As a result, a modified version of GVT collection service is implemented in SDES, which collects the data sequentially.

The DLB *component and event transfer mechanism* is designed to collect components using the mechanism outlined in section 4.6.2 . The LPs maintain a pointer to the local component with the highest workload when processing events, and the total local workload is also tracked throughout the simulation, which provide reference points for the transfer mechanism. The highest workload component forms the origin of component collection, and the total amount of events that need to be collected is a variable ratio in comparison to the total workload in a local LP. This ratio has an upper boundary of 40%, which allows the LP to transfer a maximum 40% of the total workload in a local LP in any one DLB operation. Although there is no lower boundary imposed on the ratio, the MWB and MWD imposed by the monitor process (see section 5.2.4 ) protects the DLB from transferring too little workload.

The 40% workload ratio is set to prevent DLB zigzagging, because a lower ratio of workload is then transferred between the processes, creating a barrier against the DLB moving the same component back to its original position. This is because, the 40% workload ratio is again imposed when the component is trying to move back, which only allows 40% of its original 40% workload to be transferred for the second time. Even in the event that this happens, the 40% ratio greatly reduces the likelihood continual transfer, and the workload based collection mechanism collects different components according to the component with the highest workload.

In an extreme case, where the workload for a single component is over 40% of the overall workload in a local LP, the LP will simply block the component transfer request and wait for the monitor process to resume normal simulation.

### *Monitor Process*

Similar to the slave process, the monitor process also has two stages in the DLB system. The first stage is the *initialization* stage. The component transfer can occur between

local LPs in a local node, or between two adjacent SDES nodes. The initiation of local node DLB is triggered if the workloads in two local LPs are greater than the MWB and the difference in workload has exceeded the MWD. The initiation of cross node DLB is more complicated. The monitors in both source and target nodes will be frozen to prevent any further DLB operation. The sender's monitor has to ensure that none of its fan-in nodes are, at the same time, engaged in any DLB operations, and if these criterions are not met then the DLB operation will be cancelled. The MWB for a cross node DLB is relaxed to *n*(workload boundary)*, where *n* is the number of LPs in an SDES node. The *operation* stage of both scenarios is explained in the rest of this section.

In a local node DLB scenario, the DLB operation carried out in the local node and is irrelevant to the external nodes, as the monitor-partitioned information is not available to external nodes anyway. However, the monitor process does need to shield itself from further DLB requests from other nodes, so any incoming DLB requests entering during this time will be rejected.

After the monitor is shielded, the monitor needs to freeze both the source and target LPs from event execution. During this freeze period, any messages that have a destination set to either of the frozen LPs will be stored in a message queue in the monitor process. When the source LP receives the freeze signal, it starts to collect the components until it meets the workload criteria, which is packed into the freeze signal. The source LP has no knowledge of the destination of these removed components, but after the DLB process it will flag these components as a foreign. Immediately after the collection of components is finished, any pending or processed events associated with the removal component are also transferred to the monitor process. Only the processed event with the highest timestamp will be transferred, so as to indicate the current logic state of a component. These components and events will be transferred to the target LP via the monitor. In the meantime, the monitor must distinguish these DLB-related unprocessed and processed event messages from normal event messages, as DLB does not stop other non-DLB associated LPs from simulation. However, the LP in a DLB operation does impose a boundary constraint on other LPs until it is defrosted by the monitor process.

Once all the DLB-related data is forwarded by the source LP, a confirmation signal is generated by the source LP and passed on to the target LP via the monitor process. An acknowledgement signal is fed back to the monitor process by the target LP, which allows the monitor process permission to resume normal simulation operation. At this point any messages stored in the monitor process that have either the source or target LP as its destination are released, and the boundary re-establishment stage starts, which is discussed later in this section. The monitor process defrosts itself, as well as the source and target LPs, so that the simulation process can resume.

In the cross-node DLB scenario, the freezing sequence is similar, but instead of freezing one monitor and two LPs, two LPs in two different nodes, as well as at least two monitor processes, will be frozen. After freezing the processes, the transfer is very similar to the local-node DLB. The difference between these two DLB scenarios is that all the components, events and control signals have to be routed through an additional monitor process. *Another* difference lies in the partition information update. In a local-node DLB, the node information of a component is not changed throughout the process, which is not the case with a cross-node DLB function. The new node information must be sent to the fan-in node that has a connection through the sender's monitor to one of these transferred components. This enables the fan-in nodes to immediately redirect any messages to the correct destination.

The simulation progress is not affected by the DLB process, because the event with the lowest timestamp will still be safe to be processed after the DLB process. A Deadlock problem may arise after the DLB process, however, additional null messages were introduced to update the timing information for all LPs, releasing the LPs from any potential deadlock created during the DLB process.

*Moreover*, the monitor process must analyse the structure of the circuit after the components have been transferred. This process may eliminate the boundary array,

introduce a new one, or even keep the same boundary. In the case where an additional boundary is created, the monitor process sends a signal to the node of the new boundary and forces this node to send out a null message so as to update the boundary between these two nodes. *Furthermore*, in a local DLB, the source and target LPs have the same monitor process, which is different from a cross-node DLB. In a DLB, the acknowledge signal generated by the target LP is only readable by monitor process of the target LP. This process will defrost the local target LP as well as the source LP's process, which will trigger a further round of defrost signals to the source LP and all frozen fan-in nodes.

## 4.7 *Summary*

This chapter covers four main topics related to the SDES system implementation.

- **The SpiNNaker emulator** is an important part of the overall SDES system. The emulator replaces the interface between the SDES system and the SpiNNaker hardware system. It emulates a simple packet receive interrupt in a local process, and a packet passing mechanism which is transparent to the SDES system.
- **The SDES system** and its implementation process are also discussed in this chapter, and the landscape of all the data in the SDES system is laid out. The division of labour between monitor and slave processes is explained; slave processes (LPs) perform event processing tasks, monitor processes focus on message routing and dynamic load balancing controls. The modelling of digital circuit states is also introduced.
- **The Communication platform** encodes and decodes the 32-bit packets and allows the SDES LPs to communicate with their peers using messages, but leaves the low level communication packets to the platform.
- **The Dynamic load balancing** technique employed is a sender-initiated diffusional technique which can cope with the fully distributed SpiNNaker system. Three main parameters can be modified to change the DLB behaviour, namely MWB, MWD and DLB frequency.

# Chapter 5 Results

The goal of developing SDES is letting the simulator utilize the advantages of the SpiNNaker to outperform conventional parallel simulation. A way to illustrate this is to use SDES on a wide variety of different circuits. This chapter focuses on the performance of SDES , applied to the portfolio of test circuits. The chapter has two sections, discussing the partitioning and simulation results respectively.

## 5.1 *Partitioning Results*

Prior to introducing the simulation results, the simulator must first be able to partition the input circuits under certain parameters and constraints. There are two types of partitioning mechanisms in SDES, static and dynamic. The effect of dynamic partitioning can only be illustrated under live simulation. Hence, the effect of dynamic partitioning is discussed in the next section of this chapter.

The static partitioning system in SDES deals with the input circuit just before the simulation, which is a CLG. The aim of static partitioning is to reduce the number of wire cuts produced by partitioning the circuit. Therefore, in order to justify the movement of a component, the movement must produce a reduction in wire cut cost. In addition, the gate count balance between different partitions has to remain within certain limits. This problem is solved in SDES by limiting the maximum and minimum number of components resident in a single partition. Once a partition reaches the maximum number of components, it will not accept any further inflow of components. Conversely, when a partition reaches the lower boundary, outflow is prohibited.

To test the efficacy of the partitioning circuit, a DES circuit is used. For the purposes of increasing the size of the circuit, additional DES blocks are connected in series, as illustrated in Figure 89. A single DES consists of 5.5K gates in CLG.

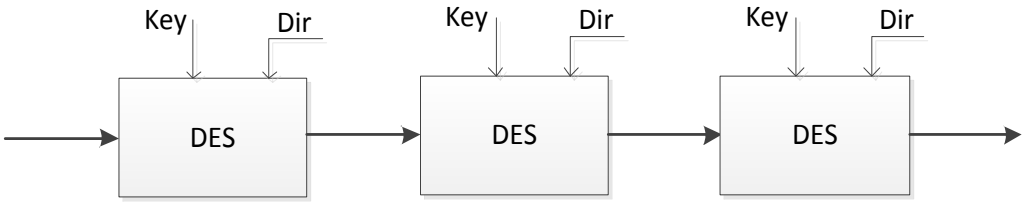



Figure 89 A chain of DES blocks used in the partitioning test

Static partitioning requires an initial partitioning before performing the multi-way re-distribution. This initial partition can be trivial for example random partitioning or naïve partitioning for example in the order of component ID (block partitioning). By using block partitioning, the number of wire cuts for single, triple and quintuple DES circuits are shown in Figure 90.

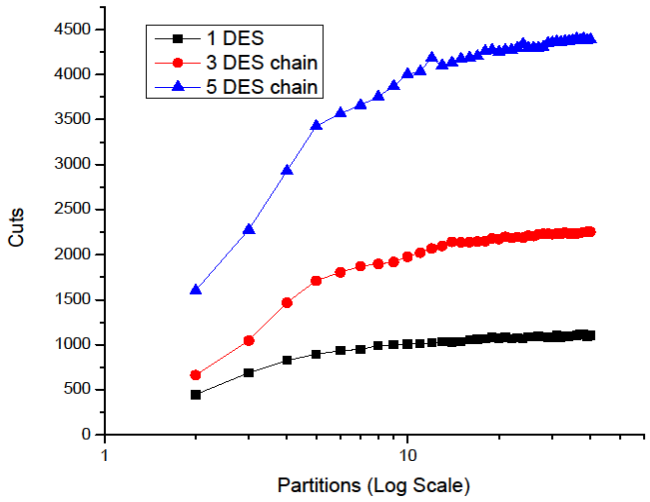



Figure 90 Initial wire cuts under block partitioning

The wire cuts in Figure 90 represent the total number of wire cuts within the system. As the number of partitions increases, the total number of communication cut by partitioning increases, this cut is referred as partitioning cost or wire cuts. However, the partitioning cost saturates when it approaches a higher number of partitions. This is due to fewer numbers of gates located in each LP as the number of partitions increases. Hence the possible wire cut count within each LP decreases as the partition size increases, as shown in Figure 91. The trend is towards a single digit wire cut per partition, when the number of partitions approaches the number of gates within a circuit.

The clock signals and other multiple fan-out gates prevent the wire cut per partition reducing to zero.

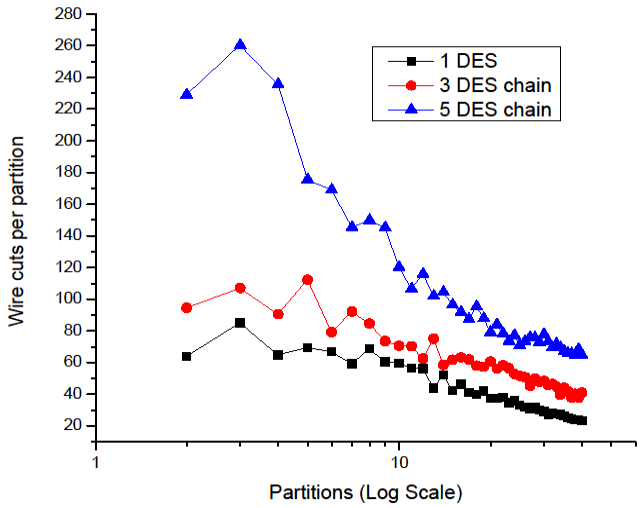



Figure 91 Initial wire cut distribution

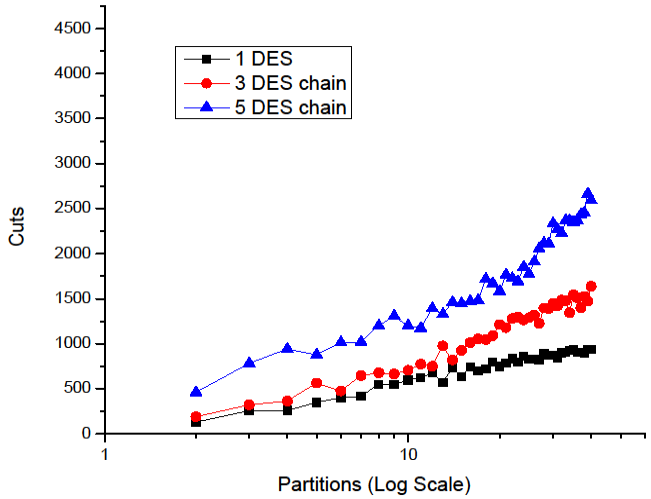



Figure 92 Improved wire cut distribution under block partitioning

The multi-way partitioning algorithm, which is based on Kernighan–Lin algorithm, but extended to support multiple partitions (see section 2.3.1 for detail), can effectively reduce the number of wire cuts within the system, as shown in Figure 92. The reduction is over 40% if the initial wire cuts per partition are over 60. The multi-way partition result gradually converges to that of the block partitioning method, due to the reduction

in the number of gates in each partition. When each gate is assigned to a distinct partition, there is no way for the partitioner to reduce wire cut count while balancing the number of gates assigned to each partition.

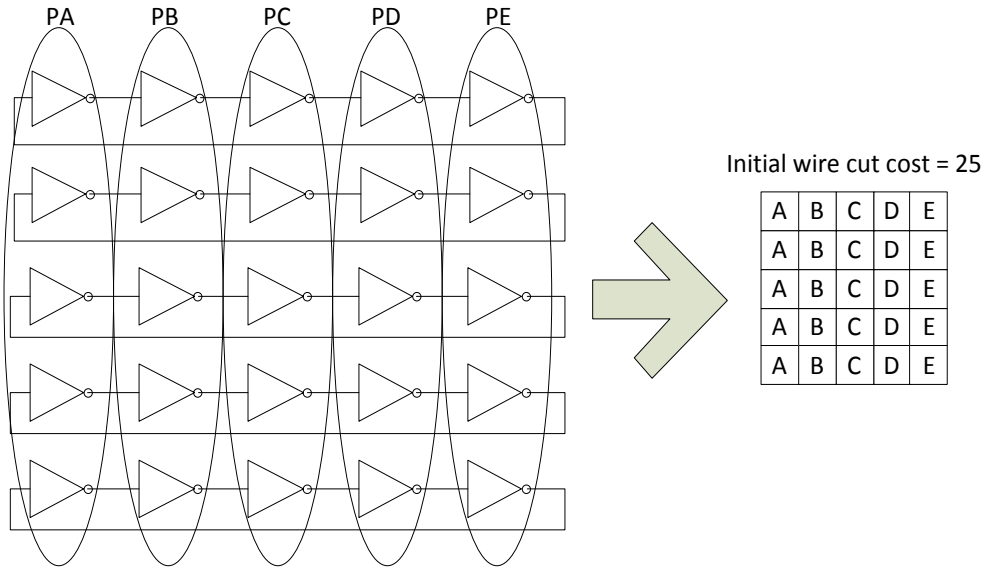



Figure 93 Five individual 5-stage clock generation circuits

In some extreme cases, the multi-way partitioning algorithm may fail to perform as well as manual partitioning. Take Figure 93 as an example, the 25 inverters are split into 5 partitions where *all* wires have been cut. The best possible partition can be easily identified manually, where no wire cuts are required and the circuit is perfectly balanced in terms of gate count. When this circuit is fed to the multi-way partition algorithm, the algorithm fails to identify the optimum partitioning setting. This is because of the lack of general structure analysis during the partitioning process. In other partitioning methods, such as clustering, an analysis of local connectivity can prevent the split between tightly connected components. Due to the limitations in time, the multi-way partitioning method is implemented. From the wire cut performance point of view, the number of wire cuts is reduced from 25 to only 9. In this sense, the algorithm still works.

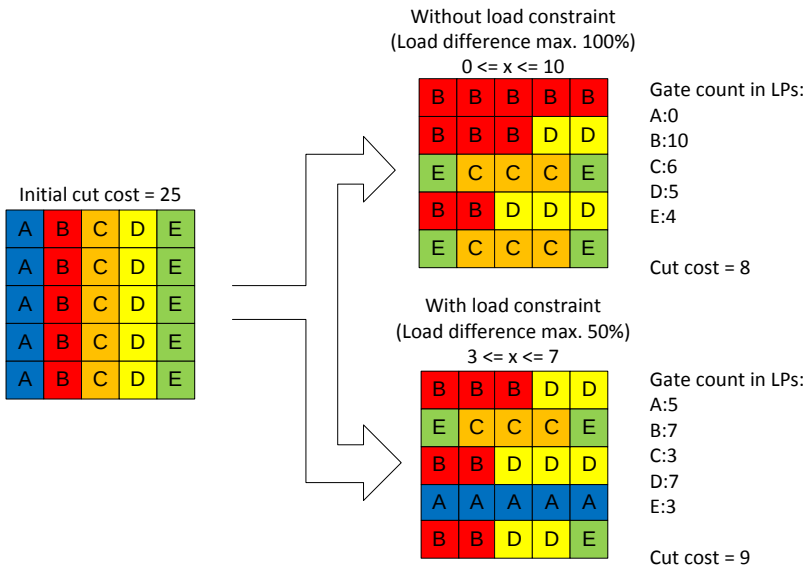



Figure 94 Partitioning result map for 5-stage clock circuit

Figure 94 shows results generated by the static partitioning algorithm. With a loose load constraint, the static partitioning method can tolerate 100% load difference between partitions. This can lead to empty partitions, due to the tendency of putting all the workload into a single partition. To avoid this, a stricter load constraint which only allows 50% load difference in the test below shown in Figure 95. After imposing the limit, the cut cost increases by 12.5%, but the standard deviation of workload between LPs reduces from 3.6 to 2, a 44% reduction.

The previous test illustrates that the multi-way partitioning algorithm cannot handle extreme cases, but in other cases, it shows the capability of reaching a good partitioning result.

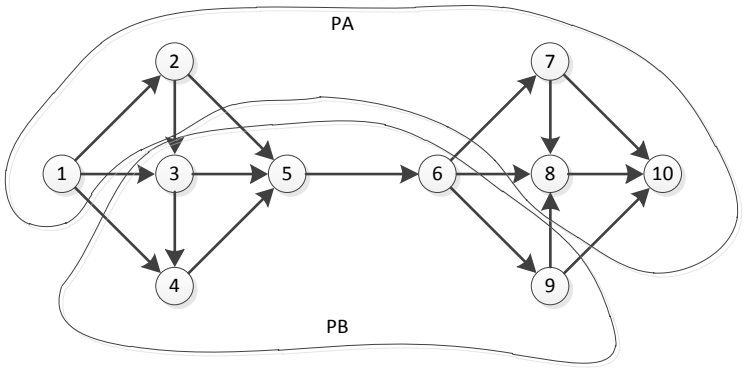



Figure 95 Sample circuit illustrating partitioning effectiveness

In the case of Figure 95, the best partition can also be easily identified, which lies between gate 5 and gate 6, but the initial partition setting mixed the gates in both gate blocks and hides the best possible cut wire in one of the partitions. The process of multi-way partitioning shown in Figure 96, demonstrates the partitioning changes occurred during the process. The number in each component represents the number of wire cuts that can be reduced. Positive number means a reduction of communications cut can be achieved, where negative means an increase in communications cut if the component is shifted to another partition. In short, the multi-way partitioning method may not be able to find out best partitioning setting, but it can reduce the wire cut cost, and hence communications cost.

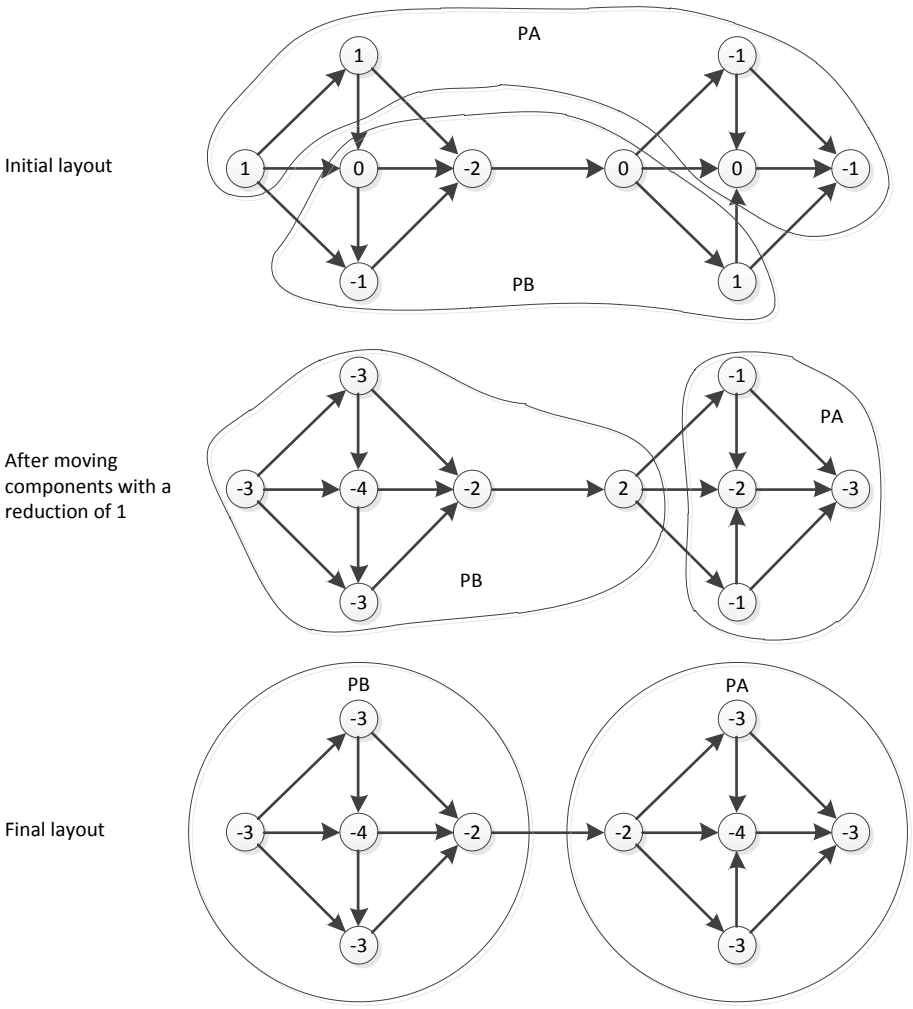



Figure 96 Sample circuit partitioning process

## 5.2 *Simulation Results*

In chapter 3, a portfolio of circuits designed to exercise in various aspects of SDES was presented. Their aims are to show the performance of SDES under specifically designed conditions. Most of these conditions try to reach the performance corner cases for SDES, such as high combinational circuits or pure sequential circuits. Hence, the performance of SDES for various other different circuits can be estimated according to their simulation performance under Iridis cluster. As explained in section 3.3 , there are three aspects of SDES whose performance is of interest: event processing, communication and dynamic partitioning. The tests carried out to evaluate their performance are explained in detail in this section.

### 5.2.1 Instrumentation

In order to provide a clear image of how the system performs, a set of uniform performance gauging references are required. On the simulator side, a mechanism to measure the timing and workload is required. They provide the performance data which can be compared to other references. On the portfolio side, they need to be characterized as well as the simulator. This is because the complexity of a simulation varies even under the same circuit but different in test benches. As a result, this section is naturally divided into two parts.

#### *SDES performance instrumentation*

In order to demonstrate the performance of SDES, a probing mechanism is required. Among the mountains of data produced by SDES, *timing* and *workload distribution* is especially important. Traditionally, the Multi-Processing Environment (MPE) package from MPICH, an implementation of the MPI standard, is used to collect *timing* information. MPE creates a log file full of timing information of the discrete time instances during simulation. After the simulation, this log file can be viewed via a dedicated log file viewer Jumpshot. However, the size of SDES states overwhelms the logging system of MPE. MPE stores too much information and the analysis tool is unable to cope with the size of the performance data generated by SDES, which easily generates more than 10Gbytes of logging data. As a result, the MPE system is not employed. Instead a simplified version of it was implemented in SDES.

Instead of using the complex MPE logging command, SDES stores all the timing information in memory as an array, which is then exported to a file and analysed by an external program. As the communication time is the only time that can be scaled down in the process, the time monitoring activities are focused on communication time.

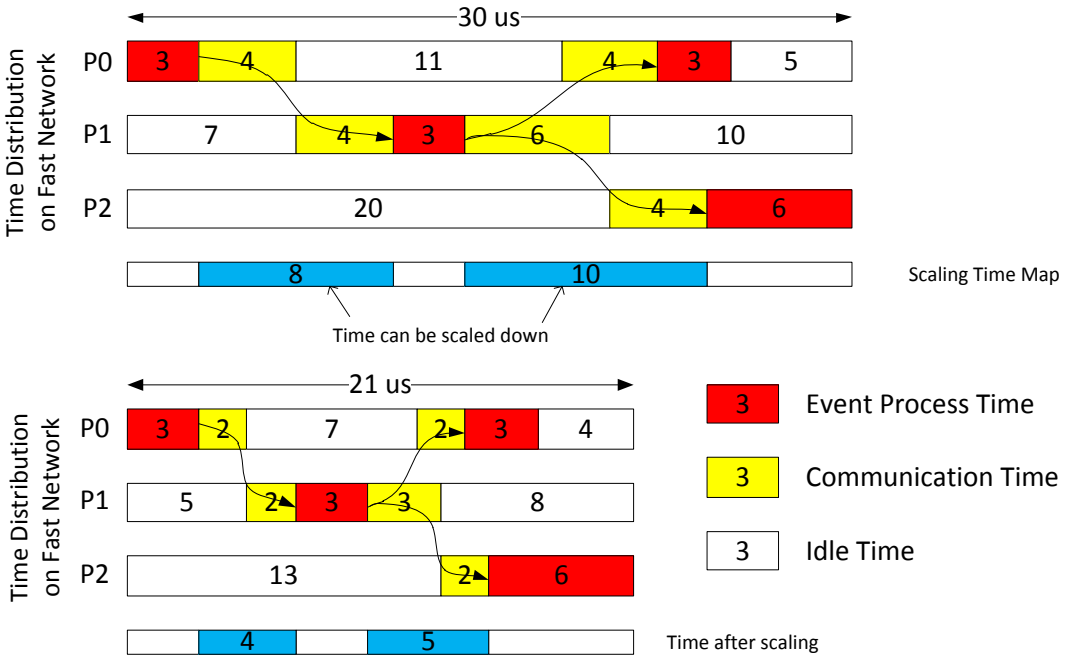



Figure 97 Communication time scaling example

The scaling mechanism is explained in section 3.3.4 . And as a reminder, the description figure is shown again in Figure 97. By scaling down the communication time labelled by the time sampling points, the communication time cost after transferring the design onto SpiNNaker can be estimated. The speedup of simulation can also be calculated by dividing the simulation time on single processor by the time on parallel processors.

$$Speedup = \frac{T_{sequential}}{T_{parallel}}$$

***Workload distribution*** on the other hand requires relatively detailed information at different discrete time instances. Workload in SDES is defined as the accumulated number of events being processed on a LP. DLB in SDES requires each gate component having an event counter attached to it, which counts all the activities occurring on each gate. This detailed information provides a perfect reference for DLB with which can work. Workload information is the key to dynamic load balancing. The dynamic nature

of the workload distribution means that enormous quantities of runtime data must be collected. However, not all the details are useful from the point of view of the observer. As a result, a coarse measurement of the workload is applied.

***Gate distribution*** collects the gate count in all LPs whenever a change in partition occurs. This provides an overview of the workload among LPs as a function of time. Although equilibrium in gate count does not necessarily mean an equilibrium state in workload, it can provide an insight to where the workload is and how much DLB has shifted the workload from one LP to another. Furthermore, gate distribution only generates a few data points which dramatically reduce the number of logging data in comparison to workload distribution, and both distributions tell a similar story from the point of view of the observer.

Along with the gate distribution data, the size of the event list gives a good estimate of the future workload on a single LP. In general, workload on a single LP should roughly equal to the integral of the size of the local event list (Figure 98). The differences between the two are caused by events that do not change the value of a gate or disappear at the output port of the system. This relationship can predict the workload distribution and hence compensate the gate distribution data in terms of the real workload. With these two monitoring mechanisms incorporated into SDES, the performance of SDES can be extracted with ease.

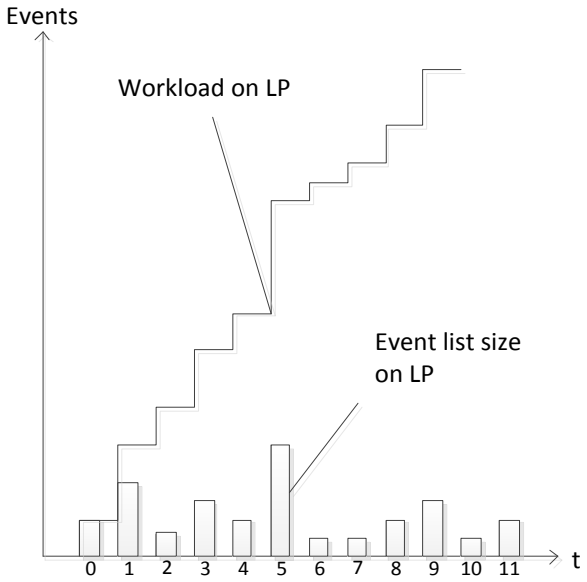



Figure 98 Relationship between event list and workload

### 5.2.2 Event Processing Speed

The performance of the event processing engine can be described in two parts. The first question raised is the event processing speed during sequential simulation. This provides a guideline to compare the performance of SDES with other discrete event simulators of similar type. The second part is the parallel performance of SDES, which shows how the event processing power grows with the available computing power provides an insight to the scalability of the simulator.

**Sequential simulation speed**

The first performance parameter for a simulator is the event execution speed. This is an essential parameter, as it shows the event processing effectiveness of a simulator. It can be compared across different systems, as shown at the end of this section. The size of the main event list at any time instance has a huge impact on the event execution speed. In order to demonstrate this effect, circuits with different size are simulated in sequential mode.

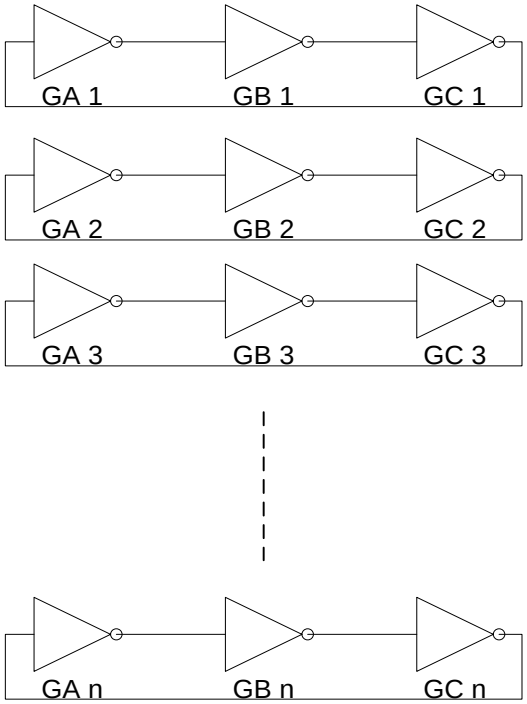



Figure 99 Individual clock generation circuit

An array of ring oscillator circuits is employed to explore the event execution speed. Each individual clock generation circuit is a 3-stage ring oscillator (Figure 99). All of them are running at a clock speed of 333MHz. Each oscillator generates 1 event at any discrete time instance. In this test, the number of inverter oscillators involved ranges from 1 to 1000 ring oscillators. The tests measure the overall simulation time for the simulator to process a fixed amount of events across different sizes of circuit. The amount of events simulated ranges from 1 million to 10 million. The simulation time against the number of simulated events is plotted in Figure 100.

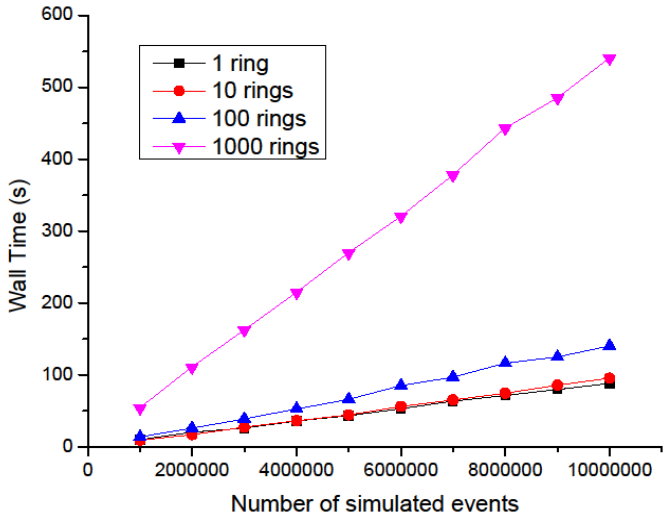



Figure 100 Simulation time in wall clock vs. number of simulated events

The time difference for simulating the same number of events is due to the increase in search space when new events are generated, as a newly generated event needs to search and compare with all the existing events that fall in the timestamp. The last event in a 10 ring oscillator circuit requires comparing with 9 other events to check if there are any duplicated value alternations. If these duplicated events are not removed from local event list, after propagating through the communication network, the order of execution for events carrying a same timestamp can be out of order, and hence the wrong result can be produced. The total number of search and compare operation needs to be carried in a single time step out can be represented as $^{\mathrm{N}(\mathrm{N}-1)}/_{2}$ , where N is the number events sharing the same timestamp. If the total number of search and compare operations for a 1 ring oscillator circuit is *x*, the *n* oscillator circuit needs to perform $^{x(n-1)}/_{2}$ operations.

The linear relationship between the overall simulation time and the total number of simulated events shows the event execution speed is fixed for a circuit. However, the execution speed varies wildly. Based on Figure 100, the event execution speed for the four different circuits are 110K events per second (eps) for 1 ring oscillator, 108K eps for 10 rings, 73K eps for 100 rings, and 18K eps for 1000 rings. This gives a spectrum

of speed for SDES to compare with other discrete event simulators. Other work [89], [136], [137] achieves a speed of 2M eps and 500K eps in serial mode respectively. Both techniques are pursuing the time warp technique, which tested to have a better performance than deadlock avoidance technique in some cases. This shows that there is a 20X – 100X sequential performance gap between the SDES and other implementations. This is a great room of performance improvement in the future. The focus of this project is to develop a simulation system which is capable of large scalability. The absolute speed is not the main focus at this stage.

**Parallel simulation speed**

The relationship between the input computing power and the performance speedup defines the performance of a parallel simulation. This is different from the evaluation of sequential speed.

In a parallel environment, simulators have to deal with communication between processors as well as event execution. However, it is the event execution that counts in the speedup measurement. If communication processing time dominates a simulation, the speedup in this simulation is capped due to the limited time contributed to event processing, upon which the speedup measurement is based.

The individual clock generation circuit is developed to reach the computational end of the performance spectrum. If the individual clock generation circuits are ideally distributed among LPs, there is no communication involved during a simulation.

The delay on each inverter is 1 ns, which means the clock rate in each ring oscillator is 333MHz. The simulation end time is set to 20 μs, so each of the ring oscillators generates 20,000 events in the overall simulation. As a result, by simply stacking up the number of ring oscillators, the density of events in the test circuit increases.

If the speedup that is shown by the SDES continues to increase until the number of LPs reaches the number of gates in the system, the SDES can be asserted to have a good scalability. In this test, three sets of ring oscillator arrays are employed, containing 25, 50, and 100 oscillators. They have 75, 150, and 300 gates respectively.

A key parameter needs to be determined before applying these tests: the optimal number of LPs to be employed in each virtual SpiNNaker node. The physical structure of the SpiNNaker hardware system has a two-tiered processing system, the monitor process and the slave processes. The monitor process might become a potential bottleneck of the SDES system.

To establish the value of this parameter, the 100 ring oscillator array is employed, which contains 300 gates. By fixing the number of LPs to around 300 and simulating under different numbers of virtual SpiNNaker nodes, an insight provided to this problem. In addition, this test also reaches the point where the number of LP equals to the number of gates in a CuS.

The only parameter changes during the simulation is the ratio between monitor and slave processes. The number of LPs and monitors is shown in Table 7. The first row shows the number of LPs per node, which is the only parameter changed during the simulation. The second row shows the number of nodes required to reach 300 LPs, as there is only one monitor process per node, this row also presents the number of monitors exists during a simulation. The third row shows the number of LPs which are the effective number of processes engaged in event processing. The total number of processes is the sum of the number of monitors and LPs *plus* an additional overseer process which controls the initialization and finalization stage of the system.

| LPs/node | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Total Nodes / No. of Monitors | 300 | 150 | 100 | 75 | 60 | 50 | 43 | 38 | 33 | 30 | 27 | 25 | 23 | 21 | 20 |
| Total LPs | 300 | 300 | 300 | 300 | 300 | 300 | 301 | 304 | 297 | 300 | 297 | 300 | 299 | 294 | 300 |
| Total Processes | 601 | 451 | 401 | 376 | 361 | 351 | 345 | 343 | 331 | 331 | 325 | 326 | 323 | 316 | 321 |

Table 7 Total process count during simulation

The graph (Figure 101) shows that the simulation with 5 LPs per node has the fastest performance relative to other numbers of LPs per node. In other words, a monitor to slave process ratio of 1:5 is the most effective rate for monitor process to avoid being a bottleneck. The initial improvement of performance is mainly due to the increase in event processing power while the cost of monitor is relatively fixed. However, as the number of LPs on a node increases, the bottleneck effect of monitor process starts to appear, as more and more LPs request attention from the monitor process.

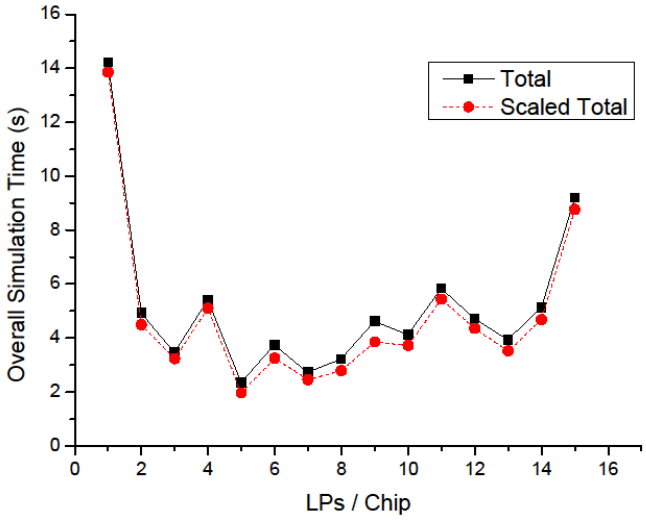



Figure 101 Performance under various numbers of LPs per node

The ring oscillator array can now be simulated using five LPs per node to compare the performance speedup relationship across different circuit sizes. By applying the 34X scaling, the maximum speedup that can be achieved should be much higher than it is on the Iridis platform, as the event processing time is negligible compared to the communication required to get the event across the system. The result graph is shown in Figure 102.

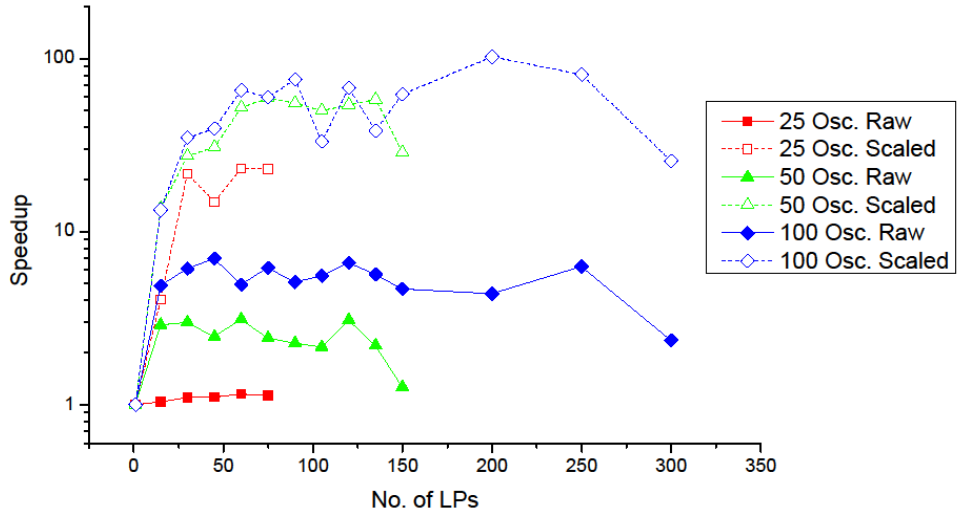



Figure 102 Scalability test with different ring oscillator array size

As the size of the ring oscillator array increases, the maximum speed up before scaling increases and saturates at a certain level. Even with more than 100 LPs simulating the 100 oscillator array; the maximum speedup is only around 4 times faster than simulating using 1 LP. This is mainly caused by the ever increase in the communication that travels between different LPs. When the communication cost is scaled down to 3% of its original, the speedup shoots up to over 10 times compared to its original speedup. As a result, communication cost replaces the event processing cost.

However, if the communication cost is scaled down, the effect on the overall simulation time can be easily observed. When simulating a 50 and 100 ring oscillator array circuit, the average speedup achieved after scaling down the communication cost is around 60 times compared to only a single digit speedup without the communication cost scaling. The SDES is capable of achieving a high level of scalability, if the CuS (circuit under simulation) possesses good parallelism.

Apart from the advantages that parallel simulation brings, the worst case scenario must also be explored. At the other end of the spectrum where communication activity dominates a simulation, the performance is hard hit by the lack of parallelism in the

simulated circuit. Based on this idea, a circuit that is fully sequential is devised to examine the parallel overhead that is introduced when the simulation is parallelized.

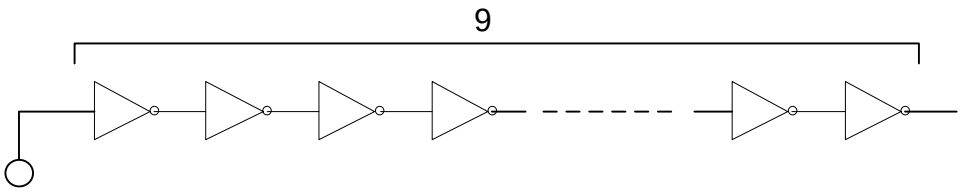



Figure 103 Sequential chain of inverters

Figure 103 shows a fully sequential system which consists of 9 inverters with a delay of 1 ns on each inverter. A source which continuously generates events at 333MHz is connected to them. This circuit is simulated under ten LPs, but they are mapped in two different settings. In one simulation setup, the nine gates plus the source are mapped to 10 different nodes, each node has 1 monitor process and 1 slave process therefore only 1 LP is available for event processing. Each holds only one simulation component. In the second simulation setup, all ten components are located in a single node, but in 10 separate LPs on this node, i.e. 10 LPs sharing 1 monitor process only. This can differentiate the boundary setup cost that is involved, as the first setup uses 1 monitor process to serve 1 LP, and the second setup uses only 1 monitor process for all 10 LPs. As all the communication, which consists of P2P packets, in the first CuS requires going through 2 monitor processes, whereas the second one only travels through 1 monitor process. By removing the total communication and total event processing time required, the cost for using different types of boundary can be revealed.

$$Boundary\ Cost = t_p - \frac{t_s}{n} - c_p$$

$t_p$ is the overall simulation time for running in parallel. $t_s$ is the overall simulation time for running the simulation with only one LP. $n$ is the number of LPs in the parallel simulation, and $c_p$ is the total communication time taken during the simulation.

| | Comm. Cost (s) | Overall Time (s) | Events per LPs | Boundary Cost (s) |
|---|---|---|---|---|
| Sequential | 0.000162 | 0.940687 | 100000 | n/a |
| Proc. Bound. | 1.100189 | 1.776259 | 10000 | 0.000058200 |
| Chip Bound. | 1.553753 | 3.4083 | 10000 | 0.000176048 |

Table 8 Boundary cost test result

The simulation is based on 10 different tests, and the average result is shown in Table 8. Simulation shows that node boundary is around 3 times more expensive than process boundary.

### 5.2.3 Communication Speed

Communication speed is the major difference between the SpiNNaker system and the conventional computer cluster. In the previous section, the performance scaling factor shows the theoretical difference in performance of the SpiNNaker system and the Iridis clusters. In this section, the communication performance of the underlying clusters along with the communication performance in SDES is discussed.

**Message level**

The communication speed of a *computer cluster* can be split into two parts, latency and bandwidth. The latency is the time it takes for a computer cluster to initiate a communication link, and the bandwidth is the speed of a communication link that lies between two nodes. The higher the bandwidth gets the lower the message transfer time. In order to determine the value for both of these values in Iridis, Ping-Pong like message passing mechanism was established between different sets of cores to extract both bandwidth and delay information.

```
if (rank==master) {
        start = MPI_Wtime();
        for (i=0;i<cnt_times;i++) {
                MPI_Send(&send,msg_length,MPI_BYTE,slave,0,MPI_COMM_WORLD);
                MPI_Recv(&recv,msg_length,MPI_BYTE,slave,0,MPI_COMM_WORLD,&status);
        }
        end = MPI_Wtime();
        log1 = end - start;
}
else if (rank == slave) {
        for (i=0;i<cnt_times;i++) {
                MPI_Recv(&recv,msg_length,MPI_BYTE,master,0,MPI_COMM_WORLD,&status);
                MPI_Send(&send,msg_length,MPI_BYTE,master,0,MPI_COMM_WORLD);
        }
}
```

Figure 104 Ping-Pong like communication speed measurement code

The code in Figure 104 measures the time it takes for a specific length of data (msg_length) to be sent over the network. The theoretical relationship between the size of a message and its communication cost is shown in Figure 105. There is an offset time before any data could be transferred over the network. By sending messages using different message size, the gradient can be measured and the offset time value can also be evaluated by tracing back the origin of the trend line, which is the latency of sending a message. The gradient is the bandwidth of the communication link. Higher bandwidth leads to steeper angle in the relationship graph.

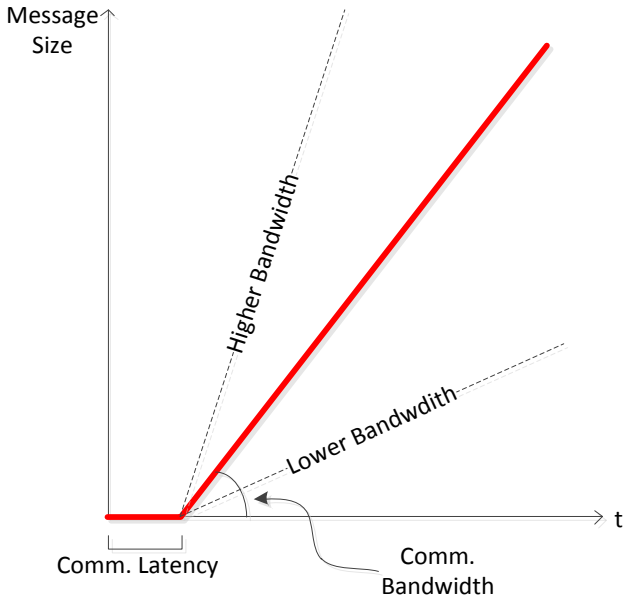
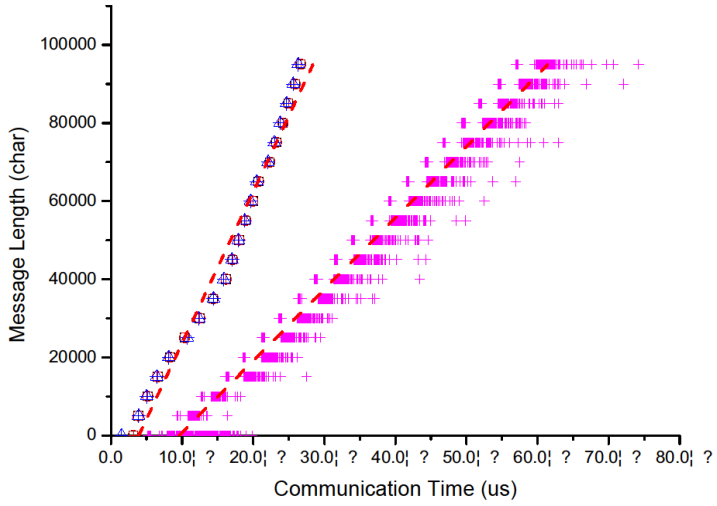



Figure 105 Communication cost vs. message size

The result of this Ping-Pong test shows that there are two latencies that exist in Iridis. If both ends of a communication link lie in the same node, the communication delay is around 4 μs, but the latency suddenly shoots up to around 11 μs when two nodes try to communicate with each other. This is the minimal time required to complete a transfer of message in MPI on the Iridis cluster. The test results are shown in Figure 106.




| | | Average | Max | Min |
|---|---|---|---|---|
| Same Node | Latency (s) | 0.0000037877 | | |
| | Bandwidth(bytes/s) | 3,690,036,900 | | |
| Cross Node | Latency (s) | 0.0000107793 | 0.0000138456 | 0.0000076599 |
| | Bandwidth(bytes/s) | 1,888,529,008 | 2,235,025,330 | 1,464,748,389 |

Figure 106 Cluster communication bandwidth and latency test result

The test is based on 1000 processor-cores, which consists of 250 quad-core chips, in 84 nodes. This test picks one of the cores as origin and tests the communication time for various message sizes with each quad-core chip in the rest of the system. The result can be summarized using two different trend lines. The speed of communication within a single node clearly stands out from other communication speeds. The cross node communication speed slopes sit within a range between trend lines in Figure 106. Derived from the graph, the latency and bandwidth can be calculated and they are shown in the table in Figure 106. The latencies for on-node and off-node communication are 4 µs and 11 µs on average. And the on-node and off-node bandwidths are 32Gbits/s and 16Gbits/s.

The on-node communication bandwidth is greater than off-node communication, where a transfer of a message is executed as a movement of memory data. The time taken for Iridis to transfer 4 bytes of data is in the magnitude of nanoseconds which are negligible in comparison with the initiation latency. The relationship between the number of cores and the average communication time in Iridis is shown in Figure 107.

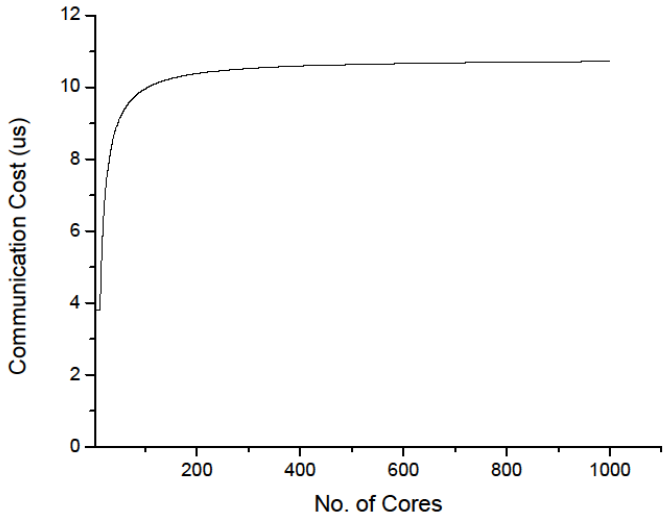



Figure 107 Relationship between average communication time and number of cores on the Iridis platform

This relationship is calculated based on the assumption that each core sends a message to every single core in the system, including cores on other nodes. The speed is the average of all the communication links. The curve gradually shifts from the average cost of an on-node communication to an off-node communication as the number of cores involved increases.

In contrast, SpiNNaker only needs 0.1 μs for a packet, which contains 32-bit header and 32-bit payload, to pass a node. If the distance between two nodes is 10 hops, the communication time for a packet to transfer from one to another is 1 μs. As there are 18 cores on SpiNNaker, the relationship between the number of cores and the communication time can be shown (Figure 108).

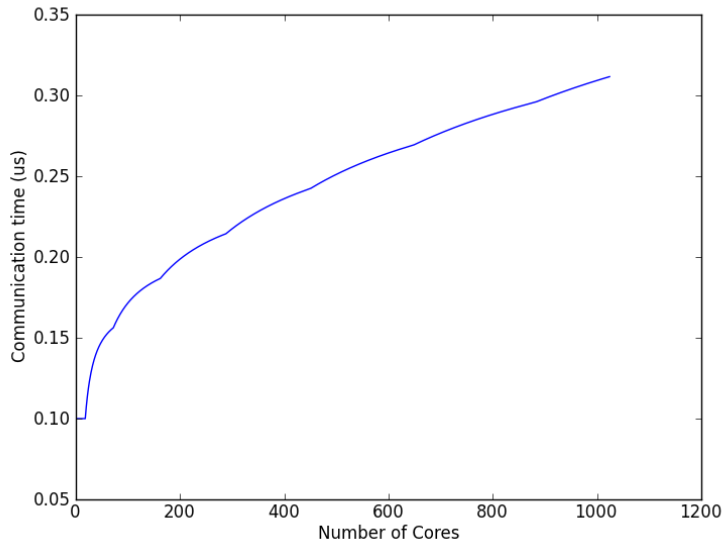



Figure 108 Relationship between average communication time and number of cores on the SpiNNaker platform

Dividing the communication cost in Iridis by the cost for SpiNNaker will produce the communication scaling factor across the 1000 core range. Within this range, the scaling factor ranges from 34.66x to 61.65x. As a result, a conservative *scaling factor* of **34 times** on communication cost can be applied when estimating the communication time for the SDES after porting to the SpiNNaker platform.

The worst-case communication delay in a square SpiNNaker node array, which is assumed to be the connectivity layout for SDES, is *two third* of the length the edge of the square. This is proven empirically in tests based on array sized from 10 x 10 to 1000 x 1000. The test sets the origin to the centre of a square array and fills each block of the array with the minimum distance to the origin. The result is shown in Figure 109. In this figure, the incremental distance is represented by a colour shift from cold colour (blue) to warm colour (red), so the dark blue spot has the lowest latency, and the dark red spots have the worst latency. By averaging the distance between the origin and all other nodes, the average communication time can be obtained, which is shown in Figure 108.

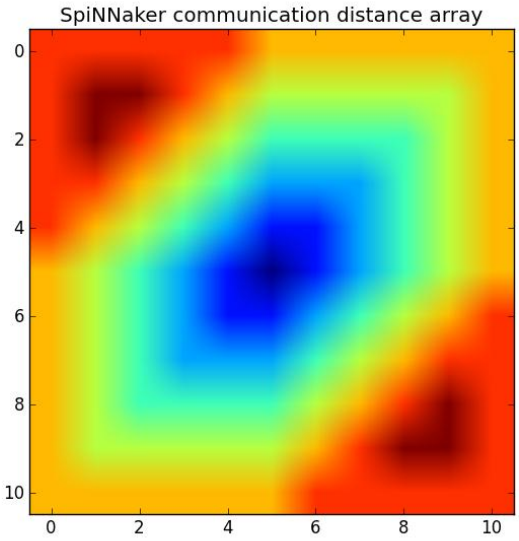



Figure 109 Communication distance from the centre of an 11 x 11 SpiNNaker array

**Event Transfer Speed**

The basic message passing speed has been evaluated. The speed of event transfer is a more abstract concept. There are two parameters to look for in this section. One of them is the *event transfer speed*, and the other is *event pass-through speed* at the monitor process. The first can be evaluated by using a similar method that evaluates the message passing speed at the lower level, but instead of using messages, the test uses events as their communication unit. The simplest way of determining this factor is to use two gates and pass events between them. One way of doing this is to use an inverter plus a buffer and map them to two individual LPs. In this test, the two LPs are situated in two separate nodes or one node with two LPs. An event needs to travel via a single or two separate monitor processes.

Subsequently, by determining the overall simulation time differences between the two tests, the second parameter, which is the event pass-through speed at the monitor process, can be measured. The time difference is caused by the extra processing time required by the additional monitor process, which is involved in the simulation. One of the unique features of the SDES is the dedicated monitor process. The events that arrive at a node must go through the monitor process before being redistributed to local LPs. This is due to the limitation of the P2P packets, which can only be delivered to a target

monitor process. The speed of an event passing through a monitor process is a vital parameter of the SDES simulator.

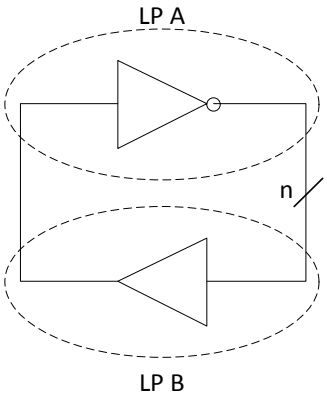



Figure 110 Simulator level communication cost test circuit

Using the test circuit in Figure 110, varying the width of the bus changes the amount of data transferred. Because the circuit is implemented in Bench file format, the bits are split into single bit gates, which are capable of generating individual single bit events. In the message level speed, there is a static cost which is the message initiation cost. At the simulator level there is also a static cost which is the parallel control overhead. A test which varies the width between 10 and 200 bits with a fixed target time of 10,000 ns is carried out. This forces the simulator to send and receive from 100,000 up to 2,000,000 event messages without modifying the boundary condition. The event transfer speed is shown in Figure 111.

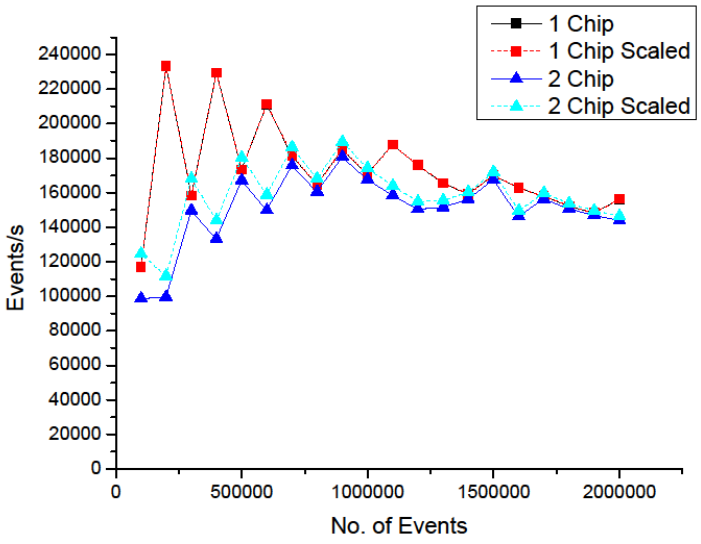



Figure 111 Event transfer time at simulator level

The communication time when employing a single node is negligible, as shown in Figure 111 as the black and red lines overlaps each other. Scaling the communication time has no effect on the event transfer speed, but a moderate difference is produced when applying the scaling with two gates on two nodes (blue and cyan). The average event transfer speed is 160K events per second. There is a clear difference in the overall simulation time between the two tests. This is the time that the additional monitor process costs to process the event. Take the average difference between the two tests, and dividing it by the number of events gives the average time taken at a monitor process. Based on the graph, a monitor process takes 2.8μs to re-route an event to a local LP.

### 5.2.4 Dynamic Partitioning

In this section, the functionality and performance of dynamic partitioning are tested. The first experiment verifies the functionality of dynamic partitioning algorithm employed in SDES. The subsequent circuits focus on the performance of dynamic partitioning.

Before this is evaluated, the correctness of the dynamic partition algorithm needs to be verified. For this purpose, a 7 stage clock generation circuit is employed. As an initial start-up condition, all seven inverters are situated in a single partition. When they are

put on seven LPs during a simulation, the ideal result would be each LP holds an inverter. Because the number of gates and the number of LPs are perfectly matched, the dynamic load balancing algorithm should evenly distribute the gates onto all the LPs. This creates an ideal balanced condition. The result of this test is shown in Figure 112.

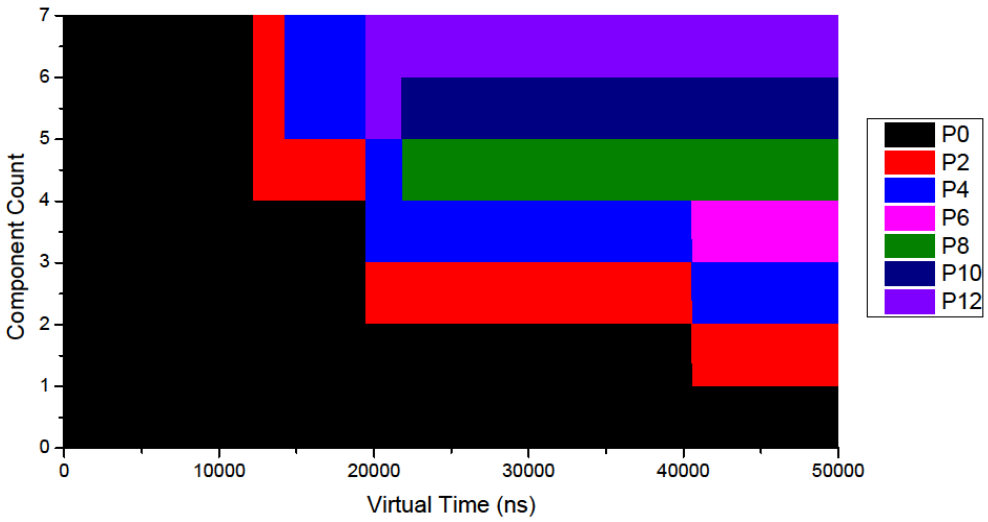



Figure 112 Component distribution vs. virtual time

As shown in Figure 112, the seven components are crowded in a single LP at the start of a simulation. Gradually as simulation proceeds, the load is spread out evenly throughout the system, where for every DLB each LP splits the load into two partitions. Through this cascading splitting process, the simulation quickly reaches the equilibrium state. This proves that dynamic partition is functionally correct.

The order of redistribution shows how the dynamic rebalancing works in SDES. The order in which the components are spread in the case of Figure 113 follows the process ID 0 -> 2 -> 4 -> 12 -> 10 -> 8 -> 6. This order shows the directions of the wave of components that is moving within the mesh. It does not imply that the movement of a single component follows this order. The batch size of each movement may be different. The detailed component movement is shown in Figure 114. The order may seem odd at first. However it strictly followed the connectivity of the processors. As a reminder, the mesh is organized in a square matric shown in Figure 113.

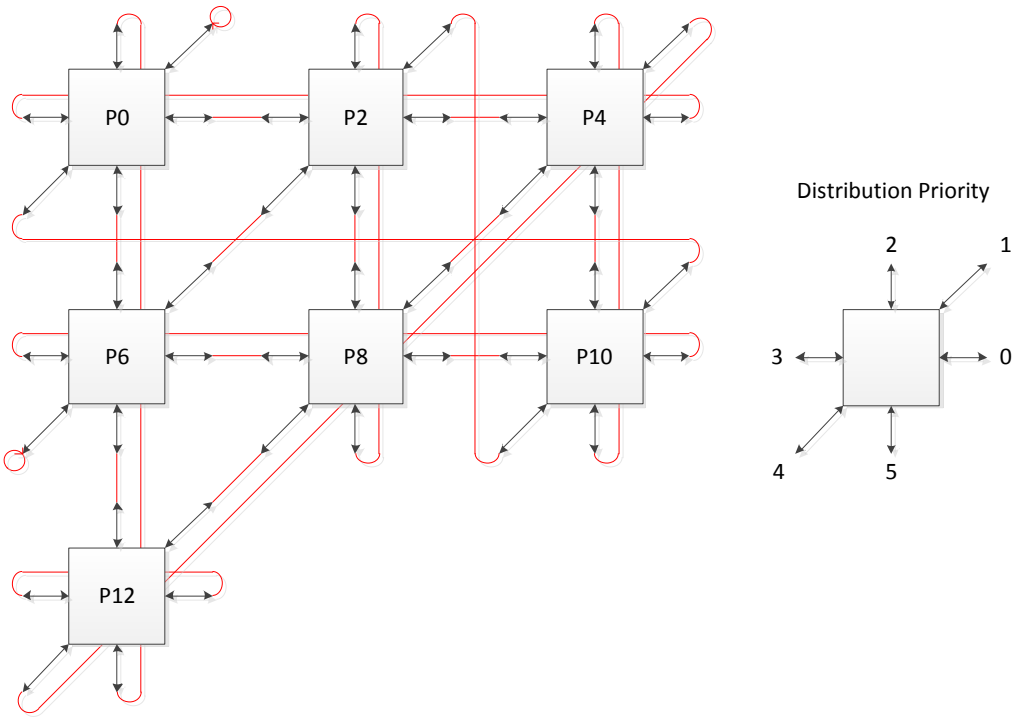



Figure 113 Incomplete square mesh connectivity

Initially, all the 7 components resided in P0. The process gradually built up the number of events and eventually reached the minimum workload level to trigger a dynamic load balance. This minimum workload level can be specified in SDES, in this case 1000 events. When dynamic load balance is active, P0 tries to propagate in the order shown on the right hand side in Figure 113, which shows P2 (priority 0) has the highest priority. So does P4 to P2 in the subsequent transfer.

The following component transfer is simulation dependent. A race condition is formed between P0 and P4, as both of them have more than one component on hand. In this particular test, P0 locked P4 first and performed the component transfer with P12.

The next stage both P12 and P4 trigger load balance at nearly the same time. But the LPs, which the components within P12 and P4, are moving to are different, P8 and P10 respectively. The reason why they can be performed concurrently is explained by drilling down to the specific component mapping during the DLB (Figure 114).

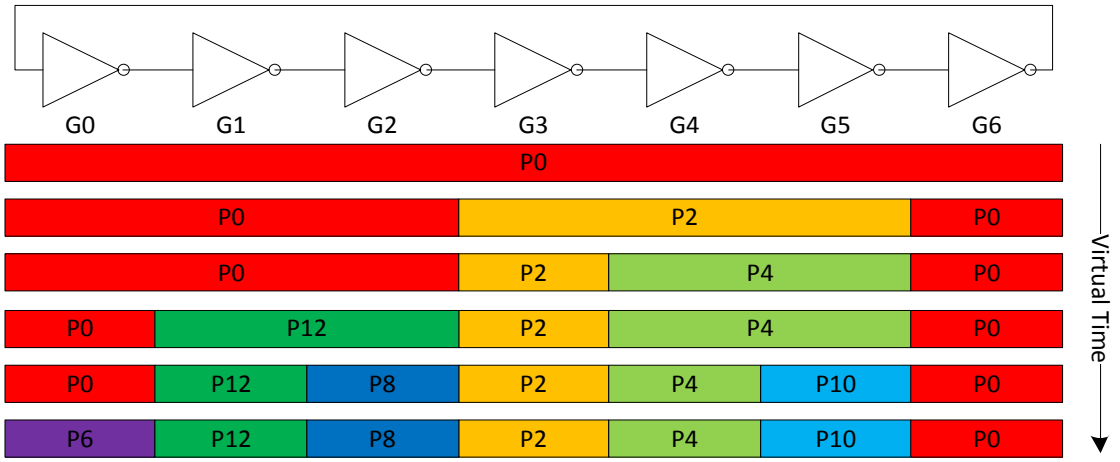



Figure 114 Component mapping during DLB

By the time P12 and P4 both have two components on hand, the set of predecessors and DLB targets are completely different. As a result both of these processes are able to freeze their predecessor and DLB targets without interfering with each other. Although P0 has two components at the same time periods, P0 requires P4 to be frozen in order to carry on the process, but P4 already has its own plan. This behaviour shows the DLB follows a strict component transfer rule.

The final spare capacity lies in P6 which ranks the lowest priority in transfer for P0. This is the only available dynamic balance opportunity in the system, which P0 carries out after a period of time. In short, the transfer of workload follows the specification and structure of the infrastructure.

Beside the position accuracy, another parameter that defines the DLB is the *minimum workload boundary* (MWB). The same 7- stage clock generator circuit example can be used as a simple yet elegant way of finding this. In the test above, the MWB is set to 1000 events. This means all transfers must occur after the local workload reaches this level. This can be illustrated by looking at the integral of the size of local queue. The local relative local workload compared to the workload at the last load balance is shown in Figure 115.

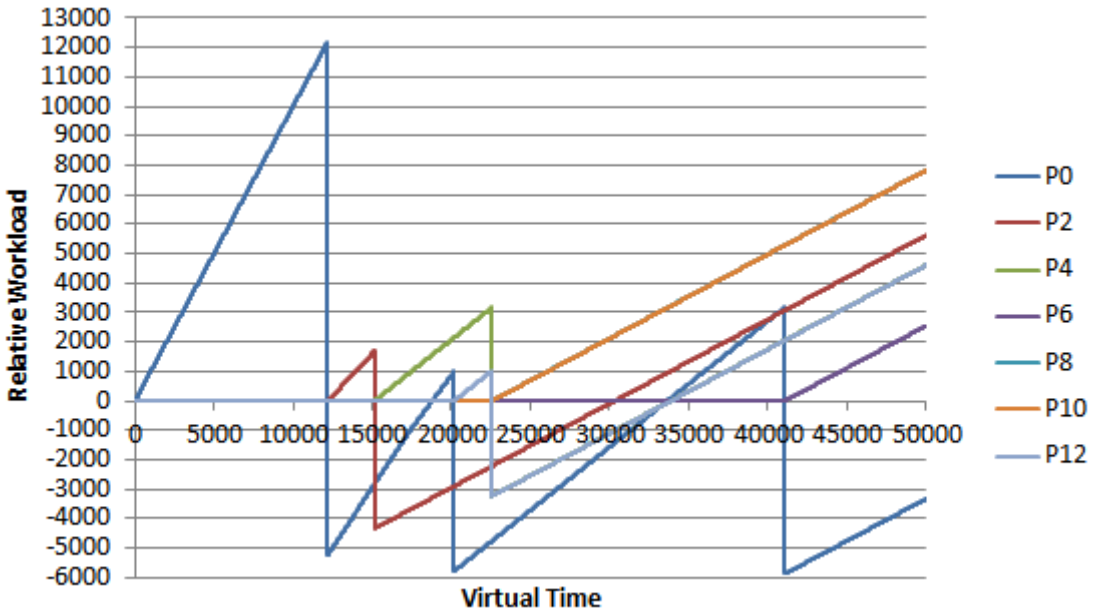



Figure 115 Relative workload compared to last load balance level

As shown in Figure 115, each DLB operation resets the local workload level to a negative value and hence inhibits the source LP from performing further DLB. All the DLB is carried out above the MWB, which is defined as 1000 events, which illustrates that SDES does obey this MWB during its simulation.

In Figure 115, the first peak of P0 is at 12000 events instead of the MWB, which is set to 1000 events. This is due to another parameter, which is the *DLB frequency*. In the initial partitioning setting, all the components are located in P0. No synchronization or communication is required with any other LPs. This is where the DLB frequency parameter comes in to pause the local simulation process, and starts to redistribute the workload accordingly. Hence, the first peak is not in control of the MWB, because of this particular initial partitioning setting.

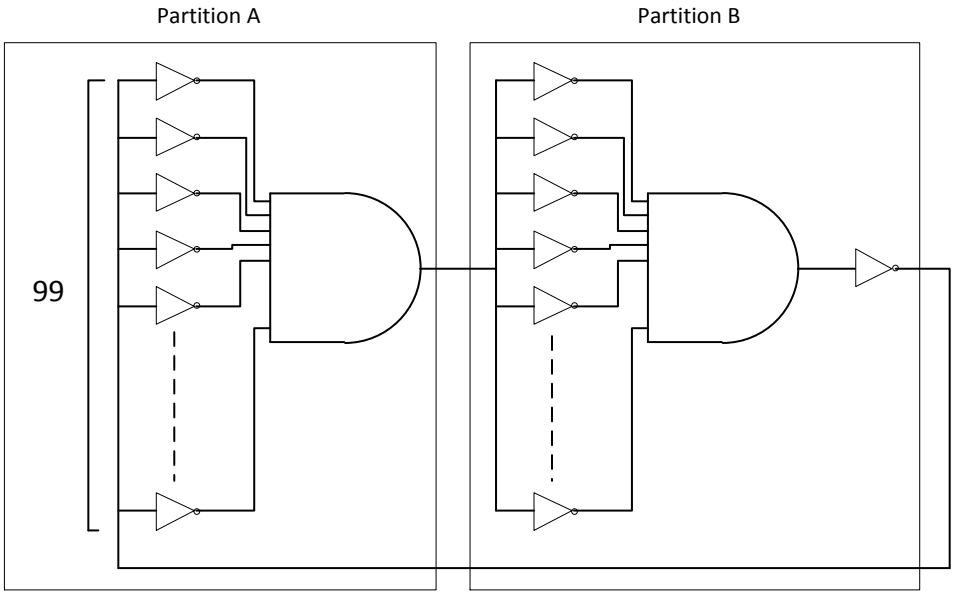



Figure 116 Two AND gates in an inverter circuit

A further test on dynamic load balancing is carried out on another special case: *two 99-input AND* gates are connected to form a clock generation circuit. Under static partitioning, this circuit is partitioned into two parts as shown in Figure 116. In the long run the workload is balanced to 1%. In addition, any temporary workload difference between the two is always 100 events. This test checks whether DLB reacts on this small workload imbalance, as well as short term workload imbalance.

When the MWB and maximum workload difference ratio (MWD) are both set to zero, a Zig-Zag effect is expected. Because both parameters are set to zero, SDES has zero tolerance to any workload difference between the two LPs. The component count difference between the two LPs during a simulation is shown in Figure 117.

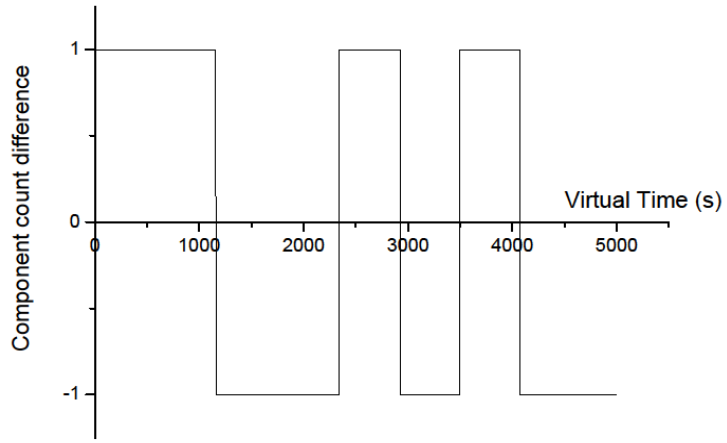



Figure 117 Component count difference between two LPs

By increasing the MWD to or above 1% without changing MWB, no DLB is carried out by SDES, because the workload difference between two LPs is not enough to trigger a DLB. The short term workload difference does not trigger any DLB.

Another test is carried out by increasing the MWB and resetting MWD to zero. This incremental change of MWB is able to delay the component transfer to later stages. The pattern of DLB should change accordingly. When MWB and MWD are set to zero, the relative workload compared to the last load balancing level is shown in Figure 118. The uniform gap between load balancing operations is caused by the DLB frequency limit imposed by SDES to avoid aggressive DLB operations, as well as leaving computation for the monitor process for other operations, such as redirecting messages. The DLB frequency is set in wall time, which can be configured by user. (By default, it is set to 3 seconds) In short, every time there is a DLB transition, it means a wall time 3 second has passed during the simulation. Since the MWB and MWD are both set to zero, every time it checks the workload difference, the transfer request is always valid, due to the imbalanced nature of the circuit under simulation.

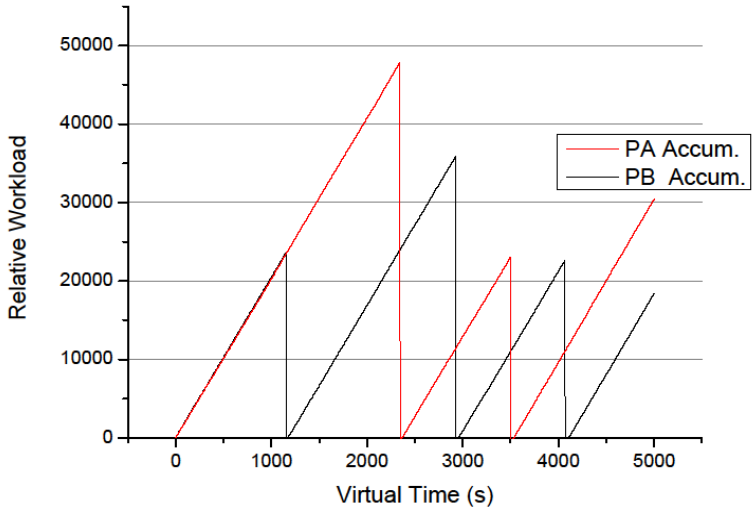



Figure 118 Relative workload with zero workload difference tolerance

By increasing the MWB to 80,000, and leaving the MWD at zero, a DLB is only being carried out when one of the LB reaches the 80,000 level. The relative workload under the new constraint is shown in Figure 119.

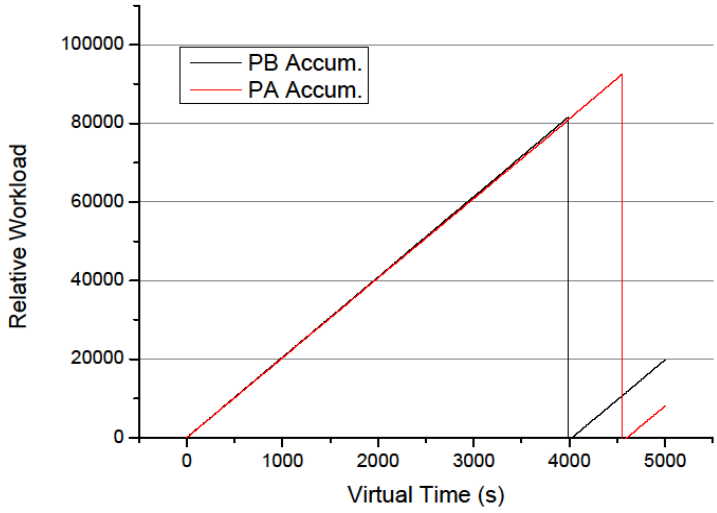



Figure 119 Relative workload with 80000 MWB limit

When both LP reach the 80000 MWB limit, PB holds 1% more workload than PA, as a result, PB starts the DLB operation first. After the wall time gap imposed by the DLB frequency parameter, PA finds that it has 1% more workload than PB, and the single gate is being shifted back to PA right after the gap ends.

In the last circuit, the two 99-input AND gates are testing the DLB response to either small or short term workload imbalance. The result shows SDES is able to detect small workload differences between the two processes, but it is not sensitive to short term workload imbalance. In the next example, a long term workload imbalance case is tested and discussed.

Another extreme case which can test DLB function is a delayed oscillation circuit which is shown in Figure 120. Each clock block consists of 50 333MHz clock generators. This will force the DLB to balance the workload constantly to balance the workload between LPs. When the first clock block is triggered, DLB tries to spread this heavy computational block into other LPs. However, when the second clock block is triggered, DLB has to readjust the workload among the 5 LPs. In the final state, the workload should be balanced out, although the partition may be different from initial partition in all LPs.

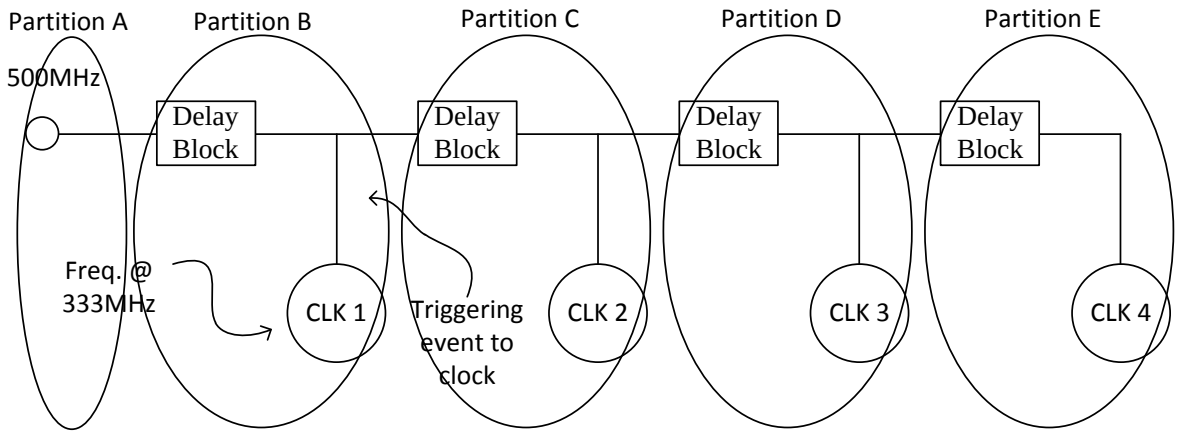



Figure 120 Delayed activation of oscillators

The delayed oscillation circuit is constructed using hierarchical bench file format. Each of the components in this circuit is created as cells and is later constructed together to form the entire circuit. Part of the code is shown in Figure 121. In the partial code, an edge detection cell is specified and is later involved in the main circuit. The subsequent cells, such as the counter and oscillation circuits are all specified before this partial code. The circuit parser stores the specifications of these individual cells in a temporary

library. When they are called in another cell or the main circuit, the parser automatically adds the cell to the directed graph of the target circuit. The edge detector and counter circuit forms a frequency divider. The array of twenty oscillator circuits forms the clock block that generates the heavy computational workloads.

```
……

CELL EDGE_DETECT (IP,OP,CLK,nRESET)
INPUT (IP)
INPUT (CLK)
OUTPUT (OP)
INPUT (nRESET)
DFF DFF_1 (IP,Q1,CLK,nRESET)
DFF DFF_2 (Q1,Q2,CLK,nRESET)
OP = XOR(Q2,Q1)
END CELL

INPUT (initiate)
INPUT (nRESET)
INPUT (CLK)

EDGE_DETECT ED_1 (initiate,IP_1,CLK,nRESET)
COUNTER4 CNT_1(IP_1,W_1,CLK,nRESET)
OSC OSC_1_0(W_1,CLK_1_0)
OSC OSC_1_1(W_1,CLK_1_1)
OSC OSC_1_2(W_1,CLK_1_2)
OSC OSC_1_3(W_1,CLK_1_3)
OSC OSC_1_4(W_1,CLK_1_4)
OSC OSC_1_5(W_1,CLK_1_5)
OSC OSC_1_6(W_1,CLK_1_6)
OSC OSC_1_7(W_1,CLK_1_7)
OSC OSC_1_8(W_1,CLK_1_8)
OSC OSC_1_9(W_1,CLK_1_9)

……
```

Figure 121 Bench file description for delayed oscillation circuit (partial)

In theory the DLB tries to rebalance the workload at least 4 times due to the 4 clock blocks being introduced during the simulation period. The trigger time of these 4 clock blocks are 203ns, 1363ns, 2683ns and 5283ns respectively. The balancing work starts shortly after the activations of new computational workload are triggered. The event distribution is shown in Figure 122 in the form of event list size. The first graph shows the accumulated event list size across the system, and the second graph shows the size of event list in each individual LP. It shows that DLB is dealing with the new workload by continuously shifting the workload towards the idle LPs. When clocks at the later stages are activated, the workload is shifted back to the original LPs, which is expected.

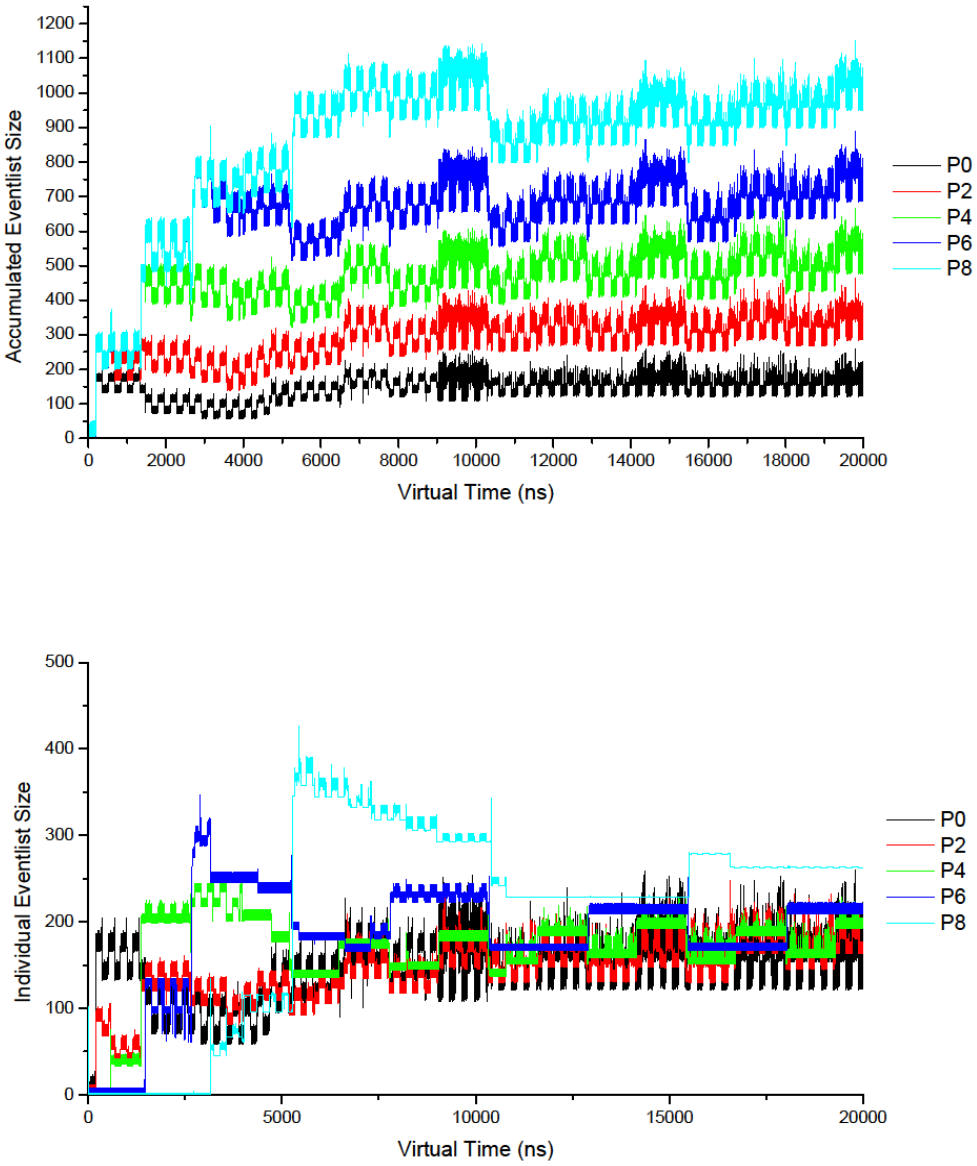



Figure 122 Event list size for delayed oscillation circuit

The spikes of activities, generated by activation of clock generation blocks, can be seen from the individual event list size graph. The spikes can be seen with P0 and P2, P4, P6, and P8. Because the system is gates distributed in the initialization stage, the first clock generation block is partly mapped to P0 (Black). As a result, the first spike can be seen from both P0 and P2 (Red).

Every time a clock generation block is activated, there is a large gap of workload between different LPs. Take the activation of clock generation block at P8 as an example. The size of event list in P8 jumped from around 150 to 400. The DLB gradually spread the workload across the system. Shortly after 10000 ns, the sizes of event lists across LPs fall into the same cluster, around 150 events. The drop in overall event count, at around 10000 ns, is due to activities that lie in the delay block circuit.

This test shows DLB is able to adjust the workload constantly to smooth out the workload difference among LPs. It is able to deal the large workload differences in LPs and limit the workload in LPs around a small region.

### 5.2.5 DES and FIR circuit

Apart from the test cases manually created above, the SDES performance is also evaluated against real digital systems. In this section, a DES circuit and an FIR circuit are tested on SDES. The effect of dynamic load balancing, the predicted performance on the SpiNNaker system, and the effect of multiple LPs on a SpiNNaker node are presented.

The SDES simulates a DES codec circuit written in behavioural VHDL, which is converted into SDES directed graphs. The DES circuit, when converted into CLG, consists of 5503 gates. As this is the first real digital application test circuit, the speed of simulation of the SDES can be used to compare with the original deadlock avoidance technique, and understand if the SDES can outperform the original deadlock avoidance technique. The original deadlock avoidance technique data is tested on the Iridis platform without the scaling of communication speed. Based on this idea the test is put together, the result is shown in Figure 123.

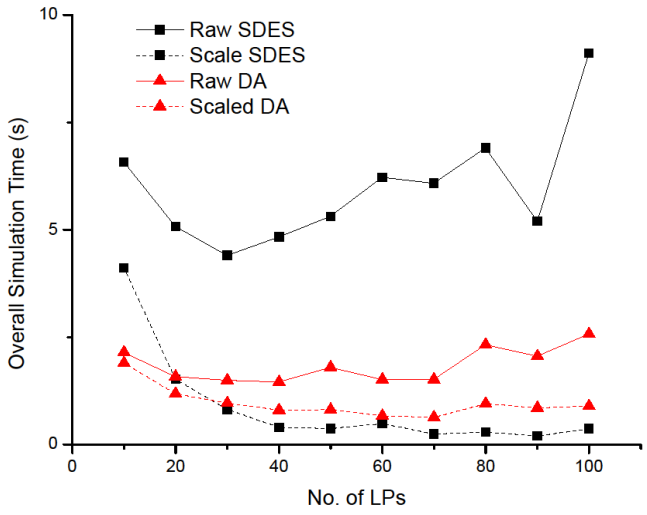



Figure 123 The SDES vs. deadlock avoidance technique

The test is based on 5 LPs per node for the SDES and the same number of LPs for the original deadlock avoidance (DA) technique. As can be seen from Figure 123, without the scaling of communication cost, the performance of SDES is worse than the original deadlock avoidance technique. However, after the scaling down the communication time in both cases, the SDES outperforms the deadlock avoidance technique by a decent margin when the number of LPs exceeds 20. When more than 30 LPs are engaged in the simulation, the overall simulation saves around 50% compares to the original technique.

In the last test, the number of processors only reaches 100 LPs. This is due to the limited timing analysis processing power. During the simulation, each LP is generating around 2 million time sampling points, each sampling points has a 64-bit absolute time, which translates to 128Mbits per LP. 100 LPs generate 10Gbits of data, which then forms an event timeline for further analysis. If it goes further, the program is likely to overflow the memory for a single processing node. And without the timing analysis, simulation makes little difference between the SDES and the original deadlock avoidance technique, as demonstrated.

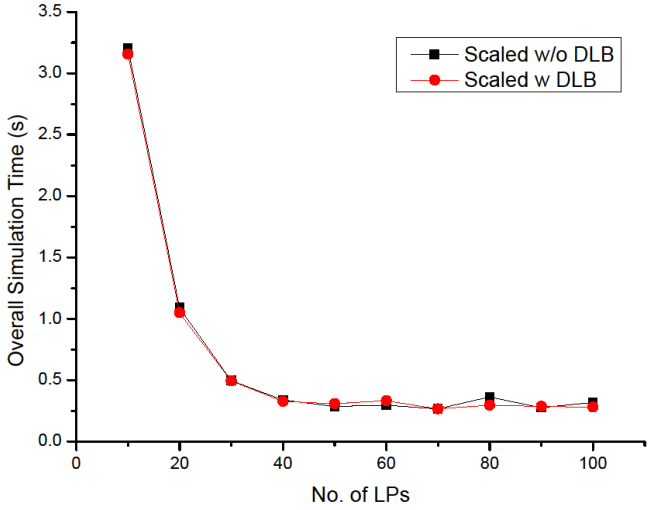



Figure 124 DLB effect when simulating DES circuit

The effect of dynamic load balancing is demonstrated in Figure 124. DLB has little effect on the overall performance of the simulation, which is due to limited amount of workload to it is a small circuit. The dynamic partitioning technique has a clearer effect if the absolute workload difference between the LPs is more significant. The total number of events in the system is only around 250K, which is distributed among 100 LPs. That leaves only around 2500 events in each LP: it is hard for DLB to make an impact on the performance. In order to increase the parallelism, five DES encoding and decoding pipeline circuits are laid out to form a 55K gates circuit. The layout of the circuit is shown in Figure 125.

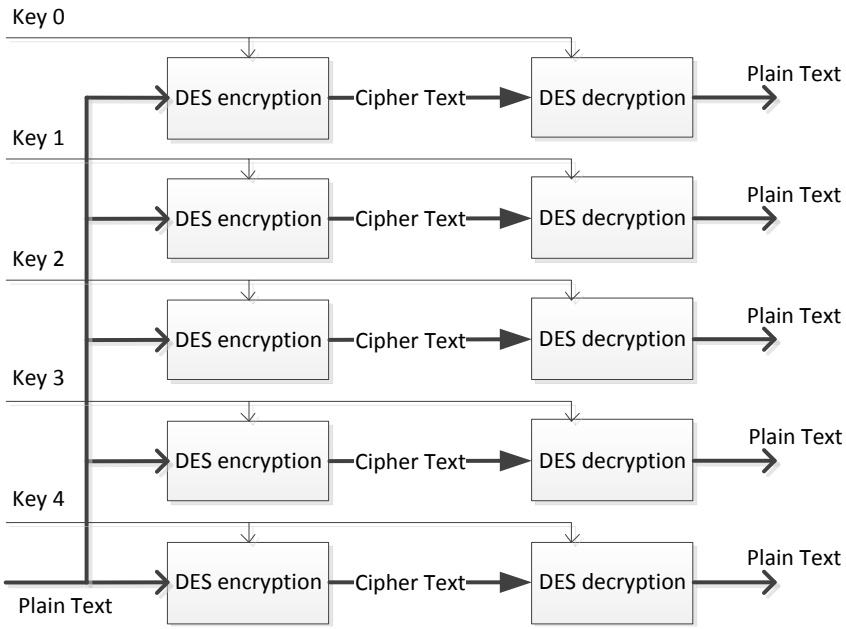



Figure 125 Five DES encoding and decoding pipeline circuits

This DES circuit layout enables the system to validate itself. A same piece of plain text is fed into the five encoding and decoding pipelines and each pipeline employs its own key. Although they are ciphering the same set of plain texts, the intermediate results are very different from one another. The outputs at the end of the pipelines are the same as its inputs. This easily validates the correctness of the simulator. The circuit is tested for 15000 ns in virtual time, which gives the simulator time to pass two blocks of plain text through the system.

The simulation result waveform is shown in Figure 126. The results are grouped in three sets. The first set of signals is the plain text input to the codec pipeline. The second set of signals is the cipher text generated by the DES encryption circuit. This cipher text is fed into the DES decryption circuits which produce the third set of data. The final set of data is the final output of the whole circuit. This result should be identical to the plain text input two stages into the history. In Figure 126, the top five signals are the plain text input, the middle five are the intermediate signal between the two layers of DES circuits. As each pipeline uses a different encryption key, the intermediate signals are very different from each other. However, the final outputs are identical as expected.

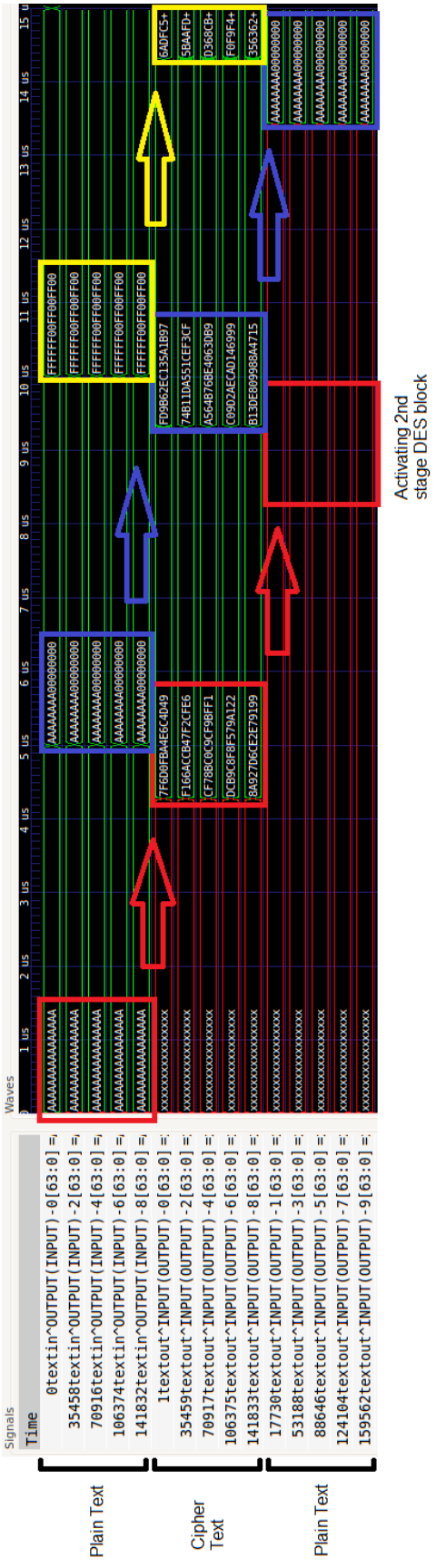



Figure 126 DES simulation result

Throughout the simulation, each DES block generates around 250,000 events, so the overall circuit provides 2,500,000 events for the simulator. The effect of DLB is shown in Figure 127. The graph shows a similar performance before and after applying DLB. Most of the time during the simulation, it shows a modest improvement in the overall simulation. However, the effect of DLB can be seen much clearer in the individual LP level.

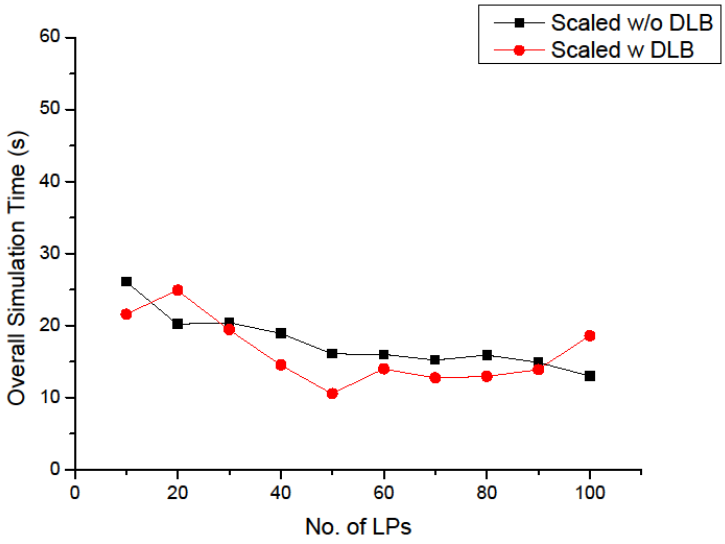



Figure 127 Effect of DLB in DES array simulation

In order to demonstrate the workload shift over the overall simulation clearly, the same simulation is tested using 9 virtual nodes; each node has only 1 LP. This time the workload in each individual LP is recorded. The graph before and after applying DLB is shown in Figure 128 and Figure 129 respectively.

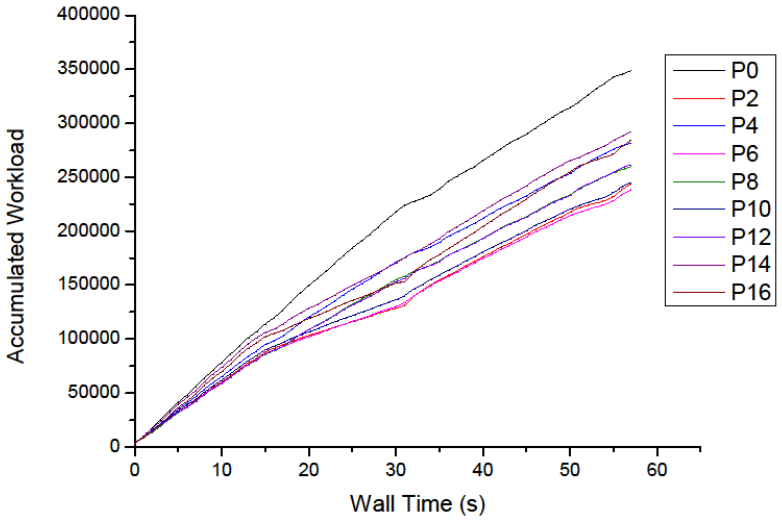



Figure 128 Accumulated workload graph before using DLB

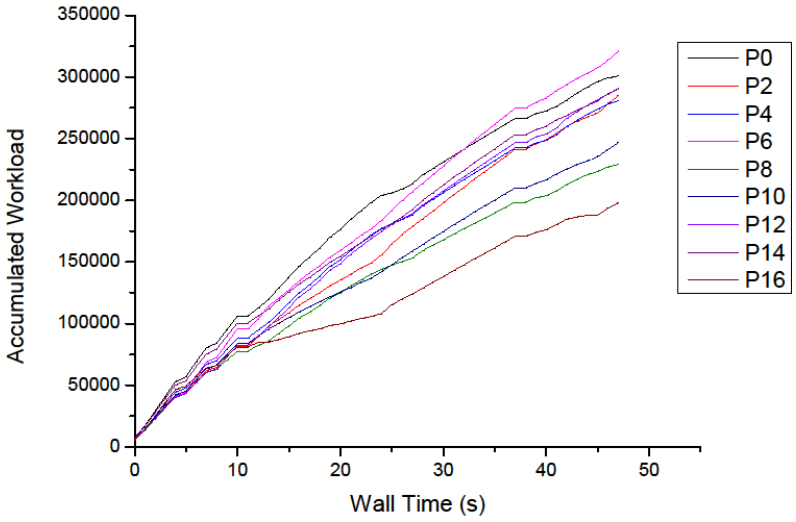



Figure 129 Accumulated workload graph after using DLB

Before applying DLB, P0 has more workload than any other LP, which becomes a drag on the overall processing power. After applying the DLB, the workload in P0 is distributed among other LPs. The average event execution speed increases from 42,331 events per second to 50,869 events per second, hence it reducing the overall simulation time as shown in the X-axis of the last two figures.

Another interesting point shown in the accumulated workload graph is that even though the DLB is employed, the workload is not fully balanced. This is because, the nature of the DLB employed in SDES is a fully distributed model. As a result, the workload difference can only be detected regionally. It does not guarantee the global workload balance. In addition, although the heavier workload at P0 is distributed other LPs, the LPs with lighter workload do not seem to be filled with enough workload. This is because to the type of DLB mechanism in SDES is sender-initiated-diffusion. The LPs with higher workload initiates the DLB with neighbouring LPs, where the LPs with less workload tend to be receiving workload from other LPs.

Another general circuit, the **FIR circuit**, is put under test using SDES. The FIR circuit is constructed using behavioural VHDL and goes through the process of MOODS synthesis, transformed into MOODS library digraph and finally converted into logic component directed graph. The simulation results of SDES are proven correct by simulating MOODS synthesized model of FIR using ModelSim. The difference lies in the timing information. Because the synthesized model went through another round of conversion through circuit converter, which turns the MOODS library components into basic gates, the propagation delay is distorted by this conversion operation. However the end results are identical. The main purpose of simulating FIR circuit is proving that the tool chain is capable of other circuit types other than DES.

The FIR circuit in the portfolio has 16 taps, which means it has 16 multipliers. The 16-bit multiplier generates 1540 gates and 16 of them generate 24,640 gates in total. A test is carried out to evaluate the performance of SDES under various numbers of LPs, and the effect of DLB when using different numbers of LPs during a simulation. The overall simulation generates around 359K events.

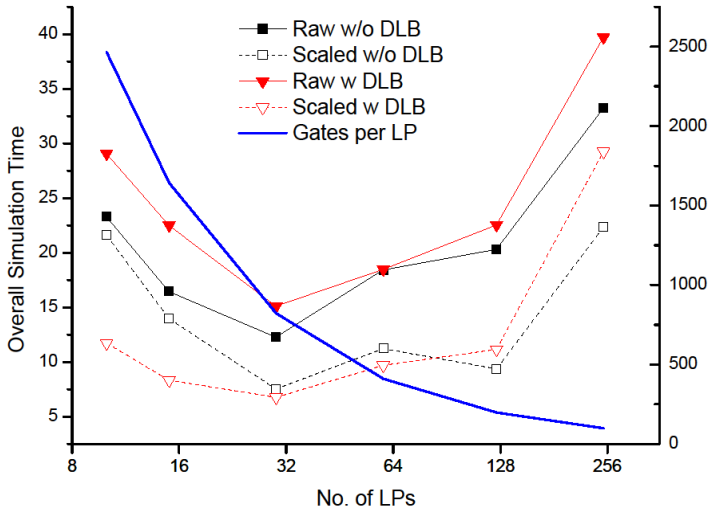



Figure 130 FIR simulation result over a number of LPs

The number of LPs in the test shown in Figure 130 ranges from 10 LPs to 250 LPs. The upper limit is capped at 250, because when the number of LPs over 300, the time analysis program starts to run out of memory again. In general, as the workload on each LP (indicated as Blue) decreases, the performance gap between the DLB simulation data with and without communication scaling (Red lines) shrinks. As the axis for the number of LPs in Figure 130 is logarithmic , the workload quickly drops off from 2500 gates per LP to 10 gates per LP. The performance peaked at around 1000 gates per LP. This indicates that the incremental benefit of both additional computational power and DLB operation is outclassed by the control overhead of simulation.

The figure shows when the LPs are less than 30 LPs, the scaled performance shows that DLB can bring a moderate improvement for the performance. However at the other end of the spectrum where the number of LPs is over 125, the activation of DLB actually let the performance deteriorate. This is due to the additional workload update required by DLB has overwhelmed the benefit that DLB bring to a simulation. The workload update cost across different number of LPs is relatively stable, as the update only required to be performed within a virtual SpiNNaker node as well as six neighbouring virtual node. It is irrelevant to the size of the overall system. However, the benefit of introducing DLB fades as the number of LPs increases, this is because, and the number of gates that a LP

holds reduces. Along with the number of gates, the benefit of moving heavy workload also fades, because, less and less workload is able to generate enough processing time difference, before and after a DLB.

# Chapter 6 Final Comments

## 6.1 *Conclusions*

First of all, this project has discovered a range of properties about how a generic simulation system will behave and perform under the new SpiNNaker platform. This is carried out by emulating the behaviour of the SpiNNaker in a conventional computing cluster, Iridis, and the performance estimation is carried out by scaling down the communication cost in Iridis. Based on the experiments targeting partitioning, event processing engine, communication system and general circuits, we can draw several conclusions about the properties of the SDES system.

By porting a simulation from a conventional platform to the SpiNNaker platform, it can potentially reduce the overall simulation time by a great margin, as tests shown in ring oscillator and DES circuit. In a real life circuit like DES, an improvement in performance around 50% is caused by reduction in communication time. As discovered in this project, a factor of 34 in communication speed can be potentially achieved by porting the simulation from conventional cluster to the SpiNNaker platform, when the number of LPs involved in simulation is less than 1000. This is the minimum average communication speedup between the SpiNNaker and Iridis platform.

This is consistent with the main ***hypothesis*** that was proposed at the beginning of this project. The fast communication network in SpiNNaker does bring performance benefit when the number of LPs increases, but only up to a point. As demonstrated by the FIR circuit test, the initial increase in simulation speed gradually saturates at a peak performance plateau as the number of LPs involved increases. If even more LPs are involved, the performance deteriorates.

Due to the unique SpiNNaker architecture, the processing powers are naturally divided as nodes, each consists of 18 cores. The generic simulation can take advantage of this design if the simulation problem can be grouped into batches, which have tightly

coupled connections within. However, the monitor process can impose a bottleneck effect on system performance, as all the packets are required to be processed before sending them to its target process. The optimal number of LPs per virtual node is found at 5 when simulating the DES circuit. The test result indicates that this is the optimal point for generic simulation performance. If less LPs exist in a virtual node, it will increase the computational time. On the other hand the bottleneck effect will appear when there are more LPs per node.

The DLB mechanism in SDES is capable of distributing workload in fine detail. The results show the two advantages ***hypothesised*** at the beginning of this project. Firstly, the DLB is also able to shift the workload freely between processors during a simulation, as indicated in the delayed oscillation circuit test. The DLB mechanism was gradually moving workloads around to cope with the sudden increase in regional switching activity. A mixture of large and small steps it took during the process shows that this DLB mechanism is very dynamic. The DLB mechanism was able to balance the workload effectively to let the execution workload, i.e. size of event list in this case, to approximately the same level as the simulation carries on.

In addition, it has the capability to detect and shift minor workload difference between LPs, which makes the workload distribution more dynamic, as shown in the 99-input *AND* gates circuit. However, this DLB mechanism ignores temporary workload difference between two essentially workload balanced processes, as the large workload difference averages out over time. This shows the accuracy and resilience of this mechanism.

However, this DLB has a better performance when simulating a large sized network with fewer activities. The DLB system only came into effect if there is over 1K of gates in each LP in the case of simulating the FIR circuit. The FIR generates only 1.5 times the amount of events that produced by the single DES simulation, but the FIR has nearly 5 times the number of gates in the simulation. The effect of DLB can be seen more clearly in the FIR test, but not the DES circuit. When the size of DES has been

multiplied by ten times, the effect of DLB starts to kicks in, but the effect is not as clear as it has shown in the FIR case.

## 6.2 *Circuit Import Methods*

There are further improvements that can be done in SDES but due to the lack of time they are not implemented. Here are some of the key points that can be done with SDES.

The real application circuits tested in SDES are all imported from the MOODS synthesis tool. MOODS generates synthesized structural VHDL codes which needs to be converted into basic combinational gates which SDES can deal with. Additional synthesis tool may be preferable to improve the range of varieties in circuits. Take Synplify tool as an example, if a customized library is written with SDES simulation gates or MOODS library gates, the output circuit can be directly translated into a digraph that SDES can simulate with. This can shorten the tool chain and provide a fresh perspective in terms of different circuit structure.

## 6.3 *Porting for SpiNNaker*

SDES is designed to execute on the SpiNNaker platform; however, the platform is not ready on time. As a result, I can only emulate the SpiNNaker platform using computer clusters. The measurements on performance can only provide a glimpse of how parallel simulation can benefit from the SpiNNaker platform. Further work is required in order to port the program onto SpiNNaker. For example, the program loading from external computers to each individual SpiNNaker node was not written in this project. *Moreover* the memory used in SDES is around 1MB but it is still beyond the scale of SpiNNaker's CPU program memory of 32KB. The use of 128MB shared memory is required when porting SDES on the SpiNNaker platform. This will bring in additional changes to the program.

## 6.4 *The Load Balancing Algorithm*

The load balancing algorithm currently used in SDES is not efficient enough. Further improvement on selecting the group of moving components can be done to reduce the overall simulation time. The only measurement that DLB takes into account in SDES is the activity ratio between moving and all the local components. This does not take into account the extra wire cut that will be made when moving a heavily loaded component to other partitions.

There is a great room for improvement in the existing implementation of the DLB algorithm. By adding awareness to the number of wire cuts, the communication cost will reduce. Furthermore, it may also increase the parallelism within a partitioned circuit. This is because a reduction in communication links will tend to reduce the data dependences between processes. Therefore, more events can be processed before anyone of them reaches the edge of a local LP.

The DLB may be made aware of the number of necessary wire cuts needed to balance the workload between two LPs by utilising the weight parameter assigned to each individual component. This will give an approximate indication of how much communication will be generated after every load balancing operation. Combining this connection weight with the amount of increase/decrease of wire cut generated by load balancing, will improve the performance of the DLB algorithm.

## 6.5 *Data Visibility*

At the moment, performance data and simulation results can only be obtained at the end of the simulation. This requires a lengthy process of gathering data after each simulation and it cannot provide live data inside the SpiNNaker system. Clearly more can be done to improve the data visibility in SpiNNaker system. A possible way is letting the monitor process to collect live data from slave processes, and pack them together to reduce the number of floating messages in the overall communication system. However, only selective data can be transferred through live communication system. As the event and communication workload on monitor process can be saturated even without this

extra data visibility communication traffic. Switching parameters which specify the type of live data feed it requires will help solve this problem.

## 6.6 *Future Work*

As the SDES LP is capable of simulating various sized circuit, ranging from a single gate to the entire circuit, the event processing time can vary wildly depends on the length of the event list, as indicated in section 5.2.2 . In order to exploit this advantage and convert it into the advantage of the SpiNNaker platform, which is the fast communication network, it may be better to impose a limit on the LPs to contain only one component.

A nano-simulation wrapper that only holds a single gate may convert computational cost into communication cost, which SpiNNaker system has an advantage over conventional computing platform. As the simulated target in a nano-simulator is a fix and small object, the control overhead and the subsequent event processing time will be reduced. This performance merit will be obtained through faster data search and access time. These nano-simulation wrappers can be squeezed into a single LP where the LP only needs to process the communication events that are generated by nano-simulation wrappers. Moreover, the self-contained nature of these nano-simulation wrappers enables dynamic load balance to move individual components at a lower cost, as the collection of events and adjustments to boundary will be taken care of in these wrappers rather than the LP itself.

Another performance gold mine to be exploited is to fully customize the SDES to use MC packets combined with the P2P packets in the SpiNNaker system. As the P2P network can only deliver packets to a monitor process, monitor processes add another layer of packet processing delay when dealing with messages. Replacing the P2P packet with MC packet will greatly reduce the waiting time in the processing system, as it removes the processing time in monitor processes, as well as removing the bottleneck in performance when the number of LPs in a virtual SpiNNaker node increases. However this creates difficulties during DLB session, as the limited sized routing table will have to be updated along the paths of all input connections to the shifted components.

# Appendix A Performance Data on Eventlist

**Event list**

The event list is one of the key components in the simulation system. Its performance plays an important role in the overall system performance. The container used in the event list is a STL (Standard Template Library) list. There are five main activities for an event list: adding events, copying other event lists, scanning through an event list and reading all the events out of an event list. The time taken to execute these activities is expected to grow as the number of events held within an event list (i.e. the size of an event list) increases.

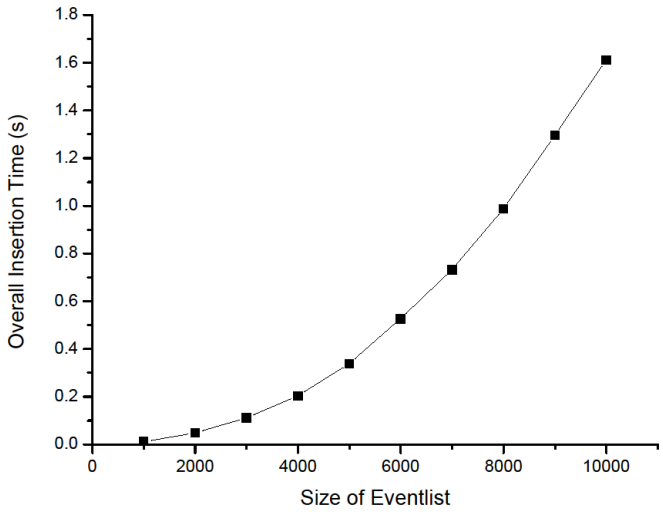



Figure 131 Event list performance on adding events to an event list

The execution time in relationship with the size of an event list is shown in Figure 131. This is tested by inserting different number of events to form an event list. The sizes ranged from 1000 to 10000, with a step of 1000, and 10 data points were collected. The figure shows there is an exponential relationship between the time cost of event list creation and the size of the created event list. The timing information carried in the events is randomly generated. The average insertion position of a new event can be assumed as the middle point in an event list. As an event list expands, the average time

taken to insert a new event increases. Therefore, an exponential relation exists between the total creation time and the size of the final event list.

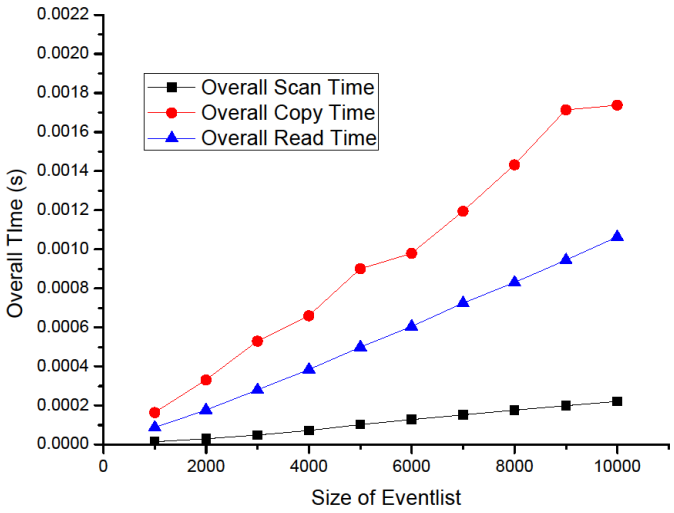



Figure 132 Event list performance on copying, scanning and reading

Figure 132 shows that the scanning, copying and reading operation has a linear relationship with the size of an event list. Scanning iterates and reads each event within an event list. Reading pops all events out of the event list. Scanning does not reduce the overall event list size, whereas reading will create an empty event list at the end of the activity. The copying operation copies all the entries in the old eventlist to an empty one. Although all these operations have linear relationships with the size of eventlist, the cost of these operations is different, where copying is the most expensive operations follow by reading and scanning operations.

# Appendix B Circuit Generation

## *B.1 MOODS*

MOODS (Multiple Objective Optimization in control and Data-path Synthesis) [138], [139] is a behavioural synthesis tool. It takes behavioural VHDL as its input and constructs structural VHDL to implement the defined behaviour. Firstly, it translates behavioural VHDL into an intermediate code (ICODE). At the behavioural VHDL level, a single statement can represent a complex equation. The idea of ICODE is to break up the complex equations into a set of operations for simplicity. ICODE can describe the behavioural VHDL function in full. Secondly, ICODE is mapped onto two abstract graphs, the control and data path graphs. The control graph defines the state of the circuit and hence controls the operations in data path. The data path modifies the data according to the control signals from the control path. The output of MOODS is a circuit with a control and data path graph structure (CDPG). The CDPG is a standard representation for sequential parts of the behavioural hardware description. The CDPG representation of a simple example is demonstrated in Figure 133.

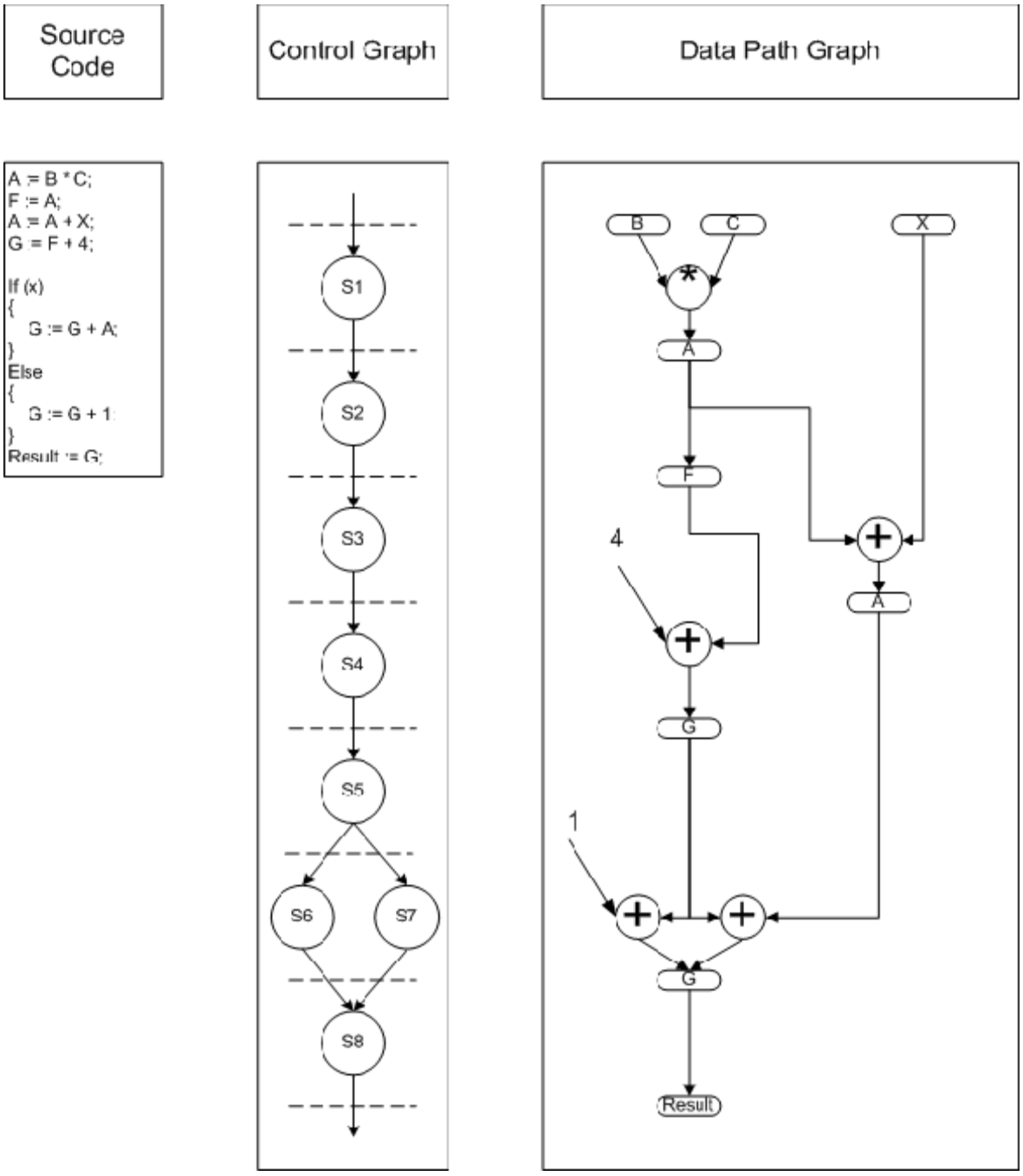



Figure 133 Control and data path graph representation of a simple behavioural description

## *B.2 Function Generator*

The function generator is part of the behavioural VHDL processing chain. The function generator creates components according to the input specification (Figure 134). In the output of MOODS, all the components are instantiations of its library components, but with variations on their input and output specifications. For example, if the library has

an AND gate component, and the circuit needs a 16-bit AND gate, the function generator will modify the AND gate template to provide a 16-bit AND gate for the circuit graph.

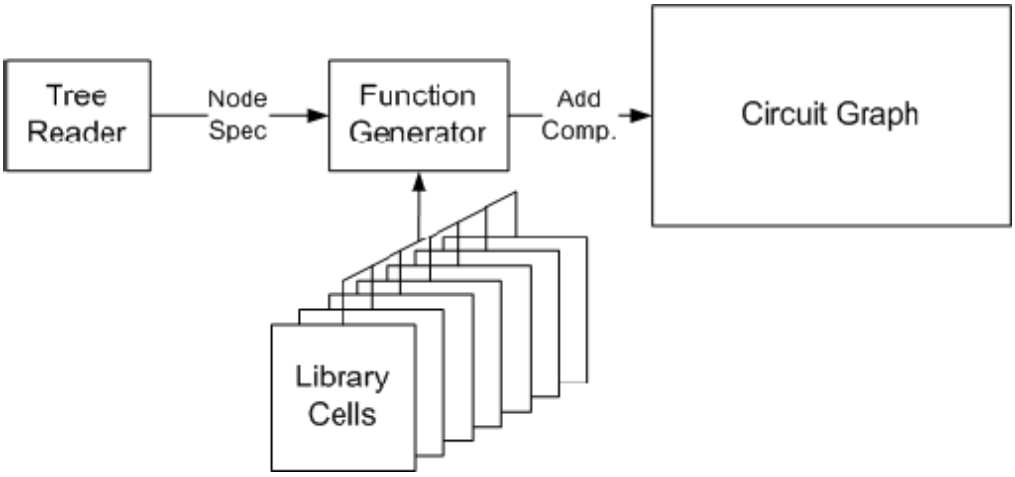



Figure 134 Function generator

There are 54 component templates in the MOODS library. Table 9 shows all the cells in the MOODS library. There are other custom components inside the function generator. Most of the components are pure combinational logic. Two components which are notable: the control general cell is used to implement the MOODS-generated control graph (The circuit is shown in Figure 135.), and the control call cell, which controls the procedural hierarchy (Figure 136). Its function is similar to a control call controller in software, but it implements the function in hardware. In this project, all library cells are assumed to have a unit propagation delay. In general, the library only specifies how the components should behave and how many IOs they possess. It does not specify the width of its inputs and outputs.

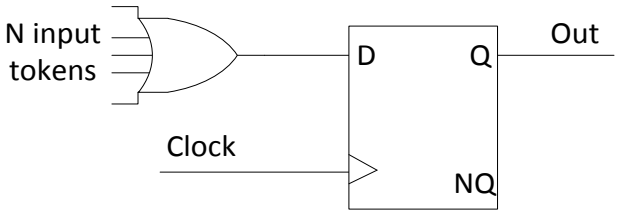



Figure 135 General control cell

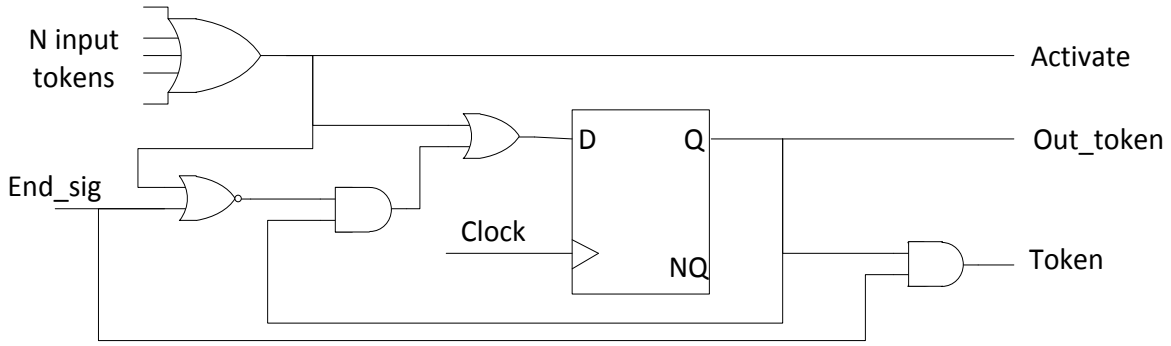



Figure 136 Control call cell

Some of the components in the library are complex circuits by themselves. For example, the multipliers are the largest components. These kinds of component could be decomposed into many basic gates, which will be converted into combinational logic in the circuit converter.

The output of function generator is a directed graph described using MOODS library components. This intermediate directed graph is designed to be used as a direct representation of the original MOODS synthesized circuit. This graph will be decomposed into basic logic gates using circuit converter.

| Function ID | Functionality |
|---|---|
| 0 | Wire |
| 1 | Global input |
| 2 | Global output |
| 3 | NOT gate (NOT) |
| 4 | AND gate (AND) |
| 5 | OR gate (OR) |
| 6 | NAND gate (NAND) |
| 7 | NOR gate (NOR) |
| 8 | XOR gate (XOR) |
| 9 | XNOR gate (XNOR) |
| 10 | Equal Test (EQ) |
| 11 | Not Equal Test (NEQ) |
| 12 | Unsigned Less Than (ULT) |
| 13 | Signed Less Than (SLT) |
| 14 | Unsigned Less Than or Equal (ULTE) |
| 15 | Signed Less Than or Equal (SLTE) |
| 16 | Unsigned Greater Than (UGT) |
| 17 | Signed Greater Than (SGT) |
| 18 | Unsigned Greater Than or Equal (UGTE) |
| 19 | Signed Greater Than or Equal (SGTE) |
| 20 | Unsigned Greater/Less Than or Equal comparator (GR ULTE) |
| 21 | Signed Greater/Less Than or Equal comparator (GR SLTE) |
| 22 | Logical Left Shift (SLL) |
| 23 | Logical Right Shift (SRL) |
| 24 | Arithmetic Left Shift (SLA) |
| 25 | Arithmetic Right Shift (SRA) |
| 26 | Rotation Left Shift (ROL) |
| 27 | Rotation Right Shift (ROR) |
| 28 | Shift left/right ALU (SHIFT) |
| 29 | Rotate left/right shift ALU (ROTATE) |
| 30 | Signed Minus (SMINUS) |
| 31 | Unsigned Adder (UADD) |
| 32 | Signed Adder (SADD) |
| 33 | Unsigned Subtracter (USUB) |
| 34 | Signed Subtracter (SSUB) |
| 35 | Unsigned Ripple Carry Adder/Subtracter (RCAS) |
| 36 | Signed Ripple Carry Adder/Subtracter (RCAS SGN) |
| 37 | Unsigned Increment counter (UINC) |
| 38 | Signed Increment counter (SINC) |
| 39 | Unsigned Decrement counter (UDEC) |
| 40 | Signed Decrement counter (SDEC) |
| 41 | Unsigned Multiplier (UMULT) |
| 42 | Signed Multiplier (SMULT) |
| 43 | Absolute value operator (SABS) |
| 44 | 1-bit D-FlipFlop (REG_BIT) |
| 45 | Register with individual load enable bit (REG) |
| 46 | Up counter with individual load enable bit (COUNTUP) |
| 47 | Down counter with individual load enable bit (COUNTDN) |
| 48 | Multiplexer without decoder (MUX 1) |
| 49 | Tristate Multiplexer without decoder (MUX 2) |
| 50 | Multiplexer with decoder (NMUX) |
| 51 | Decoder (DECODE) |
| 52 | General Control, output high when any of input lines high (CTRL GENERAL) |
| 53 | Control Call, hand shaking with other modules (CTRL CALL) |
| 54 | Concatenation (CONCAT) |
| 55 | Unsigned Extend (UEXT) |
| 56 | Signed Extend (SEXT) |
| 57 | Buffer |
| 58 | Internal input (Hierarchy IO) |
| 59 | Internal output (Hierarchy IO) |
| 60 | Data bus pooling, fill data bus with a single bit value |
| 61 | ROM decoder |

Table 9 Library components (components with brackets are MOODS library component)

## *B.3 Circuit Converter*

The circuit converter is an essential part of SDES. A directed graph using a MOODS library component can be converted into simple logic gates, which can be simulated by SDES. There are eight types of components in the library, logic gates, comparators, shift operators, algorithmic operators, counters, multiplexers, control units and bus extension operators. In this section, the implementation of these different types of operators is explained.

The first type of operator is basic logic, which can be simply implemented using the corresponding gate in the final circuit directed graph. In the original MOODS library, the width of the gate can be defined in VHDL. Hence, the circuit converter uses this as the width definition of the gate. The graph representation of a logic gate is shown in Figure 137.

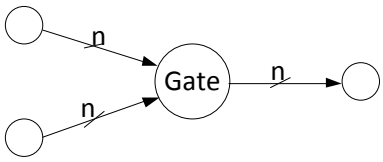



Figure 137 Logic gate implementation

The second type of operator is the comparator. This can be broken down into smaller categories, equal to, less/greater than, less/greater and equal, complex comparator block, and their signed counterpart versions.

The equal its test circuit is implemented using two gates and a bus splitter. The two inputs are first fed into an XOR gate for bit comparison, the output is then fed into a bus splitter which generates *n* 1-bit signals for an *n-bit* signal bus. These 1-bit signals are fed into either an OR gate or a NOR gate dependant on whether the equal test circuit is an equal or unequal test.

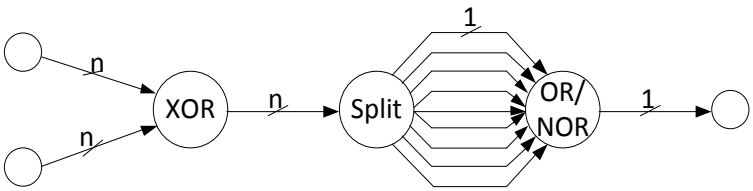



Figure 138 Equal test circuit implementation

The unsigned comparator can be implemented using an array of logic gates. To demonstrate the pattern of gates requires implementing a less than comparator; two 4-bit signals A and B are used.

During comparison, more significant bits have a higher priority than less significant bits. If the MSB of both signals are equal, the comparator carries on comparing the next significant bit and so on and so forth. According to this rule, the comparison result can be represented by the following functions for an unsigned less than comparator.

$MSB\ comparison\ result: R3 = A3.\overline{(A3.B3)}$

$Next\ MSB\ comparison\ result: R2 = A2.\overline{(A2.B2)}.\overline{(A3 \oplus B3)}$

$Next\ LSB\ comparison\ result: R1 = A1.\overline{(A1.B1)}.\overline{(A2 \oplus B2)}.\overline{(A3 \oplus B3)}$

$LSB\ comparison\ result: R0 = \ A0.\overline{(A0.B0)}.\overline{(A1 \oplus B1)}.\overline{(A2 \oplus B2)}.\overline{(A3 \oplus B3)}$

$Less\ than\ comparison\ result: R = \overline{R0.R1.R2.R3}$

According to the pattern of gates in the formula above, three layers of gates will be able to implement this comparator function. The first and second layers are NAND gates and XNOR gates that sit between each bit of the two inputs. The final layer are the AND gates that combine the outputs of each bit and produce a result for each bit. A final NAND gate collects the result from each bit and produces the final comparison output. The layout of this gate is shown in Figure 139.

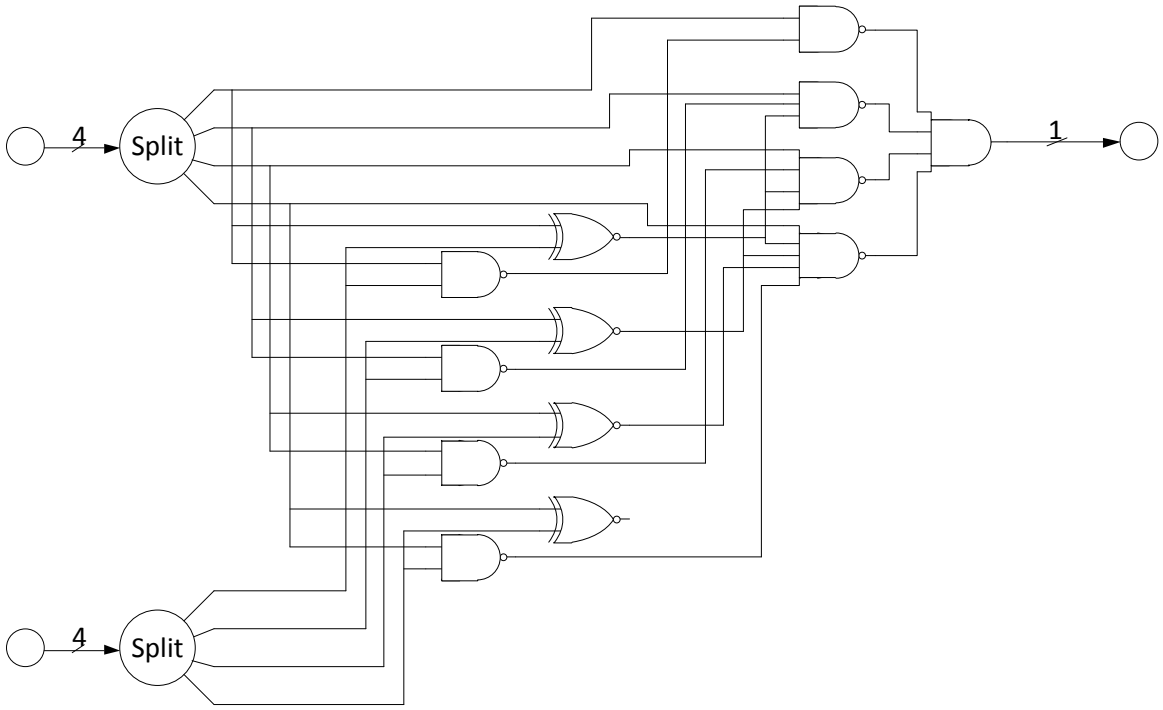



Figure 139 Unsigned Less Than (ULT) implementation

For the signed operations, negative numbers are represented in 2's complement form. The unsigned comparison circuit can be reused by splitting the signed bit from the comparison. For example, if the two inputs are 0001 (1 in decimal) and 1011 (-5 in decimal) the result of comparison of the lower 3 bits shows that 1 is greater than -5. However, the signed bit comparison will show the opposite. This leads to the truth table shown in Table 10, which shows how input signed bits and lower bits comparison result affects the final signed comparison result.

| Lower bits comparison result (C) | IP A signed bit (A) | IP B signed bit (B) | Final Comparison Result (R) |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |

Table 10 Signed comparison truth table

The truth table can be represented and simplified as

$$R = \bar{C}.A.\bar{B} + C.\bar{A}.\bar{B} + C.A.\bar{B} + C.A.B = C.A + \bar{B}.(C \oplus A)$$

Implementing this function and adding it on top of the unsigned comparison circuit block, the final comparison output would be the signed comparison result. The same technique is applied when implementing the GR_ULTE component, which provides a selection signal to choose which function it was performing. The truth table for selection signals and the comparison output is listed in Table 11.

| Comparison result (C) | Selection bit 1 (S1) | Selection bit 0 (S0) | Final Comparison result (C) |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |

Table 11 GR_ULTE selection signal truth table

The truth table can be represented and simplified as

$$R = \bar{A}.\overline{S1}.S0 + A.S1.\overline{S0} = \overline{(A + S1)}.S0 + A.S1.\overline{S0}$$

The third type is shift operators. They are implemented using the same principle, which uses a cascade of 1-bit shift operators and a multiplexer to produce the output. The circuit structure is shown in Figure 140.

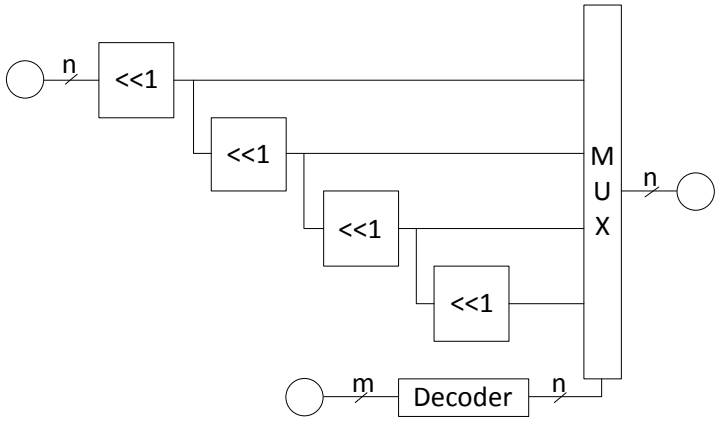



Figure 140 Shift operation circuit structure

The fourth type is algorithmic operators. Adders, subtractors and multipliers are in this category. The first two operations can be implemented using a chain of full adders, bus extension and a set of 2's complement circuits. As an example, an unsigned subtractor is shown in Figure 141.

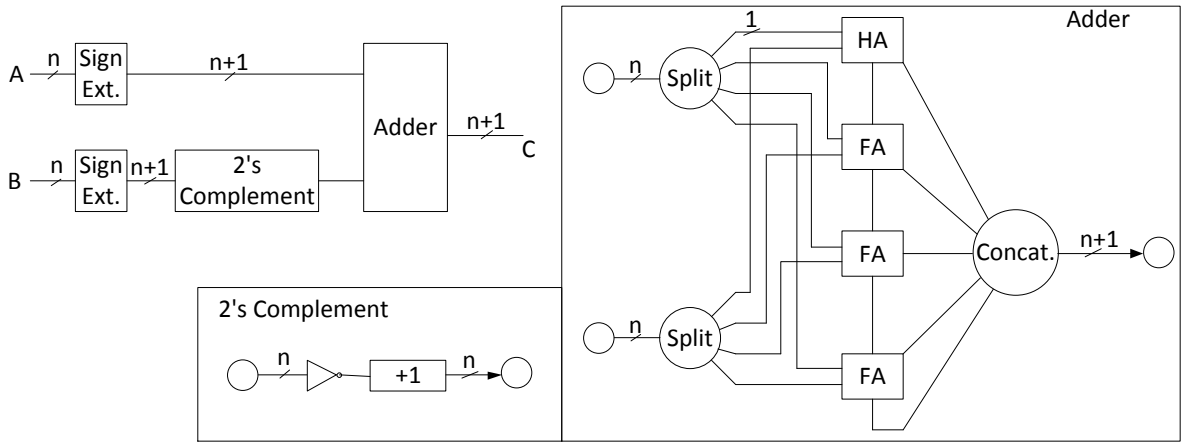



Figure 141 Signed subtractor circuit structure

Other adder and subtractor circuits can be derived from the structure shown above by removing part of the circuit. When implementing the Ripple Carry Adder and Subtractor (RCAS) circuit, an additional multiplexer is added after the 2's complement block. This multiplexer lets either the original signal or its 2's complement form through to the adder, i.e. controlling whether the circuit as a whole is performing addition or subtraction.

The multiplier circuit is implemented using an array of full adders. The size of two inputs must be identical, but the maximum size is only limited by the maximum bus size in the simulation system, which is 256-bits in case of SDES. The circuit structure is shown in Figure 142.

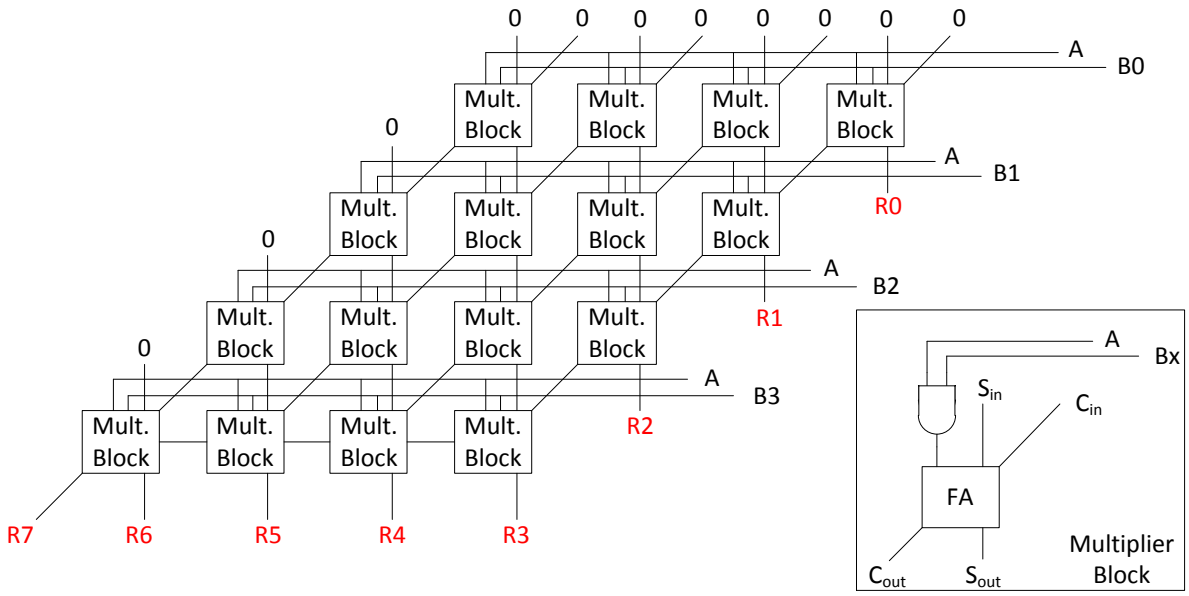



Figure 142 Multiplier circuit structure

The signed version of the multiplier is constructed by adding three sets of 2's complement circuits and a multiplexer, as the magnitude of the multiply product does not change with signs. The multiplier is computed by converting operands into positive numbers at the input and later converting to the correct form at the output stage. The structure of a signed multiplier is shown in Figure 143.

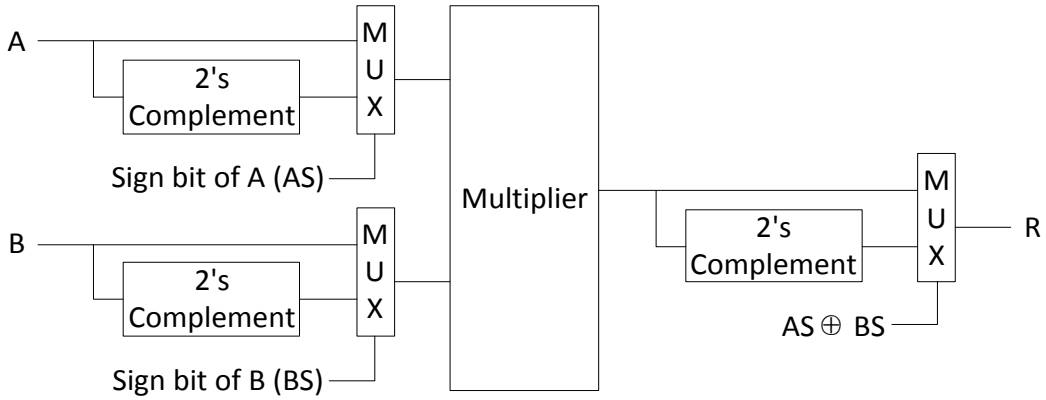



Figure 143 Signed multiplier circuit structure

The fifth category is counters. This is the first type of component that includes a storage element in the circuit. As a result, prior to the introduction of different types of counters, the structure of the basic flip-flop is explained first. A flip-flop in a MOODS library requires a load enable and low reset function in addition to the basic storage function. However not all the flip-flops require a load enable signal in the system, such as control units, as a result two versions of flip-flop exist in this system. The structure of a flip-flop in function generator is shown in Figure 144. The optional part of the flip-flop can be removed to form a flip-flop without a load enable function. The multiple bit register function is implemented by duplicating the flip-flop circuit until it reaches the number of bits specified in the MOODS component directed graph.

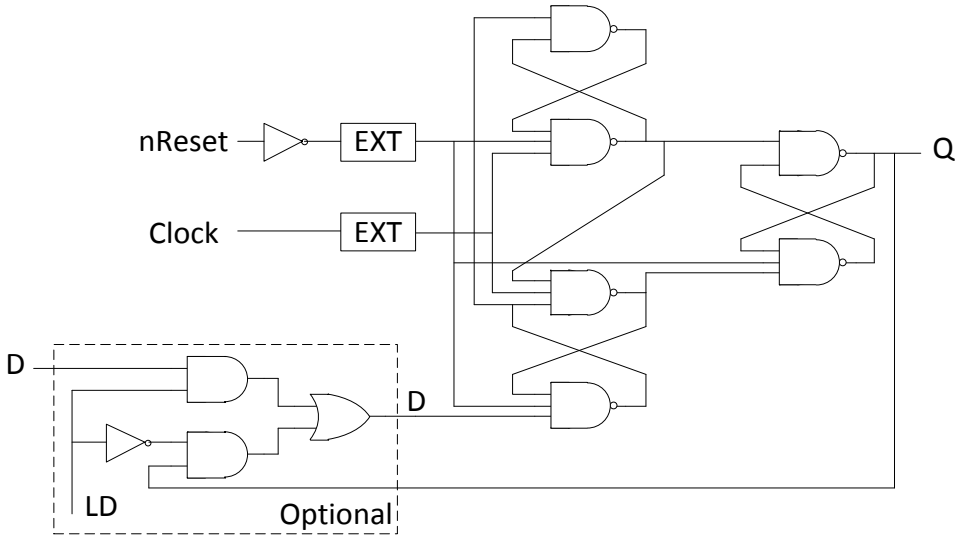



Figure 144 Flip-flop circuit structure

After acquiring a multiple bit register, a counter circuit can be implemented. A counter in the MOODS library requires two control signals, load and count. When the load signal is high, the counter is enabled to load new values. Whether it loads a new value from the input or counting computation is controlled by count control signal. The implementation of a count up circuit is shown in Figure 145.

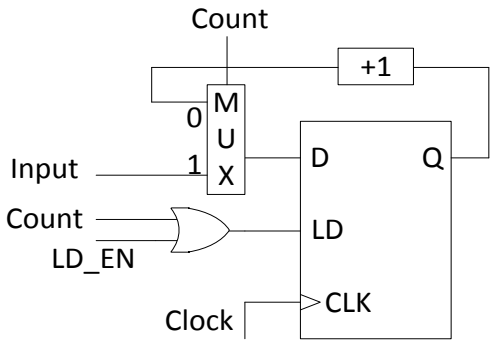



Figure 145 Count up circuit structure

The sixth type is the multiplexer. This is used multiple times in the previous types of component. It is an essential component in this circuit converter. There are two types of multiplexers in MOODS library, one without a decoder and the other with a decoder. The multiplexer without a decoder is implemented using a chain of AND gates and an OR gate at the output. The decoder is implemented as a connection map of each bit of the selection signal. The structure of a multiplexer with decoder circuit is shown in Figure 146.

Last two categories are control unit and bus extension circuits. The control unit is shown in Figure 135 and Figure 136, and the bus extension is a trivial circuit which is straight forward and therefore omitted from explaining.

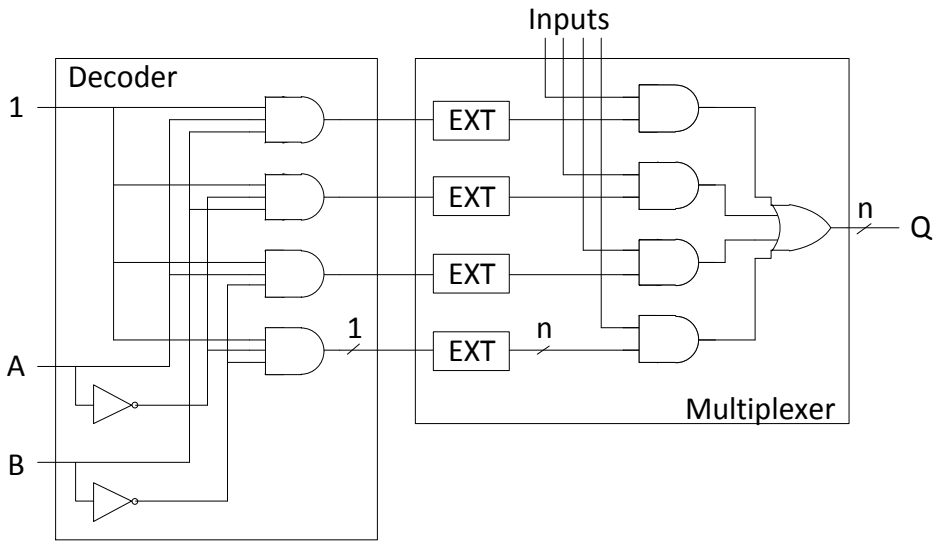



Figure 146 Multiplexer with decoder circuit structure

## *B.4 Bench File Parser*

The Bench format is one of the input circuit formats. A Bench circuit reader to import circuits has been written. There is no formal document written to define the Bench syntax, but the structure is simple and it is widely used to describe the benchmark circuits [140] in the ISCAS85 benchmark circuit, there are only 7 kinds of basic logic gates: AND, NOR, BUFF, OR, NOT, XOR, and NAND.

```
System IOs:
INPUT/OUTPUT (IO wire name)

Examples:
INPUT(A),OUTPUT(C)

Gates:
target wire name = gate function (inputs...)

Examples:
C = AND(A,B)
C = XOR(A,B)
```

However there are shortcomings when it comes to signal buses, which Bench format does not support. It is necessary to split a bus into many individual single bit signals. The language has been extended to support hierarchy in this project, making the format more user-friendly and usable. The format is:

```
CELL core name (IO names,...)
cell circuit description
...
END CELL
```

This sub architecture can only define a single level of hierarchy. It cannot accept nested CELL definitions. However, within this single level, there can be calls to other cells.

```
CELL A(C,D,E)
...
END CELL
CELL B(F,G,H)
A(F,G,H) // call to cell A
...
END CELL
```

If the circuit is defined as above, the parser will accept this architectural call. However, if they are hierarchically defined as below, it will not be able to parse the file.

```
CELL A(C,D,E)
CELL B(F,G,H)
...
END CELL
END CELL
```

In the basic parsing model, when the parser has finished reading a line, a corresponding component will be generated and stored in the circuit graph. When the parser finishes reading the Bench file, a complete circuit graph is generated. In hierarchy mode, a sub-circuit is generated and stored as a complete circuit but instead of adding it to the final circuit graph, it is stored in a cell library. Whenever a call to a predefined cell occurs, the corresponding circuit graph in the cell library is copied and added in either another cell circuit or the main circuit graph.

Unlike the standard syntax, a hierarchical call to cells is allowed to have multiple inputs and multiple outputs. The order of IOs in the calling function must match the order of IOs inside the cell definition line. Although the order of IOs in the definition line is fixed, the order of IO definitions inside the cell does not matter. For instance, the ports of a cell can be defined as C, D, E, but the order of the definitions of them can be random as shown below.

```
CELL A (C,D,E)
OUTPUT(E)
INPUT(C)
OUTPUT(D)
...
END CELL
```

## *B.5 Event Loader*

The previous sections introduced the processing of input circuits. This part focuses on the input events for the circuits. To simulate a circuit, a set of input events and possibly clock events are required. The circuit only provides the hardware information for the simulation, whereas the input events are the suppliers of circuit state change triggers.

An event in PDES consists of a component identifier, a time stamp and a new state value. The component identifier in this project is an integer number. As the simulation takes place across multiple physical machines, the memory address is no longer a reliable reference. The time stamp is an integer indicating the virtual simulation time of this state change, which is not the event execution time in real time. The new state value is a string of characters, as the state of circuit nodes can have various values, high(1), low(0), undefined(U) and unknown(X). The high and low values represent the logic level of wires. The undefined state is the initial state for all wires within the circuit. The unknown state is where a wire is driven by two gates with different driving values, or unknown input causes a gate to output this unknown value. An event can represent a change of data bus, which has multiple bits. Therefore, the state value is represented as a string of characters. The length of this value defaults to 256.

The input events are stored in a text file for the event loader to parse. The format of the file is shown below:

```
single events:

<input source name>,<time>,<value>

clock events:

<input source name>,<startup time>,<value>,<half cycle time period>
```

The source names are the input source names, the time unit is nanosecond, and the values are in binary format and the time period indicates how long it takes to switch the value to its complement. For example, a clock with a full cycle period of 20 ns would be entered as "clock, 0, 0, 10". An event with value '1' on wire "data" at time 35 ns is

represented as "data, 35, 1". In the end, the event loader translates these formatted strings into events. Although the source names are in text format, they are transformed to corresponding integers according to a name-ID table provided by the circuit parser. The events can then be passed on to the SDES for processing.

# Appendix C Development Platforms

**PC**

The SDES is developed on PC platform which provides an easy to use debugging platform in comparison to the console based parallel environment. The PC platform employed has a 2.0GHz dual-core processor and 4Gbits of memory. Although there are only two physical processors present during a simulation, the MPI platform is able to initiate and map multiple virtual processes on a single processor. This provides easy access to parallel debugging, but the performance based on virtual processors has little relevance to the final parallel program. As a result, after the initial development, porting to a truly parallel platform is required.

**Cluster**

In order to estimate the performance gain that can be obtained by porting the SDES to the SpiNNaker platform, a parallel computing platform should be used as an intermediate development platform. As well as forming an emulation of the SpiNNaker system on a parallel computing platform, this can also be used for future development purposes.

The cluster employed in this project is called Iridis, it ranks 74$^{th}$ in the top 500 supercomputer in the world. The cluster features 1008 Intel Nehalem compute nodes; each node has two 4-core processors. All nodes are connected to an InfiniBand network which provides high throughput, low latency and scalable communication links. The communication speed performance is evaluated in section 5.2.3 .

**SpiNNaker**

The SDES is designed to target the SpiNNaker platform. The system structure of the SDES is retro-fitted to the SpiNNaker hardware structure. However, due to unforeseeable delays in the hardware development of the SpiNNaker system, the final hardware was not available in time. The following is a detailed explanation of the internal communication mechanisms utilised by SpiNNaker.

## *C.1 Communication Packet Types*

The communication system in the SpiNNaker system is cross-node. The SpiNNaker node does not contains the ability to process complicated communication protocols. However, the communication network does support the passing of fixed-sized communication packets. These are formatted in four different types, multicast, P2P (Point-to-Point), nearest neighbour, and fixed route communication packets.

**Multicast Packet Format**

The multicast is a natural communication choice for neural networks, as the connectivity is stored in the routing table within the SpiNNaker node. The multicast (MC) packets are used to send a packet from one processor to multiple target processors. These can reach multiple targets. The multicast packet consists of a 32-bit routing key, and an 8-bit control signal with an optional 32-bit payload.

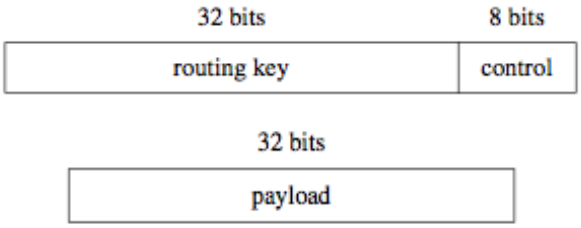



Figure 147 Multicast message packet formats

The routing key provided by the source processor controls the message targets. The router inside the SpiNNaker node directs the message to one or more ports according to the routing table. (See section 4.5.3 for more details)

**Point-to-Point Packet Format**

The structure of the point-to-point (P2P) packet is shown in Figure 148. The 16-bit source ID is the ID of the processor, which sent this packet. The 16-bit destination ID indicates the target processor ID. The destination ID is used to decode the output path(s) for the packet in the router. There are 8 possible routing for a packet, to the local monitor processor, to one of the six output links to the adjacent nodes or broadcast the packet to all six output links. There is a 64K entry 3-bit SRAM lookup table to decode

the 16-bit destination ID. Each 3-bit entry decides where the packet will be routed to the eight possible routings.

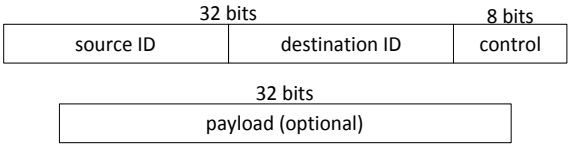



Figure 148 Point-to-Point message packet formats

**Nearest-Neighbour Packet Format**

The format of the nearest-neighbour (NN) packet is shown in Figure 149. The control bits indicate the difference between NN packets, MC packets, P2P packets, and FR packets. The NN packet has two types, one implements the write function and the other one implements the read function. The write packet may have the additional payload but the read packet will not. In **both** cases, the target neighbour will return a reply packet, the write packet will receive an empty payload reply packet from its neighbouring processor, whereas read will receive the requested data.

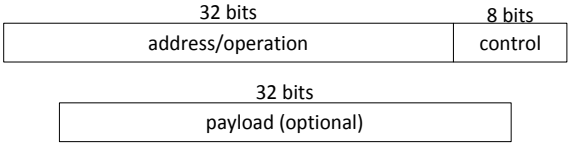



Figure 149 Nearest Neighbour message packet formats

**Fixed-Route Packet Format**

Fixed route packet is used for monitoring and debugging mechanism of the SpiNNaker platform. All packets are routed to one node which connects to the external world. As a result, the routing information is replaced with a payload, and the optional payload in other packet types becomes an additional payload in FR packet. And the format is shown in Figure 150.

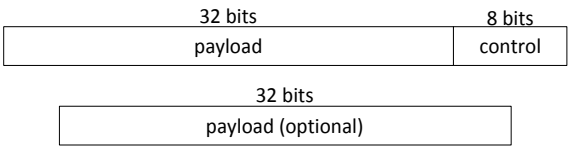



Figure 150 Fixed Route message packet formats

Throughout this project, the only communication packet that employed is P2P packets. Without the knowledge of simulated problem, the initialization stage of SpiNNaker system can only prepare the P2P routing table, not MC table. The P2P routing table is

hardware dependent rather than problem dependent. Due to the difficulty discussed in section 4.2.3 , the P2P packet is the only type of packets used in this project.

## *C.2 Communication Protocol Details*

Based on SpiNNaker communication packets, the messages that pass through the SpiNNaker system must be fitted into the 32-bit payloads carried by communication packets. As a result, a uniform communication protocol is required in order to encode and decode the messages in these 32-bit payloads.

As shown in Figure 151, the 32-bit payload is split into three sections. The first two enable the system to transfer a maximum number of 256 different 4Kbit messages between any two nodes at any time. As many different types of message exist within SDES system, the data itself needs to have a further protocol to identify the message type for message reconstructing and event execution purposes. This further identification requirement led to a further encoding of message types:

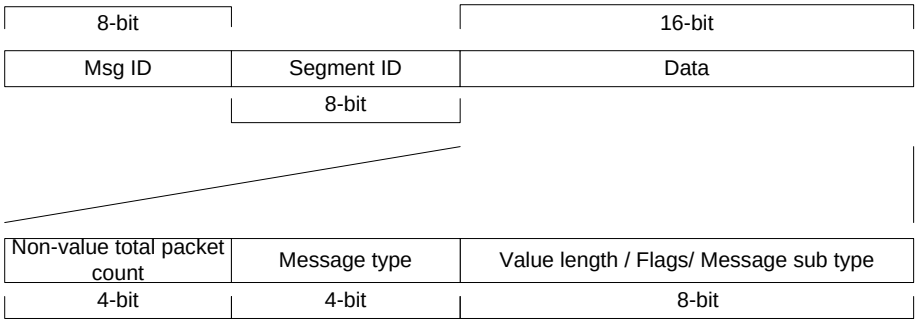



Figure 151 Header format

A detailed list of communication specification in slave processes is shown in Table 12. The 4-bit message type allows 16 different message codes. In DLB control signals a further subtype is employed. In the initialization stage, events, wires, components, array information and simulation target time are distributed through message types 0, 1, 3, 4, 5 and 6. During a simulation, events, null message and GVT are reconstructed through type 0, 7 and A. The DLB uses type 0, 2, 6, 7, B, C and D to implement the event and component transfers and the necessary control over specific LPs, such as source and target LP.

| Message code | Sub type | Message Type |
|---|---|---|
| 0 | Value Size | Event |
| 1 | n/a | Wire initialization |
| 2 | n/a | Event request |
| 3 | n/a | Component initialization |
| 4 | n/a | Routing table info. (Reserved) |
| 5 | n/a | Array info. (Reserved) |
| 6 | n/a | Reset weight count |
| 7 | n/a | Null message |
| 8 | n/a | Simulation stop signal |
| 9 | n/a | Simulation start signal |
| A | n/a | GVT sync. |
| B | n/a | Component transfer |
| C | Value Size | Component value transfer |
| D | | DLB control signals |
| | D0 | Target LP received completely |
| | D1 | Defrost LP |
| | D2 | Freezed LP as target LP |
| | D3 | Freezed LP as source LP |
| | D4 | Reset boundary |
| | D5 | Forced to send null |
| E | | (Empty) |
| F | | (Empty) |

Table 12 Slave process communication specification

Monitor processes have a more complicated communication specification than the slave processes. Slave processes only need deal with the local monitor process throughout the simulation process, whereas monitor process need to deal with all the local slave processes as well as the other monitor processes in the array of SpiNNaker nodes. The specification of the communication code is listed in Table 13.

| Message code | Sub type | Message Type |
|---|---|---|
| 0 | Value Size | Event |
| 1 | n/a | Wire initialization |
| 2 | | Load balance signals |
| | 20 | Transfer of events |
| | 21 | Transfer of states |
| | 22 | Comp. value request |
| | 23 | No comp. value found |
| | 24 | Return of null msg. requested |
| | 25 | Request neighbour LB status |
| | 26 | Return of neighbour LB status |
| | 27 | Request forced null message |
| | 28 | Return of comp. value request |
| | 29 | Lock target monitor |
| | 2A | Lock source monitor |
| | 2B | Unlocked |
| 3 | n/a | Component initialization |
| 4 | n/a | Routing table info. (Reserved) |
| 5 | n/a | Array info. (Reserved) |
| 6 | n/a | Simulation results (Reserved) |
| 7 | n/a | Null message |
| 8 | n/a | Simulation stop signal |
| 9 | n/a | Simulation start signal |
| A | n/a | GVT sync. |
| B | n/a | Component transfer |
| | B0 | Trans. between slave processes |
| | B1 | Trans. between chips |
| C | Value Size | Component value transfer |
| D | | DLB control signals |
| | D00 | Source LP finished sending comp. and events completely |
| | D01 | Source LP finished sending comp. |
| | D1 | Target LP received completely |
| | D4 | Defrost LPs |
| E | | (Empty) |
| F | | (Empty) |

Table 13 Monitor process communication specification

Most of the specifications for the monitor process are the same to the slave processes. The two different message types are 2 and D, both of them related to DLB. Most of the type 2 messages are dealing with detailed DLB execution, whereas type D messages are used as overall DLB control. A DLB process starts from the monitor process requesting a load balancing status (LB status) from a target neighbour monitor using message type 25 and 26. If the target monitor is free, the source monitor process will check if preceding monitor processes can be locked from DLB with other processes using message type 29, 2A and 2B. When the transfer of components starts, events and component values are sent using message types 20 and 21. If some of the input components are not local to current LP, a request for component values will be sent over the network and returned to the source monitor process using message types 22, 23, 24 and 28. After these system statuses are updated, a new boundary is established and

missing timing information requested directly using message types 24 and 27. The detailed DLB flow is explained in appendix D.5.

# Appendix D Implementation Action Minutiae

In this project a chain of software links the core simulation system to the outside world. From the input side there is the VHDL parser, function generator; directed graph converter and the bench file parser. These tools form a seamless bridge between the VHDL and Bench circuit description files and the SDES system. They are integral parts of the implementation of the circuit simulator.

The overall project is about discrete event digital circuit simulation. As a result, there are some fundamental components that are essential for all the software involved in the project to compute and to communicate. Components such as events, event lists, and directed circuit graphs form the basis of circuit simulation.

## *D.1 Events*

An event updates the system status in a simulation. In a circuit simulation, it represents a change of the logic state in the circuit. As the simulation in this project is a discrete event system, the event will also be a discrete event, which simplifies the contents of an event drastically in comparison to a continuous system. In a discrete logic event, three basic atoms of information need to be specified, a discrete time, an identifier and a new system state. For the implementation of this project, both time and identifier are integers and the system state is defined as an array of characters, where the maximum length is defined as 256 and can represent 256-bit logic state. The header file is shown below.

```
class logic_event {
public:

    bool operator== (const logic_event& e);
    bool operator!= (const logic_event& e);
    bool operator< (const logic_event& e);
    bool operator> (const logic_event& e);
    int time;
    int id;
    char value[128];
};
```

The operator overloading functions enable the eventlist (which will be described in the next section) to sort events within a data structure container according to their timestamps. The overloaded equal/unequal comparison operators only compare the timestamp and the identifier of an event.

## *D.2 The Eventlist*

As the number of events grows within a system, a means is required to store and manage them. Unlike other containers, the eventlist will be a sorted container by default, because the events are executed in sequential time order and it will speed up the access time if the container is sorted before execution. However, there is no standard container that fits this profile; as a result, a new container template is created in order to fulfil the need of eventlists, in this case, the template is called *sort_q*.

For *sort_q*, requirements from different techniques will define the functions included within the template. The first basic need of using an eventlist is the ability to add random events without worrying about the order of them within the structure. The cost associated with each type of sequence containers is listed in Table 14. [141]

| | vector<T> | deque<T> | list<T> |
|---|---|---|---|
| Insert/erase at start | linear | constant | constant |
| Insert/erase at end | constant | constant | constant |
| Insert/erase at middle | linear | linear | constant |

Table 14 Cost of container operations

The best suited container for this requirement is the *list* container. As the insertion time at a random position within a list is a constant value, it will create a huge benefit compared to the other two candidate structures.

Based on list structure, additional functions were added to the new structure template. Functions such as adding and removing events, existence of an event and garbage collection, were added to the new container template to meet the need of eventlist during simulation.

```
template< typename Object >
class sort_q : public std::list<Object> {
public:
       typedef typename std::list<Object>::iterator iterator;
       typedef typename std::list<Object>::const_iterator const_iterator;
       typedef typename std::list<Object>::reverse_iterator r_iterator;
       typedef typename std::list<Object>::const_reverse_iterator const_r_iterator;
       iterator insert_n (iterator itr, const Object & x);
       iterator insert_n (r_iterator itr, const Object & x);
       iterator insert (Object & x);
       iterator insert_n (const Object & x);
       int extract_id(int id,sort_q<Object> * op_q);
       bool exist(Object & x);
       void garbage_collection(Object & x);
       bool annihilate(Object & x);
};
```

The adding and removing event will automatically insert an event to the list while maintaining the time order within the list. The *sort_q* template will sort the eventlist in *ascending* order, user access both ends of the list depends on the individual application requirement. For example, the main eventlist in a LP will process the beginning of the eventlist to gain access to the event which holds the lowest timestamp in the list. In comparison, the event log of a system will simply push the event at the back of a *sort_q*. As the timestamp in the newly generated event will always carry a timestamp not older than the ones already exist in the log.

The existence check and garbage collection are designed to meet the criteria of time warp simulation. The existence check is used when the lazy cancellation technique is used. After each simulation rollback, the recomputed events check against the event sending log to check whether the new event should be sent over again.

The garbage collection function simplifies the garbage collection process by a simple function call. The user only requires a GVT and all the events that have timestamp smaller than GVT are be discarded automatically and the time order of the rest of events is maintained.

## *D.3 Directed Circuit Graphs*

The system states describing the logic state of each discrete digital component within the simulator are stored in a directed graph (digraph) container. The digraph structure is not a standard library container, but it is based on the standard list structure. The digraph allows the data to be stored as a network of components with explicit arc connections between them: a digital circuit can be retro-fitted to it without any modifications. There are two sets of data that are stored in a digraph, node and arc data. A digital circuit can be mapped to it by storing digital gate information on the node of digraph and the wire connection information stored on the arc of digraph. These arc connections have direction information associated with them, which can express the concept of inputs and outputs of a digital gate (Figure 152).

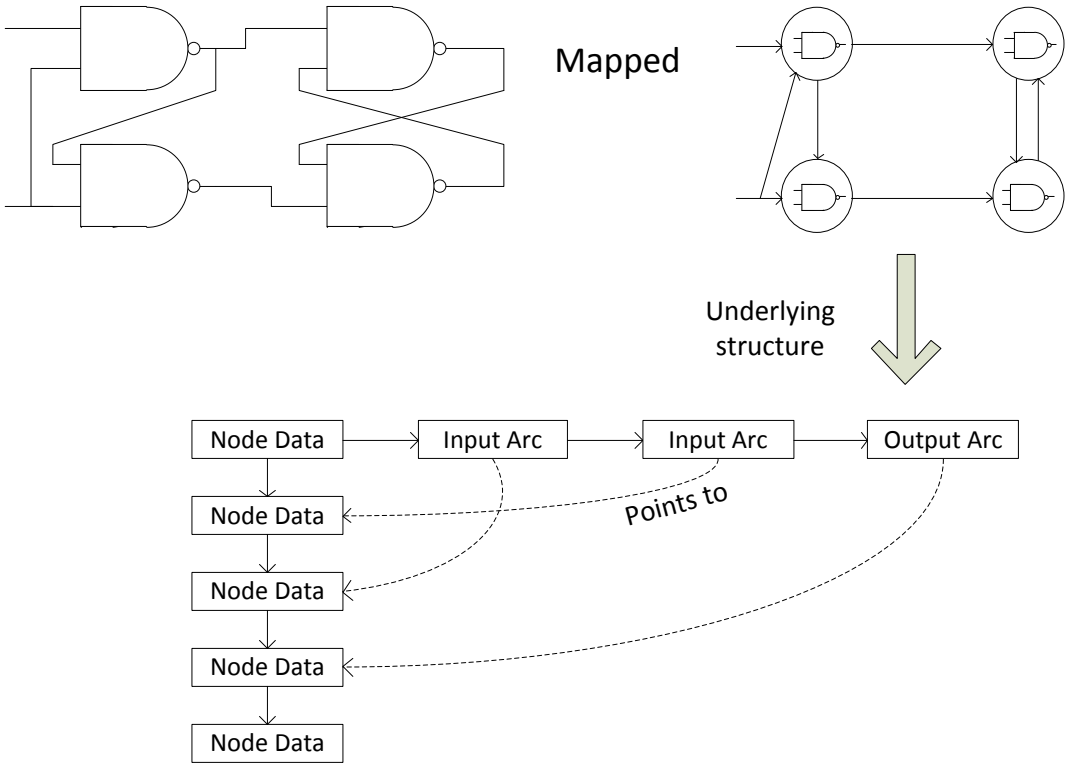



Figure 152 Directed Graph Mapping of Circuit

There are two types of circuit description within this project. There is an intermediate layer which directly maps MOODS library components into the node. The other structure stores the basic logic gates which are expanded from the MOODS library components in the nodes of a digraph. The conversion action is implemented by the

circuit generation software which is explained in appendix B.3. In this section, only the basic logic gates are explained.

```
class component {
public:
	component();
	int init(int,int,int,int,int,string,int,int);
	int size();

	int id;
	int fn;
	int delay;
	int partition;
	int flag;
	int msb,lsb;
	int weight;
	char value[128];
};
```

The gate information is stored in the node structure shown above. This includes identifier, gate type, the propagation delay, partitioning information, length of value and the actual value of the gate. The additional weight information is created purely for the purpose of dynamic partitioning, as it is an essential parameter in order for the partition algorithm to monitor the overall activities during a simulation. The types of logic gates are defined as integers and listed in Table 15.

| Type | Function |
|---|---|
| AND | 0 |
| NAND | 1 |
| OR | 2 |
| NOR | 3 |
| XOR | 4 |
| XNOR | 5 |
| BUFF | 6 |
| NOT | 7 |
| INPUT | 8 |
| OUTPUT | 9 |
| CONCAT | 10 |
| UEXT | 11 |
| SEXT | 12 |
| XORB | 13 |
| XNORB | 14 |
| ORB | 15 |
| ANDB | 16 |

Table 15 Function type definition table

The gates are categorized into 4 types, logic operators, I/Os, wiring switches and pull down operators. The first two categories are straight forward. Wiring switches emulate different wiring combinations. CONCAT joins different signals together to create a new signal bus. The UEXT and SEXT will extend the signal bus width to a new value. The fourth category is a custom gate type. The gates have the same truth table as their corresponding logic operators when the inputs are 0s and 1s. However, when inputs are outside the 0s and 1s range, the output of these gates will be 0. Pull down operators are very different from the logic operators which can output Xs and Us when the same inputs are fed into them. An example truth table of XOR and XORB was shown in Figure 153.

| INPUTS | 0 | 1 | U | X |
|---|---|---|---|---|
| 0 | 0 | 1 | U | X |
| 1 | 1 | 0 | U | X |
| U | U | U | U | U |
| X | X | X | U | X |

(a) XOR Truth Table

| INPUTS | 0 | 1 | U | X |
|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 |
| U | 0 | 0 | 0 | 0 |
| X | 0 | 0 | 0 | 0 |

(b) XORB Truth Table

Figure 153 XOR and XORB truth table difference

The pull-down gates are necessary when implementing the multiplexers for the selecting signals, which is necessary to always select the lowest order of input by default when selection signal is not ready.

```
class wire {
public:
    wire();
    int init(int iid,int idelay,int imsb,int ilsb);
    bool operator==(wire& wr)
    bool is_eq(wire& wr);
    int id;
    int delay;
    int msb;
    int lsb;
};
```

The arc structure stores wiring information and the header file is shown above. Wire class has a much simpler data set. It has a unique identifier, propagation delay and width information. The Most Significant Bit (MSB) and Least Significant Bit (LSB) of wires must lie in the range of width of the gates. However, the width of wire can have a smaller or equal to the width of a gate. For example, if the MSB and LSB of a gate are 15 and 0, the range that an output wire can have will be bounded within this range and

the width can vary from a minimum of 1 bit to a maximum of 16 bits. Further information on the digraph container can be found in [22].

The focus here is how the directed circuit graph was constructed and what functions should it feature. From the perspective of a user, there is no iterator or pointer involved in a digital circuit, as a result, the construction of a circuit should purely be based on the identifier of the gate. From the perspective of the digraph itself, the arcs cannot be stored if either end of an arc does not exist in the system, which might happen if arcs are added to a graph without a complete knowledge of nodes within a circuit. In summary, a circuit digraph will be inputted based on the identifier in a circuit and the nodes will be added to the digraph before any arcs are added.

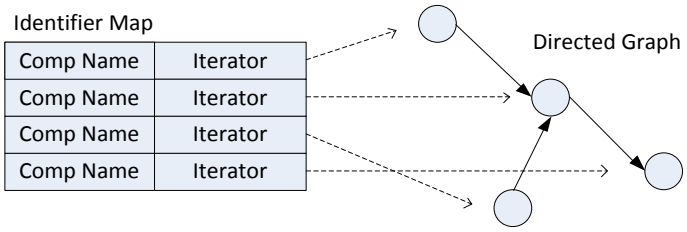



Figure 154 Circuit Digraph Data Structure

The final implementation of circuit digraph includes two containers as shown in Figure 154. One of them is identifier to iterator associated map container, the other one is the digraph container. The first container stores the associated information of the gate identifier and the iterators which points to the stored data within the digraph. The second container takes care of the heavy duty storage function. The associated map container plays an important role when the simulation executes. In a parallel simulation, there are multiple copies of the same circuit, however the memory space are different for every copy of the circuit in each logic process. Therefore, a global identifier must be established and this associate map will translate the global identifier to a local pointer which has direct access to the actual data.

```
class comp_graph {
public:
	comp_graph();
	int add_node(component nid);
	int reset_value(component nid);
	int change(component nid);
	int change_partition(component nid);
	int change_flag(component nid);
	int add_arc(int nid1, int nid2, wire wr);
	int get_size();
	int get_size(int chip,int proc);
	database::comp_digraph* get_cg();
	database::int_it_map* get_node_map();
	void update_map();
	void clear();
	comp_graph operator= (comp_graph& in);
private:
	database::comp_digraph cg;
	database::int_it_map node_map;
};
```

The circuit digraph class header is shown above. There are several functions that need further explanation. The *reset_value* function sets the initial value of all gates to 'U', which initializes the system after the construction of the circuit graph. The a*dd_arc* function takes the identifiers of a source gate and a target gate and the specification of the wire connection to establish the connection between the two gates. *Update_map* function updates the identifier to iterator map. This is required when this circuit digraph is copying the content of another circuit digraph.

## *D.4 Discrete Event Simulation*

After the basic data structures described in the previous sections, the implementation of the actual simulation part is investigated in this section. The event processing part of the system is enclosed in a single simulation engine class. The simulation engine class must be initialized with the simulated circuit which will be produced by the circuit generation circuit. During a simulation, the simulation engine class takes an eventlist and a target time as its input and processes all events carrying lower timestamps than the target time specified in input.

```
class sim_engine {
public:
	sim_engine();
	void run(int,sort_q<logic_event>);
	void load_cir(comp_graph);
	comp_graph* get_cg();
	void printvcd(string,int,map<int,string>);
private:
	void process_event(logic_event,sort_q<logic_event>*)
	bool diff_test(comp_digraph_it*,logic_event);
	string extract_input(arc_iterator,logic_event);
	bool reevaluate(comp_digraph_it,logic_event,sort_q<logic_event>*);
	string switch_endian(string);
	comp_graph cg;
	logic_event loc_e;
	int sim_time;
	sort_q<logic_event> q,tmp_q;
	int end_time;
	sort_q<logic_event> log_q;
	comm_cmd cc;
};
```

The header file for simulation engine is shown above. The relationship between the different functions is shown in the flowchart of Figure 155. The chart shows how the external interface is implemented and the relationship between the different functions.

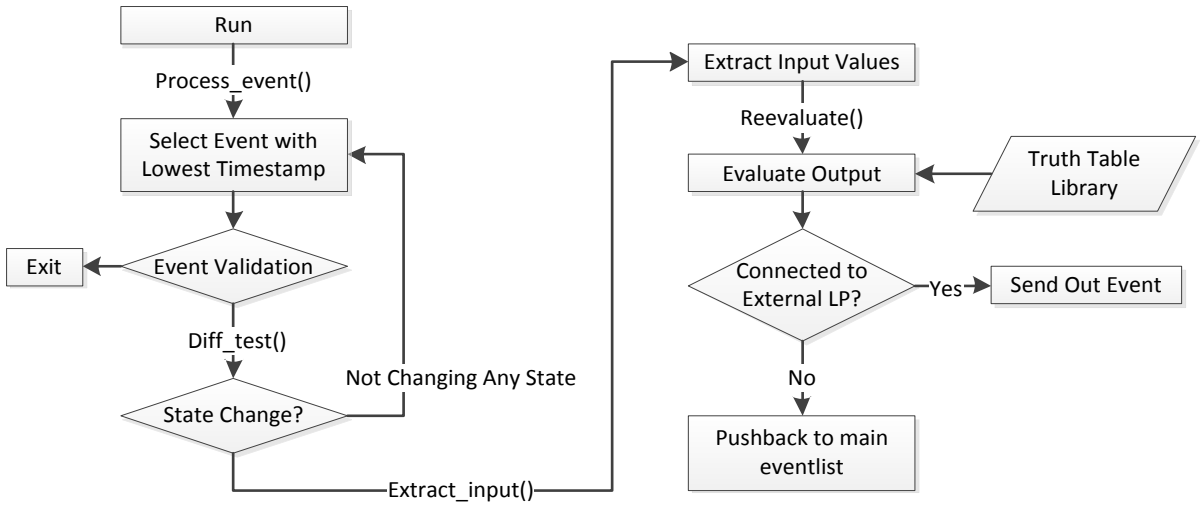



Figure 155 Simulation engine flowchart

The event processing starts off by validating the event with the lowest timestamp. If the event at the front of the list has a timestamp later than the specified simulation target time or the main event list is empty, the process will stop the event processing until the next *run()* function is called.

When the event has been validated for further processing, the system checks the current value in the circuit digraph against any new value in the event to see if the new event will change the system state or not. If the value is identical, no further processing is required as the state of the circuit will stay the same. Otherwise the simulation engine will start collecting all the iterators of gates that take the gate specified in the event as an input. The connectivity can be acquired by querying the circuit digraph and a vector of gate iterators will be returned. The states of these collected gates will be updated one by one. The input values of these gates will be collected using *extract_input().* The *revaluation()* feeds these input values into a truth table library and the correct output will be returned by the library. If any of collected gates belong to another LP, the simulation engine will label the event as an external event and send it out after all the gates in the collection have been updated.

## *D.5 SDES Load Balancing Mechanism*

The most complicated part of the simulation system is the message processing system. The process of message reconstructing was explained in section 4.5.3 . The communication protocol that is designed to reconstruct the communication packets is explained in C.2. This section focuses on how to incorporate a load balancing system into the message processing system.

The dynamic load balancing (DLB) is designed for SDES to compensate for the lack of parallelism within the deadlock avoidance technique. DLB has the ability to move part of the simulation workload to another LP which will even out the calculation time among different LPs. The difficulties are all laid out in 4.6.2 they include message handling while performing DLB and synchronizing the distributed data after DLB. Both problems are addressed in detail in this section.

First of all, the message handling problem can be solved by creating a buffer eventlist. The DLB in SDES is only executed between two LPs. As a result, only part of a node is engaged in the load balancing operation. In order to maintain the functionality of the

rest of the node, the monitor process must handle messages regardless to the status of the load balancing. The implementation of the monitor process uses a large eventlist to store all the messages that are meant to be sent to a LP in balancing mode. However the events that are recognized as part of the load balancing operation will be passed on to target LP.

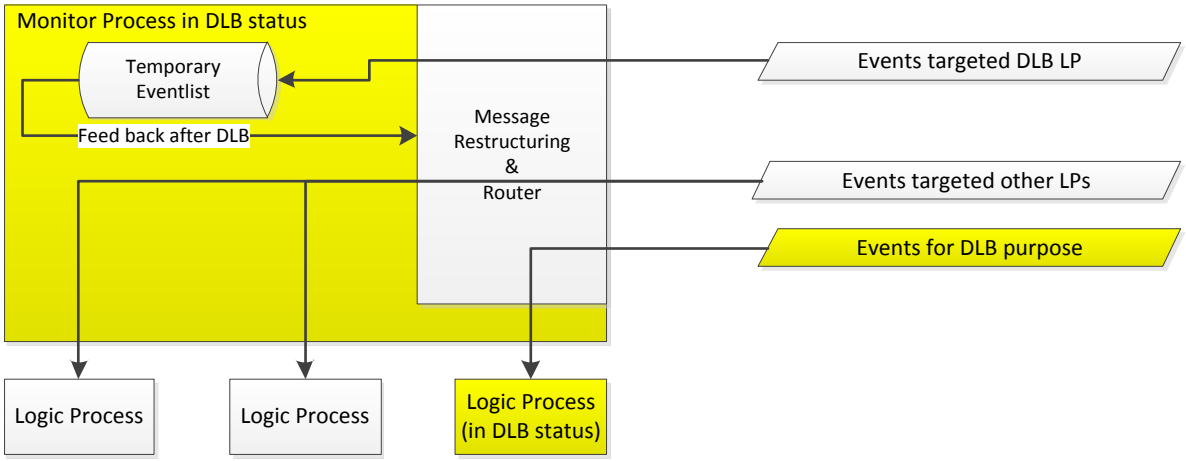



Figure 156 Monitor process message handling during DLB

The storage of messages allows the monitor to be able to direct messages sent during a DLB to the appropriate target LP after the DLB. There might be changes in the target LP for some of the LPs, so storing the events temporarily will guard against loss of events which will lead to incorrect simulation results. When the buffer eventlist has been processed and all the events within it have been sent to appropriate target LP, a guaranteed up-to-date eventlist will be in place for the target LP.

Secondly, synchronization of data is dealt with by sharing the history of components that are involved in the simulation. The transfer source LP in DLB will produce an event history associated with transferring components. However, a source LP may not have a complete history of all the components that are transferred. This may due to no event being executed on the gate or the component belongs to another LP. This requires the source LP to send an event history request to an external LP. This request will fetch the latest event on the gate from an external LP and the source LP will send it to the target LP in the batch. This operation will keep an up-to-date system state in the target LP. The flow of data is represented in Figure 157.

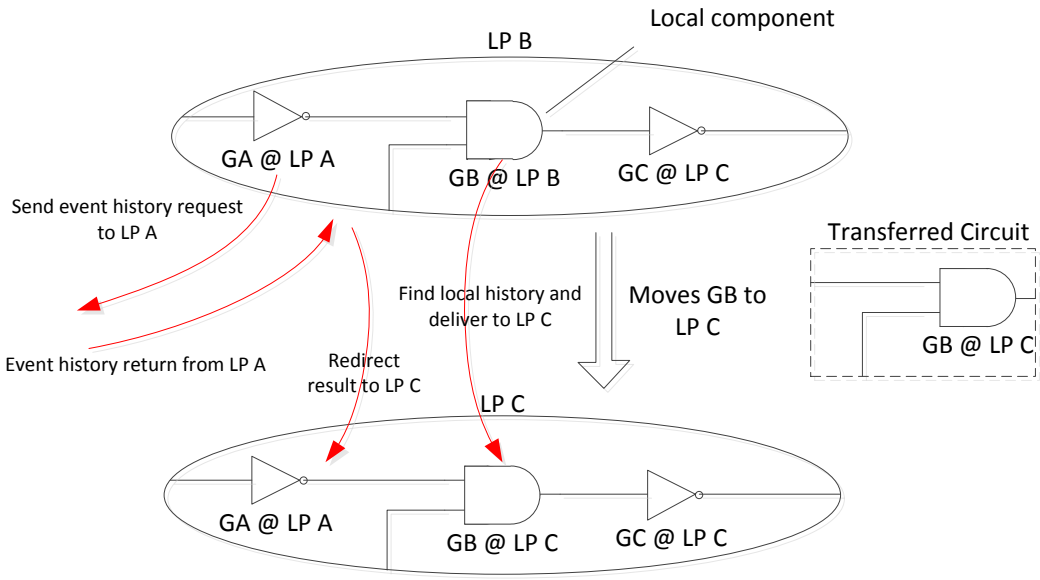



Figure 157 Event history flow during DLB

Thirdly, due to the nature of SDES, multiple DLB operations may be carried simultaneously. In order to present mutual disruption, a freezing mechanism is introduced. This mechanism ensures all nodes that have connection lead to the LPs in DLB status are free from any DLB operation requests. This avoids the risk of out-of-date partitioning information being held by the predecessor nodes. As the core simulation algorithm of SDES is the deadlock avoidance technique, the establishment of the boundary requires the partitioning information to be accurate. If multiple DLB operations are carried out at the same time, the system may run into a state where LP will be expecting null messages from the wrong source.

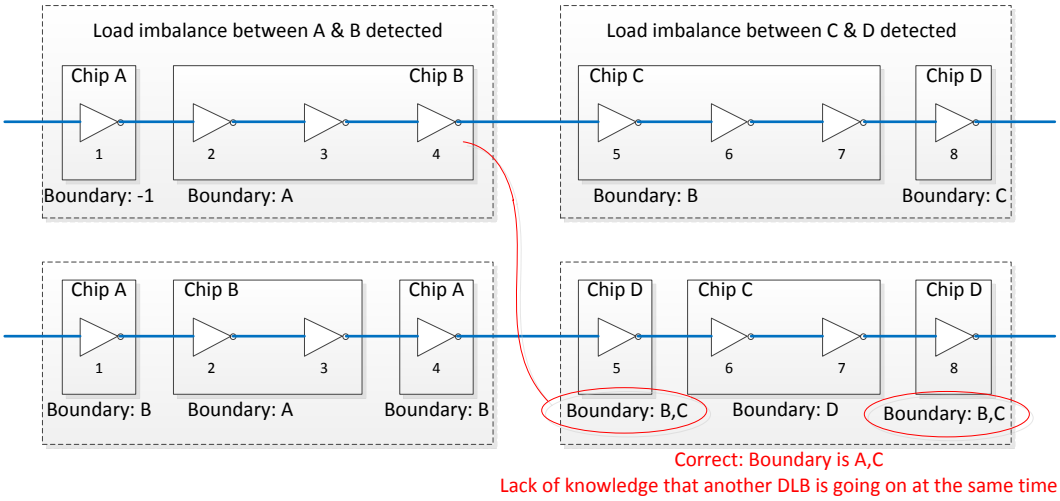



Figure 158 Concurrent DLB partition information mismatch case

Figure 158 shows an example where boundary information has been misinterpreted by the target cell. The two pairs of nodes, A & B and C & D carried out DLB concurrently. Due to the swift update of partitioning information, the C & D pair do not have the knowledge that gate 4 has been moved from node B to node A. This leads node D to recognize node B and node C as its boundary inputs where they should be node A and node C. As explained in the previous paragraph, the solution is to freeze the nodes which have input connections to the nodes that will carry out a DLB operation. This prevents the change of partition in the input nodes and hence guarantees the accuracy of the newly established boundary after DLB.

# List of References


[1] G. Moore, “Cramming more components onto integrated circuits,” in *Readings in Computer Architecture*, G. S. Hill, Mark D. and Jouppi, Norman P. and Sohi, Ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2000, pp. 56–59.

[2] K. M. Chandy and J. Misra, “Distributed simulation: A case study in design and verification of distributed programs,” *Software Engineering, IEEE Transactions on*, no. 5, pp. 440–452, 1979.

[3] D. M. Nicol, “Parallel Simulation Of FCFS Stochastic Queueing Networks,” in *Proceedings of the ACM/SIGPLAN conference on Parallel programming: experience with applications, languages and systems*, 1988, pp. 124–137.

[4] J. S. Steinman, “Discrete-event simulation and the event horizon,” *ACM SIGSIM Simulation Digest*, vol. 24, no. 1, pp. 39–49, 1994.

[5] R. De Vries, “Reducing null messages in Misra’s distributed discrete event simulation method,” *Software Engineering, IEEE Transactions on*, vol. 16, no. 1, pp. 82–91, 1990.

[6] D. Jefferson and H. Sowizral, *Fast concurrent simulation using the Time Warp mechanism, part I: Local control*. Defense Technical Information Center, 1982.

[7] D. Jefferson, “Virtual time,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 7, no. 3, pp. 404–425, Jul. 1985.

[8] “The BIMPA project.” [Online]. Available: http://apt.cs.man.ac.uk/projects/SpiNNaker/.

[9] S. Furber, S. Temple, and A. Brown, “On-chip and inter-chip networks for modeling large-scale neural systems,” *Circuits and Systems, 2006. ISCAS ’06. Proceedings of the 2006 International Symposium on*, 2006.

[10] A. D. Brown, “SpiNNaker (internal design notes),” 2008.

[11] S. Furber and A. Brown, “Biologically-inspired massively-parallel architectures-computing beyond a million processors,” *Application of Concurrency to System Design, 2009. ACSD’09. Ninth International Conference on*, pp. 3–12, Jul. 2009.

[12] M. M. Khan, D. R. Lester, and L. a. Plana, “SpiNNaker: mapping neural networks onto a massively-parallel chip multiprocessor,” *Neural Networks, 2008. IJCNN 2008.(IEEE World Congress on Computational Intelligence). IEEE International Joint Conference on*, pp. 2849–2856, Jun. 2008.

[13] F. Galluppi, A. Rast, S. Davies, and S. Furber, “A general-purpose model translation system for a universal neural chip,” *Proceedings of the 17th*

*international conference on Neural information processing: theory and algorithms - Volume Part I*, pp. 58–65, 2010.

[14] L. G. Birta and G. Arbez, *Modelling and Simulation : Exploring Dynamic System Behaviour*. Springer-Verlag London Ltd., 2007.

[15] S. Palnitkar, *Verilog HDL: A guide to digital design and synthesis*. Prentice Hall, 1996.

[16] B. Cohen, *VHDL Coding Styles and Methodologies*. Norwell, MA, USA: Kluwer Academic Publishers, 1995.

[17] M. D. Ciletti, *Advanced digital design with the Verilog HDL*, First. Prentice Hall, 2003, pp. 119–120.

[18] P. Ashenden, *The designer's guide to VHDL*. San Francisco: Morgan Kaufmann, 2008, p. 688.

[19] Mentor Graphics, "ModelSim ® User's Manual." 2010.

[20] R. M. Fujimoto, "Parallel discrete event simulation," *Communications of the ACM*, vol. 33, no. 10, pp. 30–53, 1990.

[21] N. M. Josuttis, *The C++ Standard Library - A Tutorial and Reference*. Addison-Wesley U.S.A., 1999, p. 832.

[22] A. Rushton, "The STL+ C++ Library Collection." [Online]. Available: http://stlplus.sourceforge.net/.

[23] P. Reynolds Jr. and P. R. Jr, "A spectrum of options for parallel simulation," in *Proceedings of the 1988 Winter Simulation Conference*, 1988, pp. 325–332.

[24] A. Sulistio, C. Yeo, and R. Buyya, "Simulation of Parallel and Distributed Systems: A Taxonomy and Survey of Tools," *2005-04. http://www. chinagrid. net*, pp. 1–19, 2002.

[25] R. Beraldi and L. Nigro, "Performance of a Time Warp based simulator of large scale PCS networks," *Simulation Practice and Theory*, vol. 6, no. 2, pp. 149–163, Feb. 1998.

[26] F. Wieland, L. Hawley, A. Feinberg, M. Di Loreto, L. Blume, J. Ruffles, P. Reiher, B. Beckman, P. Hontalas, S. Bellenot, and D. Jefferson, "The performance of a distributed combat simulation with the time warp operating system," *Concurrency: Practice and Experience*, vol. 1, no. 1, pp. 35–50, 1989.

[27] M. Presley, P. Reiher, and S. Bellenot, "A time warp implementation of sharks world," *Proceedings of the 1990 Winter Simulation Conference*, pp. 199–203, 1990.

[28] G. Yaun, C. Carothers, S. Adali, and D. Spooner, “Optimistic parallel simulation of a large-scale view storage system,” *Proceedings of the 2001 Winter Simulation Conference*, pp. 1363–1371, 2001.

[29] D. Jefferson, “Virtual Time II: Storage Management in Distributed Simulation,” *Proceedings of the 9th annual ACM symposium on Principles of distributed computing*, 1990.

[30] C. Young, “Optimism: not just for event execution any more,” *Proceedings of the 19th workshop on Parallel and distributed simulation*, no. Lcm, 1999.

[31] C. Young, N. Abu-Ghazaleh, and P. Wilsey, “OFC: a distributed fossil-collection algorithm for time-warp,” *Distributed Computing*, 1998.

[32] M. Chetlur and P. Wilsey, “Causality information and fossil collection in timewarp simulations,” *Proceedings of the 2006 Winter Simulation Conference*, pp. 987–994, 2006.

[33] V. Vee and W. Hsu, “PAL: a new fossil collector for time warp,” *Proceedings of the 16th workshop on Parallel and Distributed Simulation*, 2002.

[34] A. Gafni, “Rollback mechanisms for optimistic distributed simulation systems.,” in *SCS Multiconference on Distributed Simulation*, 1988, pp. 61–67.

[35] H. M. Soliman Ramadan, “Throttled Lazy Cancellation in Time Warp Parallel Simulation,” *Simulation*, vol. 84, no. 2–3, pp. 149–160, Feb. 2008.

[36] M. Abrams, “The object library for parallel simulation (OLPS),” *Proceedings of the 1988 Winter Simulation Conference*, pp. 210–219, 1988.

[37] Y. Lin and E. Lazowska, “A study of time warp rollback mechanisms,” *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, vol. 1, no. 1, pp. 51–72, Jan. 1991.

[38] V. Madisetti, J. Walrand, and D. Messerschmitt, “WOLF: A rollback algorithm for optimistic distributed simulation systems,” *Proceedings of the 1988 Winter Simulation Conference*, 1988.

[39] M. Chetlur and P. Wilsey, “Causality information and proactive cancellation mechanisms,” *Concurrency and Computation: Practice and Experience*, no. February, pp. 1483–1503, 2009.

[40] B. Lubachevsky, “Efficient distributed event-driven simulations of multiple-loop networks,” *Communications of the ACM*, vol. 32, no. 1, pp. 111–123, Feb. 1989.

[41] J. Steinman, “Breathing time warp,” *ACM SIGSIM Simulation Digest*, vol. 23, no. 1, pp. 109–118, Jul. 1993.

[42] L. M. Sokol, D. P. Briscoe, and A. P. Wieland, “MTW: A Strategy for Scheduling Discrete Simulation Events for Concurrent Execution,” in

*Proceedings of the SCS Multiconference on Distributed Simulation*, 1988, pp. 34–42.

[43] V. Madisetti, J. Walrand, and D. Messerschmitt, "MTW: Experimental Results for a Constrained Optimistic Scheduling Paradigm," *Proceedings of the SCS Multiconference on Distributed Simulation*, pp. 169–173, 1987.

[44] L. Sokol, "MTW: an empirical performance study," *Proceedings of the 1991 Winter Simulation Conference*, p. 557, 1991.

[45] Y. Meng, "Performance optimization of throttled time-warp simulation," *Proceedings. 34th Annual Simulation Symposium*, pp. 211–218, 2001.

[46] S. Srinivasan and P. F. Reynolds, "Elastic time," *ACM Transactions on Modeling and Computer Simulation*, vol. 8, no. 2, pp. 103–139, Apr. 1998.

[47] R. Suppi, F. Cores, and E. Luque, "An efficient method for improving large optimistic PDES," *Proceedings 8th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (Cat. No.PR00728)*, pp. 351–357, 2000.

[48] J. Steinman, "Incremental state saving in SPEEDES using C++," *Proceedings of the 1993 Winter Simulation Conference*, pp. 687 – 696, 1993.

[49] R. Schlagenhaft and M. Ruhwandl, "Dynamic load balancing of a multi-cluster simulator on a network of workstations," *Proceedings of the 9th workshop on Parallel and distributed simulation*, vol. 25, no. 1, pp. 175–180, 1995.

[50] H. Avril and C. Tropper, "The dynamic load balancing of clustered time warp for logic simulation," *ACM SIGSIM Simulation Digest*, vol. 26, no. 1, pp. 20–27, 1996.

[51] H. Rajaei, "Local Time Warp: An implementation and performance analysis," *Proceedings of the 21st International Workshop on Principles of Advanced and Distributed Simulation (PADS'07)*, 2007.

[52] K. Chandy and L. Lamport, "Distributed snapshots: determining global states of distributed systems," *ACM Transactions on Computer Systems (TOCS)*, vol. 3, no. 1, pp. 63–75, Feb. 1985.

[53] F. Mattern, "Efficient algorithms for distributed snapshots and global virtual time approximation," *Journal of Parallel and Distributed Computing*, vol. 18, no. 4, pp. 423–434, Aug. 1993.

[54] A. Nketsa and N. Khalifa, "Timed Petri nets and prediction to improve the Chandy–Misra conservative-distributed simulation," *Applied mathematics and computation*, vol. 120, no. 1–3, pp. 235–254, May 2001.

[55] D. M. Nicol and P. R. Jr, "Problem oriented protocol design," *Proceedings of the 1984 Winter Simulation Conference*, no. June 1979, pp. 470–474, 1984.

[56] Wentong Cai and S. J. Turner, “An algorithm for distributed discrete-event simulation-the `carrier null message’ approach,” in *Proceedings of the SCS Multiconference on Distributed Simulation*, 1990, pp. 3–8.

[57] K. M. Chandy and J. Misra, “Conditional Knowledge as a basis for Distributed Simulation,” *Journal of the Electrochemical Society*, vol. 129, p. 2865, 2006.

[58] K. Chandy and J. Misra, “Asynchronous distributed simulation via a sequence of parallel computations,” *Communications of the ACM*, vol. 24, no. 4, pp. 198–206, 1981.

[59] J. Misra, “Distributed discrete-event simulation,” *ACM Computing Surveys (CSUR)*, vol. 18, no. 1, pp. 39–65, Jun. 1986.

[60] Y. Teo and S. Tay, “Efficient algorithms for conservative parallel simulation of interconnection networks,” *Parallel Architectures, Algorithms and Networks, 1994. (ISPAN), International Symposium on*, pp. 286–293, 1994.

[61] W. Bain and D. Scott, “An algorithm for time synchronization in distributed discrete event simulation.,” *Proceedings of the SCS Multiconference on Distributed Simulation*, pp. 30–33, 1988.

[62] O. Berry and D. Jefferson, “Critical path analysis of distributed simulation.,” *SCS Conference on Distributed Simulation*, pp. 57–60, 1985.

[63] S. Srinivasan and P. Reynolds, “On critical path analysis of parallel discrete event simulations,” *Computer Science Report No. TR-93-29*, 1993.

[64] S. Lin, X. Cheng, and J. Lv, “State Causality Analysis of Conservative Parallel Network Simulation,” *Simulation Symposium, 2008. ANSS 2008. 41st Annual*, pp. 251–260, Apr. 2008.

[65] B. Groselj and C. Tropper, “The time-of-next-event algorithm.,” in *SCS Multiconference on Distributed Simulation*, 1988, pp. 25–29.

[66] a. Boukerche and C. Tropper, “SGTNE: semi-global time of the next event algorithm,” *Proceedings 9th Workshop on Parallel and Distributed Simulation (ACM/IEEE)*, pp. 68–77, 1995.

[67] R. Bryant, “Simulation of packet communication architecture computer systems,” no. MIT/LCS/TR-188, 1977.

[68] E. Naroska and U. Schwiegelshohn, “A New Scheduling Method for Parallel Discrete-Event Simulation,” in *Proceedings of the Second International Euro-Par Conference on Parallel Processing-Volume II*, 1996, pp. 582–593.

[69] R. Simmonds, C. Kiddle, and B. Unger, “Addressing blocking and scalability in critical channel traversing,” *Proceedings 16th Workshop on Parallel and Distributed Simulation*, pp. 15–22, 2002.

[70] A. Boukerche and S. K. Das, “Reducing null messages overhead through load balancing in conservative distributed simulation systems,” *Journal of Parallel and Distributed Computing*, vol. 64, no. 3, pp. 330–344, Mar. 2004.

[71] S. Rizvi, K. Elleithy, and A. Riasat, “A new mathematical model for optimizing the performance of parallel and discrete event simulation systems,” *Proceedings of the 2008 Spring simulation multiconference*, 2008.

[72] B. Thomas, S. Rizvi, and K. Elleithy, “Reducing null messages using grouping and status retrieval for a conservative discrete-event simulation system,” *Proceedings of the 2009 Spring Simulation Multiconference*, no. L, pp. 1–4, 2009.

[73] M. Chung, “Improving Lookahead in Parallel Multiprocessor Simulation Using Dynamic Execution Path Prediction,” *Workshop on Principles of Advanced and Distributed Simulation (PADS’06)*, pp. 11–18, 2006.

[74] E. Deelman, “Improving lookahead in parallel discrete event simulations of large-scale applications using compiler analysis,” *Proceedings of the 15th workshop on Parallel and distributed simulation*, pp. 5–13, 2001.

[75] V. Solcany and J. Safarik, “The lookahead in a user-transparent conservative parallel simulator,” *Proceedings 16th Workshop on Parallel and Distributed Simulation*, pp. 9–14, 2002.

[76] A. Park, R. M. Fujimoto, and K. S. Perumalla, “Conservative synchronization of large-scale network simulations,” *Workshop on Parallel and Distributed Smulation (PADS’04)*, pp. 153–161, 2004.

[77] R. Bagrodia and M. Takai, “Performance evaluation of conservative algorithms in parallel simulation languages,” *Parallel and Distributed Systems, IEEE Transactions on*, pp. 1–30, 2000.

[78] R. a. Meyer and R. Bagrodia, “An empirical study of conservative scheduling,” *Proceedings Fourteenth Workshop on Parallel and Distributed Simulation*, pp. 165–172, 2000.

[79] C. Kawabata and R. Santana, “Performance evaluation of a CMB protocol,” *Proceedings of the 2006 Winter Simulation Conference*, pp. 1012–1019, 2006.

[80] I. Cidon, J. Jaffe, and M. Sidi, “Local Distributed Deadlock Detection by Cycle Detection and Clusterng,” *Software Engineering, IEEE Transactions on*, vol. SE-13, no. 1, pp. 3–14, 1987.

[81] K. Chandy and R. Sherman, “The conditional-event approach to distributed simulation,” *Proceedings of the SCS Multiconference on Distributed Simulation*, 1989.

[82] J. K. Peacock, J. Wong, and E. Manning, “Distributed simulation using a network of processors,” *Computer Networks (1976)*, vol. 3, no. 1, pp. 44–56, 1979.

[83] S. Jafer and G. Wainer, “Global Lookahead Management (GLM) Protocol for Conservative DEVS Simulation,” *2010 IEEE/ACM 14th International Symposium on Distributed Simulation and Real Time Applications*, pp. 141–148, Oct. 2010.

[84] J. Liu, “Parallel Hybrid Network Traffic Models,” *Simulation*, vol. 85, no. 4, pp. 271–286, Apr. 2009.

[85] J. Dahmann, R. Fujimoto, and R. Weatherly, “The department of defense high level architecture,” *Proceedings of the 1997 Winter Simulation Conference*, pp. 142–149, 1997.

[86] B. Liu, Y. Yao, Z. Jiang, and L. Yan, “HLA-Based Parallel Simulation: A Case Study,” *Principles of Advanced and Distributed Simulation (PADS), 2012 ACM/IEEE/SCS 26th Workshop on*, pp. 65–67, Jul. 2012.

[87] Q. Xu and C. Tropper, “XTW, a parallel and distributed logic simulator,” *Workshop on Principles of Advanced and Distributed Simulation (PADS’05)*, pp. 181–188, 2005.

[88] S. Meraji, W. Zhang, and C. Tropper, “On the scalability and dynamic load-balancing of optimistic gate level simulation,” *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 29, no. 9, pp. 1368–1380, 2010.

[89] S. Meraji, W. Zhang, and C. Tropper, “On the scalability of parallel verilog simulation,” *Parallel Processing, 2009. ICPP’09. International Conference on*, pp. 365–370, Sep. 2009.

[90] B. W. Kernighan and S. Lin, “An efficient heuristic procedure for partitioning graphs,” *Bell System Technical Journal*, vol. 49, no. 2, pp. 291–307, 1970.

[91] C. M. Fiduccia and R. M. Mattheyses, “A linear-time heuristic for improving network partitions,” *Proceedings of the 19th Design Automation Conference*, pp. 241–247, 1982.

[92] L. a. Sanchis, “Multiple-way network partitioning with different cost functions,” *Computers, IEEE Transactions on*, vol. 42, no. 12, pp. 1500–1504, 1993.

[93] L. a. Sanchis, “Multiple-way network partitioning,” *Computers, IEEE Transactions on*, vol. 38, no. 1, pp. 62–81, 1989.

[94] J. Cloutier, E. Cerny, and F. Guertin, “Model partitioning and the performance of distributed timewarp simulation of logic circuits,” *Simulation Practice and Theory*, vol. 5, no. 1, pp. 83–99, Jan. 1997.

[95] D. Kolar, J. D. Puksec, and I. Branica, “VLSI circuit partition using simulated annealing algorithm,” *IEEE Melecon*, pp. 205–208, 2004.

[96] S. Dey, F. Brglez, and G. Kedem, “Corolla based circuit partitioning and resynthesis,” *Proceedings of the 27th ACM/IEEE Design Automation Conference*, pp. 607–612, 1990.

[97] W. Fang and A. Wu, “Multiway FPGA partitioning by fully exploiting design hierarchy,” *ACM Transactions on Design Automation of Electronic Systems*, vol. 5, no. 1, pp. 34–50, Jan. 2000.

[98] G. Karypis and V. Kumar, “Multilevel k-way hypergraph partitioning,” *VLSI design*, vol. 11, no. 3, pp. 285–300, 2000.

[99] C. J. Alpert, J. Huang, and A. B. Kahng, “Multilevel circuit partitioning,” *Proceedings of the 34th annual Design Automation Conference*, vol. 17, no. 8, pp. 655–667, 1998.

[100] L. Li and C. Tropper, “A design-driven partitioning algorithm for distributed verilog simulation,” *Principles of Advanced and Distributed Simulation, 2007. PADS’07. 21st International Workshop on*, pp. 211–218, Jun. 2007.

[101] P. Konas and P. Yew, “Parallel discrete event simulation on shared-memory multiprocessors,” *Simulation Symposium, 1991., Proceedings of the 24th Annual*, 1991.

[102] A. Boukerche and C. Tropper, “A static partitioning and mapping algorithm for conservative parallel simulations,” *ACM SIGSIM Simulation Digest*, pp. 164–172, 1994.

[103] B. Nandy and W. Loucks, “On a parallel partitioning technique for use with conservative parallel simulation,” *ACM SIGSIM Simulation Digest*, pp. 43–51, 1993.

[104] Y. Artsy and R. Finkel, “Designing a process migration facility: The Charlotte experience,” *Computer*, 1989.

[105] M. H. Willebeek-lemair, A. P. Reeves, and S. Member, “Strategies for dynamic load balancing on highly parallel computers,” *Parallel and Distributed Systems, IEEE Transactions on*, vol. 4, no. 9, 1993.

[106] P. Reiher and D. Jefferson, “Virtual time based dynamic load management in the time warp operating system,” *Transactions of the Society for Computer Simulation*, no. Jefferson 1987, 1990.

[107] L. Wilson and D. Nicol, *Experiments in automated load balancing*. 1996.

[108] S. Aote and M. Kharat, “A game-theoretic model for dynamic load balancing in distributed systems,” *Proceedings of the International Conference on …*, pp. 235–238, 2009.

[109] B. Zhang, Z. Mo, G. Yang, and W. Zheng, “Dynamic load balancing efficiently in a large-scale cluster,” *International Journal of High Performance Computing and Networking*, vol. 6, no. 2, pp. 100–105, Jul. 2009.

[110] C. Xu and F. Lau, “Iterative dynamic load balancing in multicomputers,” *Journal of the Operational Research Society*, vol. 45, no. 7, pp. 786–796, Jul. 1994.

[111] G. Cybenko, “Dynamic load balancing for distributed memory multiprocessors,” *Journal of Parallel and Distributed Computing*, vol. 7, no. 2, pp. 279–301, Oct. 1989.

[112] V. Saletore, “A distributed and adaptive dynamic load balancing scheme for parallel processing of medium-grain tasks,” *Distributed Memory Computing Conference*, pp. 994–999, 1990.

[113] D. Bertsekas and J. Tsitsiklis, “Parallel and distributed computation: numerical methods. 1989,” *Englewood Clifffs, New Jersey: Printice-Hall*, p. 715, 1997.

[114] F. C. H. F. Lin and R. Keller, “Gradient Model Load Balancing Method,” *Software Engineering, IEEE Transactions on*, vol. 13, no. 1, pp. 32–38, 1987.

[115] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: portable parallel programming with the message passing interface*. MIT Press, 1999.

[116] W. Gropp, E. Lusk, and R. Thakur, *Using MPI-2: Advanced features of the message passing interface*. MIT Press, 1999.

[117] H. Frese and H. Knipp, “Performance Evaluation of MPI and MPICH on the Cray T3E,” *Cray User Group Proceedings*, pp. 1–7, 1999.

[118] “Data Encryption Standard (DES),” *Federal Information Processing Standards Publication*, vol. 2, no. 46, p. 1, 1999.

[119] “ModelSim, ASIC and FPGA design.” [Online]. Available: http://www.mentor.com/products/fv/modelsim/.

[120] IEEE Standards Association, “IEEE Standard for Verilog Hardware Description Language,” *IEEE Std 1364-1995*, 2005.

[121] D. Subcommittee, “Ieee standard vhdl language reference manual,” *IEEE Std*, 1988.

[122] K. Kerry, P. Ashenden, and M. Oudshoorn, “STEVE: A Syntax-Directed Editor for VHDL Based on SAVANT,” *Proceedings VHDL International Users’ Forum*, 1997.

[123] “Wave VCD Viewer.” [Online]. Available: http://www.iss-us.com/wavevcd/index.htm.

[124] “Dinotrace.” [Online]. Available: http://www.veripool.org/wiki/dinotrace.

[125] "GTKWave." [Online]. Available: http://gtkwave.sourceforge.net/.

[126] "Scansion." [Online]. Available: http://www.logicpoet.com/scansion/.

[127] N. Davis, D. Mannix, W. Shaw, and T. Hartrum, "Distributed discrete-event simulation using null message algorithms on hypercube architectures," *Journal of Parallel and Distributed Computing*, vol. 357, pp. 349–357, 1990.

[128] G. Pfister, "An introduction to the InfiniBand architecture." 2001.

[129] D. Lungeanu and C. J. R. Shi, "Parallel and distributed VHDL simulation," *Design, Automation and Test in Europe Conference and Exhibition 2000. Proceedings*, pp. 658–662, 2000.

[130] S. Meraji and C. Tropper, "Optimizing Techniques for Parallel Digital Logic Simulation," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 23, no. 6, pp. 1135–1146, 2012.

[131] J. Goux, J. Linderoth, M. Yoder, and J. Goux, "Metacomputing and the master-worker paradigm," *Preprint MCS/ANL-P792-0200, Feb 2000 Mathematics and Computer Science Division Argonne National Laboratory*, no. February, 2000.

[132] A. Park and R. Fujimoto, "Efficient Master/Worker Parallel Discrete Event Simulation on Metacomputing Systems," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 23, no. 5, pp. 873–880, May 2012.

[133] A. Park and R. Fujimoto, "Efficient master/worker parallel discrete event simulation," *Principles of Advanced and Distributed Simulation, 2009. PADS'09. ACM/IEEE/SCS 23rd Workshop on*, pp. 145–152, Jun. 2009.

[134] R. Garg, V. Garg, and Y. Sabharwal, "Scalable algorithms for global snapshots in distributed systems," *Proceedings of the 20th annual international conference on Supercomputing*, 2006.

[135] R. Garg, V. Garg, and Y. Sabharwal, "Efficient algorithms for global snapshots in large distributed systems," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 21, no. 5, pp. 620–630, May 2010.

[136] D. B. Jr, C. Carothers, and A. Holder, "Scalable time warp on blue gene supercomputers," *Proceedings of the 2009 ACM/IEEE/SCS 23rd Workshop on Principles of Advanced and Distributed Simulation*, pp. 35–44, Jun. 2009.

[137] A. Holder and C. Carothers, "Analysis of time warp on a 32,768 processor ibm blue gene/l supercomputer," *Proceedings of the 2009 ACM/IEEE/SCS 23rd Workshop on Principles of Advanced and Distributed Simulation*, 2008.

[138] A. C. Williams, A. D. Brown, and Z. Baidas, "Optimisation in behavioural synthesis using hierarchical expansion: module ripping," *Computers and Digital Techniques, IEE Proceedings*, vol. 148, no. 1, pp. 31–43, 2001.

[139] M. Sacker, A. D. Brown, A. J. Rushton, and P. R. Wilson, “A behavioral synthesis system for asynchronous circuits,” *IEEE Transactions on Very Large Scale Itegration (VLSI) Systems*, vol. 12, no. 9, pp. 978–994, 2004.

[140] D. Bryan, “The ISCAS’85 benchmark circuits and netlist format,” *North Carolina State University*, no. June, pp. 695–698, 1985.

[141] B. Stroustrup, *The C++ Programming Language*, Special ed. Reading, Mass.: Addison-Wesley U.S.A., 2000, p. 1019.