UNIVERSIDAD COMPLUTENSE DE MADRID
FACULTAD DE INFORMÁTICA
Departamento de Arquitectura de Computadores y Automática

TESIS DOCTORAL

Optimización de la ejecución de aplicaciones en entornos heterogéneos de computación de altas prestaciones

Applications execution optimization in heterogeneous high performance computing environments

MEMORIA PARA OPTAR AL GRADO DE DOCTOR

PRESENTADA POR

Richard Michael Wallace

Directores

José Luis Vázquez-Poletti
Daniel Mozos Muñoz

Madrid, 2017

© Richard Michael Wallace, 2016

Optimización de la Ejecución de Aplicaciones en Entornos Heterogeneos de Computación de Altas Prestaciones

TESIS DOCTORAL

Tesis presentada en cumplimiento parcial de los requisitos para el grado de Doctorado en Informática

Richard M. Wallace

Dirigida por los profesores

Dr. José Luis Vázquez-Poletti
Dr. Daniel Mozos Muñoz

Departamento de Arquitectura de Computadores y Automática
Facultad de Informática
Universidad Complutense de Madrid

2016

Application Execution Optimization in Heterogeneous High Performance Computing Environments

DOCTORAL THESIS

A thesis submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Architecture

Richard M. Wallace

Supervisory professors

Dr. José Luis Vázquez-Poletti
Dr. Daniel Mozos Muñoz

Computer Architecture and Automation Department
Faculty of Computer Science
Universidad Complutense de Madrid

2016

Acknowledgements

To my advisor, Dr. José Luis Vázquez-Poletti, who has been a gracious and good guide, and a firm believer in this work. It is by God's grace that we stood in the food line together at Super-Computing 2010 in New Orleans and I first outlined to my research ideas — and José didn't laugh . . .

To Dr. Daniel Mozos Muñoz who has provided guidance and support for this work. I would also like to thank the far flung international members of my writing teams who enabled me to successfully publish. Each of them has been a pleasure to work with:

- Mehdi Sheikhalishahi and Lucio Grandinetti from the University of Calabria, Italy
- Ginés Guerrero, José Cecilia, José M. García, and Horacio Pérez-Sánchez from the University of Murcia, Spain
- Volodymyr Turchenko, Iryna Turchenko, and Vladyslav Shults from the Ternopil National University, Ukraine
- Patrick Martin from Queen's University, Canada

Dedication

First to God,

“ I can do all things through Christ who strengtheneth me.” — Philippians 4:13

To my wife, Cathryn, who has seen to it that our family runs smoothly while I was all too busy creating this work.

To my daughter, Gwyn, who watched me work through this process for almost her entire childhood.

“Wherever you go, there you are.”

The Adventures of Buckaroo Banzai Across the 8th Dimension

Abstract

High performance computing systems are constructed from single system, cluster, or cloud resources that require writing software to treat these resources as a homogeneous compute platform although these systems are typically composed of heterogeneous computing elements. These aggregated systems depend on carefully controlled, complex, distributed management software or hardware control units. System development for high performing computing systems shows that the distinction between hardware and software is blurred. With increasing performance expectations there is an increasing need to exploit all computing elements (CE) that are CPUs, GPUs, and other multicore systems on chip (SoC). Mainstream computing architectures are based on cache-coherent multicore processors and conventional tools used for concurrent and parallel software for such multicore systems are largely based on lock and monitor abstractions developed for writing operating systems, which are not the right tools for parallel application developers. Rather than composing many elements that look like regular CPUs, a better approach, from a latency and energy-consumption perspective, is to use a diverse collection of processing elements working in concert and "tuned" to perform different types of computation and communication. These combined systems are produced as heterogeneous system architectures (HSA). As such, developers rely on source language pragmas, inter-process communication libraries, compiler directives, and link-loader directives to control programs executed on the appropriate computational processor performing empirical, manual tuning cycles for system for latency reduction.

This thesis describes the novel Regulated Isomorphic Temporal Architecture (RITA) condition-event matrix system using formal and temporal mathematics and cloud system algorithm mapping techniques for the complex cloud system management, programming, compiler design, network-on-chip (NoC), and long haul TCP/IP networking environments where these latency issues are addressed. RITA minimizes latency by reducing "chatter" between CE communications allowing for maximal application processing time. By examination and control of the conditions that create communication events, the actual information content increases as data volume decreases. The thesis describes creation methods for regulating and removing redundant events allowing only transmission of informational data that needs processing.

Current work with auto-tuners shows that there is little attention to federated, distributed methods of determining the optimal mapping of execution units across cloud-deployed computational platforms. The thesis concentrates on models of computation and allocation methods for executable components allocated across federated, distributed computation systems constructed from homogeneous and heterogeneous compute elements establishing the ability to identify workflow partitioning for algorithms through use of an ontological representation.

The thesis creates a cloud computing ontology for allocation of algorithms to computing elements mapping the available computing elements and services to the cloud environment. After ontological mapping, a physical mapping using an iterative algorithmic process is shown. The algorithm code base is scored for algorithmic match to workflow patterns. Scoring each of the categories involves scoring each of the patterns based on the number of specific instructions that that belong to the pattern, standardizing the values and then normalizing the values and calculating an overall score. This score maps an algorithm to a CE. The thesis provides a delayed differential equation calculation from a modified Lotka-Volterra formula giving a CE message carrying capacity, or equilibria, for the resources available and the latency inherent in the algorithm.

Using the RITA specification notation, an implementation of the continuous neural network work cited in prior work is used as a demonstration experiment. The demonstration uses the OMNeT++ cycle-accurate simulator for simulating communication at the NoC level for heterogeneous cloud-computing configurations. The simulation resulted in a 93-98% reduction in “chatter” without degrading data processing quality. The thesis describes future work topics for expanding RITA for a multi-data center model, determining algorithm data sensitivity, building a general-purpose compiler for RITA, and a catastrophic data loss fault-tolerance for RITA.

Resumen

Los sistemas de computación de altas prestaciones se componen de una simple máquina, de un clúster o de recursos cloud que requieren una reescritura total del código para tratar estos entornos como una plataforma de computación homogénea, incluso cuando estos sistemas están compuestos típicamente de elementos de computación heterogéneos. Estos sistemas agregados dependen de un software de gestión cuidadosamente controlado, complejo y distribuido, o de unidades de control hardware.

El desarrollo de sistemas para la computación de altas prestaciones nos muestra que la diferencia entre hardware y software es bastante difusa. El incremento de la demanda de prestaciones va acompañado de una creciente necesidad de explotar todos los elementos de computación que son CPUs, GPUs, así como otros sistemas multicore en chip. Los sistemas de computación más populares están basados en procesadores multicore con coherencia en la cache. Las herramientas convencionales usadas para el software concurrente y paralelo para dichos sistemas multicore están basados en abstracciones de bloqueo y monitorización, desarrolladas para diseñar sistemas operativos, no siendo adecuadas para los desarrolladores de aplicaciones paralelas.

En vez de componer varios elementos que asemejan a CPUs estándar, una mejor estrategia desde el punto de vista de la latencia y el consumo de energía, sería usar una colección variada de elementos de proceso optimizados para realizar diferentes tipos de cálculos y comunicación. Estos sistemas combinados son conocidos como arquitecturas de sistemas heterogéneos. Los desarrolladores se apoyan en pragmas definidos por el lenguaje de programación, librerías de comunicación entre procesos, directivas de compilación, y directivas de carga-enlazado para controlar la ejecución de programas en el procesador apropiado, mediante unos ciclos de optimización que permiten reducir la latencia.

La presente tesis doctoral describe la Arquitectura Temporal Isomórfica Regulada (Regulated Isomorphic Temporal Architecture, RITA), un sistema basado en matrices de condición-evento usando matemáticas formales y temporales, así como algoritmos de mapeo para la gestión compleja de sistemas cloud, programación, diseño de compiladores, network-on-chip (NoC), y entornos TCP/IP con gran ancho de banda, donde la latencia supone un problema.

RITA minimiza la latencia reduciendo las comunicaciones innecesarias entre elementos de computación, permitiendo el máximo tiempo de proceso para la aplicación. El tamaño de la información aumenta a la vez que el volumen de datos disminuye, gracias a

la inspección y control de las condiciones que crean los eventos de comunicación. La presente tesis describe los métodos de creación para regular y eliminar eventos redundantes, permitiendo la transmisión de exclusivamente los datos que necesitan ser procesados.

Los avances actuales en optimizadores automáticos muestran un escaso interés en métodos distribuidos y federados para el mapeo óptimo de unidades de ejecución en plataformas desplegadas gracias al cloud. Esta tesis se centra en modelos de computación y métodos de asignación para componentes ejecutables en sistemas de computación distribuidos y federados, que han sido desplegados a partir elementos de computación tanto homogéneos como heterogéneos, usando una representación ontológica para el particionado de flujos de trabajos.

La presente tesis crea una ontología para el cloud que permite la asignación de algoritmos a elementos de computación desplegados mediante esta tecnología. Después del mapeo ontológico, se muestra otro de carácter físico que emplea un proceso algorítmico iterativo. El código algorítmico es puntuado en función de su adecuación a determinados patrones de flujos de trabajos. Esta puntuación conlleva puntuar a su vez cada patrón basándose en el número de instrucciones específicas que pertenecen al mismo, normalizando los valores y así obteniendo la puntuación final. Esta puntuación mapea el algoritmo a un elemento de computación. La presente tesis ofrece una ecuación diferencia retardada a partir de una fórmula Lotka-Volterra modificada, asignándole una capacidad de envío de mensajes a un elemento de computación, o equilibrio, según los recursos disponibles y la latencia inherente al algoritmo.

En este documento, se muestra como caso de uso una implementación de una red neuronal continua, perteneciente a un trabajo previo, usando la notación de la especificación RITA. La demostración emplea el simulador OMNeT++ para las comunicaciones al nivel de NoC para configuraciones cloud heterogéneas. La simulación muestra una reducción del 93-98% de las comunicaciones superfluas sin degradar la calidad del proceso de datos.

La presente tesis finaliza describiendo el trabajo futuro que podría realizarse para expandir RITA en diferentes direcciones, como modelos de múltiples centros de procesos de datos, determinación de la sensibilidad de datos en un algoritmo, construcción de un compilador de propósito general, o incluso diseño de técnicas de tolerancia de fallos.

Contents

Abstract iv

Resumen vi

Contents viii

1 Overview 1
- 1.1 Motivation 1
- 1.2 Background 2
- 1.3 Summary of Contributions 4
- 1.4 Prior Publications 5
  - 1.4.1 Applications of Neural-based Spot Market Prediction for Cloud Computing 6
  - 1.4.2 Spot Price Prediction for Cloud Computing Using Neural Networks 9
  - 1.4.3 Regulated Condition-Event Matrices for Cloud Environments 11
  - 1.4.4 Autonomic Resource Contention-Aware Scheduling 12
  - 1.4.5 A Multi-capacity Queuing Mechanism in Multi-dimensional Resource Scheduling 15
  - 1.4.6 A Multi-Dimensional Job Scheduling 21
  - 1.4.7 A Performance/Cost Model for a CUDA Drug Discovery Application on Physical and Public Cloud Infrastructures 22
  - 1.4.8 Consideration of the TMS320C6678 Multi-Core DSP for Power Efficient High Performance Computing 25
- 1.5 Structure and Flow of Thesis Chapters 27

2 Theoretical Background 28
- 2.1 Overview 28
- 2.2 Complex Event Systems (CES) 29
  - 2.2.1 Probabilistic Events 30
  - 2.2.2 Event Precedence 31
  - 2.2.3 Event Context 31
  - 2.2.4 Function Placement 33
  - 2.2.5 Consistency 33
- 2.3 Distributed Event Based Systems (DEBS) 33
- 2.4 RITA Theory 36
  - 2.4.1 Events and Event Propagation 36
  - 2.4.2 Latency 41
    - 2.4.2.1 Markov Models 41

2.4.2.2 Delay Differential Equations 42
2.4.2.3 Latency Cost Functions 45
2.4.2.3.1 Processor Type, Cache, RAM 48
2.4.2.3.2 Network and Network-on-Chip 49
2.4.2.3.3 General Latency Equations 50
2.4.2.3.4 NoC Network Interconnects 52
2.4.2.4 RITA Latency Cost Function 57
2.4.3 Informal Event Model 60
2.4.3.1 Spike Event 61
2.4.3.2 Set-At 61
2.4.3.3 Transitional 62
2.4.4 Event Form Use 62
2.4.5 Formal Event Model 63
2.4.5.1 Spike Event Form 64
2.4.5.2 Set-At Event Form 64
2.4.5.3 Transitional Event Form 65
2.4.6 Condition-Event Matrix 65
2.4.7 Explicit Time 66
2.4.8 Temporal Logic Model 67
2.4.8.1 Deadlock Prevention 68
2.4.8.2 Temporal Constructs 68

3 Algorithm to Computational Architecture Mapping 72
3.1 Overview 72
3.2 Partitioning Patterns 73
3.2.1 Berkeley “Dwarfs” 73
3.2.2 Work-flow Patterns 77
3.3 Cloud Computing Ontology 83
3.3.1 Current Cloud System Allocations 85
3.3.2 Representation of Allocations in a Semantic Web 85
3.3.3 Agent Supplied Cloud System Data 90
3.3.3.1 Open Source IaaS and PaaS 93
3.3.3.2 Integration of RITA into OpenShift 97
3.3.4 Ontological Weighted Match for Algorithms to CE 98
3.4 Processing Allocation to Cloud Components 101
3.4.1 Algorithm Recognition and Mapping to CEs 103
3.4.2 Decomposition Mapping Example Using Tower of Hanoi 106
3.4.3 Worker-Dispatch Mapping Example 109
3.4.4 DDE Input Data for Algorithm Mapping 112
3.4.4.1 Latency in cloud provider systems: Part 1 of $\mathcal{L}_\mathcal{N}$ 112
3.4.4.2 Long-haul latency between data centers: Part 2 of $\mathcal{L}_\mathcal{N}$ 114
3.4.4.3 NoC Latency internal to CEs: Calculating $\mathcal{L}_\mathcal{S}$ 116
3.4.4.4 Heterogeneous Computing Element Compilers; Computing $\mathcal{L}_\mathcal{P}$ 125
3.4.5 Consideration of Software Defined Networks for $\mathcal{L}_\mathcal{N}$ 128
3.4.6 Example RITA DDE Operational System 129
3.4.7 DDE Calculated Optimal Message Rate 137

3.4.8 RITA Notation for Example CTNN Application 141

4 Experiment, Simulation, and Analysis 145

4.1 Choice of Simulation Experimental Environment 145

4.2 Simulation Description 151

4.3 Simulation Results 158

4.4 Analysis 162

5 Summary, Conclusions, and Future Work 164

5.1 Summary 164

5.2 Conclusions 165

5.3 Future Research 166

A RITA Language Syntax 168

B RITA Language Example Specification 173

C Formula Elaboration 177

C.1 The Malthusian model 177

D Computing Element OWL XML 178

E VXDL BNF 181

F Tower of Hanoi in Java and Go 184

F.1 Java Implementation 184

F.2 Go Implementation 187

G Example Optimization Report for Intel OpenMP* 189

H High Speed TCP Variants 192

I Lotka–Volterra 194

J OMNeT++ NED Descriptions 196

List of Algorithms

1 Queue(Job j) 14

2 ScheduleCycle1(Time t) 15

3 ScheduleJob(Job j, Time t) 15

4 Top Level Mapping Algorithm 105

5 Add Control Flow to DAG 105

6 Control Flow Processing 106

List of Figures

1.1 MLP and RNN state transition models 10

1.2 BLUE1: Resource contention graph for 16 and 8 number of core processors. Lower resource contention is better. 20

1.3 BLUE2: Resource contention graph for 16 and 8 number of core processors. Lower resource contention is better. 20

1.4 DS: Resource contention graph for 4 and 2 number of core processors. Lower resource contention is better. 21

2.1 Network of Event Precedence 32

2.2 Parallel Event Arrival 37

2.3 Modeling of processing components as they relate to event propagation . 46

2.4 June 2014 Top 500 Interconnect Types 49

2.5 Three Transport Protocols 50

2.6 Comparison of IBM Cell BE™ and Intel Phi™ Block Diagrams 54

2.7 Comparison of Intel i5 and i7 Lynnfield family processor speed steps . . . 55

2.8 Levels of Oscillation based on Initial Conditions. Varying the magnitude of k, the corrective operation. 58

2.9 "Spike" Event Form. 61

2.10 "Set-At" Event Form. 62

2.11 Transitional Event Form. 62

2.12 Event Processing 63

2.13 Extended State Z schema . . . 69
2.14 ACK Channel Z schema . . . 69
2.15 Receive Z schema . . . 69
2.16 Transaction ACK Z schema . . . 70
2.17 Operation and History Z schema . . . 70
2.18 Operation and History Z schema . . . 70
2.19 Liveness of messages Z schema . . . 71
2.20 Liveness temporal Z schema . . . 71

3.1 Dwarf Application Areas . . . 75
3.2 Computing Element Ontology . . . 88
3.3 Decision Point Ontology . . . 89
3.4 Cloud Ontology . . . 90
3.5 Cloud Reference Model. . . . 91
3.6 Distributed Flow of Control with CA and RA. . . . 94
3.7 Comparison of VM and Container Memory Footprints . . . 95
3.8 RITA Processing Cell . . . 98
3.9 Multiple RITA Systems . . . 98
3.10 Ontological Similarity Stack . . . 99
3.11 CE Architecture Overlap . . . 99
3.12 DP Scoring . . . 101
3.13 Data Parallel Calculations . . . 103
3.14 Data Parallel DAG . . . 103
3.15 Partial Verizon Trans-Atlantic Cable Map . . . 115
3.16 Mesh NoC Topology . . . 117
3.17 Torus NoC Topology . . . 117

3.18 Butterfly Fat NoC Topology 117

3.19 Butterfly Fat Extension NoC Topology 117

3.20 Fat-Tree NoC Topology 118

3.21 3-Stage Butterfly NoC Topology 118

3.22 Kim and Kim latency comparison 120

3.23 Kim and Kim SPLASH2 and SpecCPU Benchmark Topology Latency . . 121

3.24 Kim and Kim, UR (left), LOC at 90% traffic (right) 122

3.25 Benchmarks used for flit performance 123

3.26 Speedup Versus Flit Size for Selected Benchmarks 124

3.27 Network activity by most exercised source-destination pairs 124

3.28 Average network latency with varying number of FCPs 125

3.29 Average network latency under directed traffic 125

3.30 Programmable networking chronological relationship to advances in network virtualization. 129

3.31 Overview of Long-haul Example DDE System 131

3.32 SR-IOV enabled and disabled 133

3.33 SR-IOV enabled and disabled with jitter error bars 133

3.34 SR-IOV enabled, disabled, and Native modes 133

3.35 Initial Condition Stable Equilibriua 137

3.36 Initial Condition Time-phased Populations 137

3.37 Equilibriua for m = 150, High Zero-load latency (1.245sec) 138

3.38 Time-phased Populations for m = 150, High Zero-load latency (1.245sec) 138

3.39 Equilibriua for m = 60 139

3.40 Time-phased Populations for m = 60 139

3.41 Equilibriua for m = 100 139

3.42 Time-phased Populations for m = 100 139

3.43 Equilibriua for m = 150 . . . 140

3.44 Time-phased Populations for m = 150 . . . 140

3.45 Equilibriua for m = 200 . . . 140

3.46 Time-phased Populations for m = 200 . . . 140

3.47 Equilibriua for m = 250 . . . 140

3.48 Time-phased Populations for m = 250 . . . 140

3.49 Equilibriua for m = 300 . . . 141

3.50 Time-phased Populations for m = 300 . . . 141

3.51 Equilibriua for m = 350 . . . 141

3.52 Time-phased Populations for m = 350 . . . 141

4.1 Chip Socket Interconnect Xeon E5-2670 v2 (Ivy Bridge) . . . 148

4.2 Core Interconnect Xeon E5-2670 v2 (Ivy Bridge) . . . 148

4.3 nVidia Kepler GK110 Architecture . . . 149

4.4 nVidia GK110 Streaming Multiprocessor (SMX) Cell . . . 150

4.5 ARM7 Marvell ARMADA-XP® (MV78460) Architecture . . . 151

4.6 Top-level OMNeT++ Cluster Architecture . . . 151

4.7 OMNeT++ Xeon Architecture Representation. Both Xeon1 and Xeon2 are identical and only Xeon1 is shown. The term "node" is used to indicate a core top-level view in a NoC . . . 152

4.8 OMNeT++ nVidia Architecture Representation. The intermediary memory representation is subsumed in the SMX module . . . 152

4.9 OMNeT++ MaxwellXP Architecture Representation. The intermediary memory representation is subsumed in the ARM7 module . . . 153

4.10 Message state transition for ContinuousNN . . . 154

4.11 *StartMsg* Message Cascade . . . 155

4.12 *newTraining* Message Cascade . . . 156

4.13 *computeNN* Message Cascade . . . 157

4.14 Unconstrained All Message Event Trace 159

4.15 Condition-Event Constrained All Message Event Trace 160

4.16 *valueSend* Message Overlap Event Trace 161

C.1 Malthusian Growth Rate 177

List of Tables

1.1 Numerical analysis of MLP prediction results 9

1.2 HPC characteristics distribution in workload trace 17

1.3 Platforms System Specifications. 24

1.4 Potential Ultrascale Processors 27

2.1 Major Latency Components 47

2.2 Current Heterogeneous System Architectures (HSA) 48

3.1 Original Seven Dwarfs 75

3.2 Additional Dwarfs 76

3.3 Workflow Control Patterns 82

3.4 WCP to Decision-Point Matching 83

3.5 OpenShift Subsystems 95

3.6 OpenShift Term Definitions 97

3.7 OpenShift, RITA Comparison 98

3.8 Amazon Web Services (AWS) `iperf` data transfer 112

3.9 Google Compute Engine `iperf` data transfer 113

3.10 Rackspace `iperf` data transfer 113

3.11 Verizon Long-haul Network Latency 115

3.12 Traceroute from West Chester, Ohio to University of London, England . 116

3.13 Comparison of Latency Time Magnitudes 117

3.14 NoC Latency Paramenters (multiple units) 122

3.15 Example System $\mathcal{L}_{\mathcal{N}}$ Values 134

3.16 Serial and Parallel portions of Example Application 135

3.17 Instruction Types, Instruction Density, and Parallel/Serial Allocation . . 135

3.18 Calculations for J_D for $\mathcal{L}_{\mathcal{P}}$ 136

3.19 Calculation Check for Total Processing Time 136

3.20 Summary $\mathcal{L}_{\mathcal{X}}$ Empirical Values 137

3.21 Stablized Lotka-Volterra DDE 137

3.22 MATLAB variable settings for Lotka-Volterra DDE 139

4.1 Unconstrained Message processing 158

4.2 Condition-event Constrained Message processing 158

4.3 Percent Reduction in Messages 158

4.4 Cumulative Message Times (seconds) 161

Chapter 1

Overview

1.1 Motivation

CURRENT systems are grouped into two large categories. Systems that are highly specialized computing systems, supercomputing systems, and systems that are large federated commodity computing systems, grid, cluster, or cloud computing systems. These categories can be fluid due to crossover of technology from grid and cloud computing that provides utility to supercomputing systems and vice-versa. The industry also makes categorization difficult due to name confusion, how a computing system is viewed by its using community, or by vendors who market technology of computing systems to sell to a specific market, such as the LexisNexis® Data Analytics Supercomputer™[1] which is a parallel array of commodity processors for data parallel processing, which is in short a cluster computer.

Supercomputing system designs have evolved over time from single processing systems to systems with massive numbers of processors in localized clusters with homogeneous multi-core processors. Recent supercomputer designs have shown good scaling across homogeneous nodes where each node is a heterogeneous composition of GPP and GPU processors on the same die [YXMZ12] associated with high speed NUMA memory. These systems can also be seen as grid computing systems when networked with other similar systems across diverse administrative domains where processing is done opportunistically when computing resources are available.

Typically, grid computing is a federation of computing resources from multiple sites used to process data independently for a common goal. Grids are prototypically distributed with non-interactive workloads having Single Instruction, Single Data (SISD),

[1] http://www.lexisnexis.com/government/solutions/data-analytics/supercomputer.aspx

Single Instruction, Multiple Data (SIMD), Multiple Instruction, Multiple Data (MIMD), and on very rare occasions Multiple Instruction, Single Data (MISD) computing models. MISD computing is used in computational biology or cryptography. A differentiator between supercomputing systems and grid systems is that grid systems are loosely coupled, and geographically dispersed. With regard to the point made earlier, this is not always clear due to technology crossover and operational use.

1.2 Background

APPLICATION development has shown that the distinction between hardware and software is blurred. With the struggle to meet the performance expectations of current systems there is an increasing need to exploit all computing elements (CE). Commonly these are GPUs and FPGAs or microcoding for performance improvements with Intel, AMD, nVidia, and even Xilinx computing elements.

Mainstream CPU/GPP computing architectures are based on cache-coherent multicore processors. Variations on this theme include Intel's experimental Single-Chip Cloud Computer, which contains 48 cores that are not cache coherent. This program was retired by Intel in 2013 with the introduction of the Xeon Phi™ product family, which contains up to 61 cores. Xeon Phi™ implements a very high bandwidth memory subsystem where each core has a 32KB L1 instruction cache, a 32KB L1 data cache, and a 512KB unified L2 cache. These caches are fully coherent and implement the x86-memory order model.

The conventional tools used for concurrent and parallel software for such multicore systems are largely based on lock and monitor abstractions developed for writing operating systems, which are not the right tools for parallel application developers. Rather than composing many elements that look like regular CPUs, a better approach, from a latency and energy-consumption perspective, is to use a diverse collection of processing elements working in concert and tuned to perform different types of computation and communication. Large coarse-grain tasks are suitable for implementation on multicore processors. Thousands of fine-grain data-parallel computations are suitable for implementation on GPUs. Irregular fine-grain tasks needing extreme performance requirements or reduced energy consumption are suitable as implementation as digital circuits running on FPGA chips.

The heterogeneous cloud computing environment has already been deployed in the Amazon Elastic Compute Cloud with configurations that have GPUs[2]. Some kinds of

[2] http://aws.amazon.com/ec2/instance-types/

computations can be executed on GPUs and FPGAs at a performance-per-dollar ratio that is significantly better than what is achievable with a CPU. As energy use becomes more of a limiting factor in the growth of data centers, the deployment of heterogeneous architectures to help reduce latency and energy consumption will be inevitable. nVidia product specifications show that, compared to the Intel quad-core CPUs, the Tesla C2050 and C2070 processors delivered equivalent performance at 1/10th the cost and 1/20th the power consumption. With GPU chip sets showing improved efficiency there are still improvements that can be achieved. In work done at Oak Ridge National Laboratory [WVC+11], digital signal processing (DSP) chip sets were evaluated as computational elements that have significant throughput and power efficiency that is better than CPU chip sets. So, to be effective, current heterogeneous computational architectures using GPU accelerators [BDH+10a] require extensive system knowledge and esoteric programming methods. These systems are composed of commodity processors integrated with Field Programmable Gate Arrays (FPGA) and/or Graphics Programming Units (GPU). With newer DSPs a viable alternative to the FPGA/GPU combinations exist and developers can avoid the performance problems associated with integrating accelerators into computer systems.

To illustrate this point of device complexity in melding disparate computing element architectures, AMD has created an accelerated processing unit (APU) combining CPU and GPU design on the same die facilitating physical access to common processor caches, address registers, physical L1 and L2 memory, and system memory. This combined system is in production under the banner of heterogeneous system architecture (HSA) sponsored by an industry consortium of AMD, ARM, Imagination, Mediatek, Qualcomm, Samsung, and Texas Instruments[3]. In a recent trade press article on-line through ExtremeTech[4], a Ziff-Davis publication, the difficulty in constructing an APU is detailed[5]. The AMD "Steamroller" design (*c.* Sep. 2011) has a 96KB shared cache that is three-way associative and cache conflicts remain a significant problem – when two different threads are running in the same module, they can overwrite each others' code. With Kaveri, the latest iteration of the AMD Bulldozer architecture, this issue seems to have been solved but there are still latency issues with small, but significant delays in address line traffic. These kinds of issues detract from the combination of CPU and GPU on the same dies. AMD is pushing language support and tools for the major languages (OpenCL, Java, C++ and others) as well as libraries for APIs to do cache and address management automatically with fewer lines of code. Kaveri will support OpenCL 2.0, which should make it the first CPU/APU/SoC to carry that certification.

[3] http://www.hsafoundation.com/

[4] http://www.extremetech.com/

[5] http://www.extremetech.com/computing/177099-secrets-of-steamroller-digging-deep-into-amds-next-gen-core

Although this is good news for heterogeneous processing, this all leads back to application execution optimization for correct processing on the correct computing elements; even when they are on the same die.

1.3 Summary of Contributions

This thesis covers topics in mathematics, integrated circuit design, cloud system management, programming, compiler design, and network-on-chip low-level to long-haul world-wide TCP/IP high-level networking. The core of work done in these topics considers the use of the Regulated Isomorphic Temporal Architecture (RITA) condition-event methods to achieve the goal of minimizing “chatter” between CE communications allowing for maximal application processing time. When using the word “chatter” it is important to note that it is not used as a dismissive. System “chatter” can be quite useful if it carries *information* semantics. All too often this communication is highly redundant event messages for “heart-beat” or “keep-alive” status or flow control messages that do not have any flow control effect. By examination and control of the conditions that create communication events, the actual information content increases as data volume decreases. Thus it is important to design a regulation method to remove these redundant events from the most basic communication — messages between processors using a Network-on-Chip (NoC) — all the way up to the data packets exchanged between data-centers on a TCP/IP network — whether it be a LAN, CAN, or WAN — so only actual, *informational* data that needs processing interrupts an application for input or output.

After an initial overview of prior work in §1, §2 details the theoretical background needed to form the formal foundation that allows RITA to operate at any strata of computing. In §3 extensive work is done providing a mapping from theoretical to available implementations. As the area of cluster computing, grid computing, and cloud computing are all rather synonymous, and each has its own jargon and preconceived framework, this thesis provides an agnostic method to differentiate the workflow control processing and the ontology for federated, distributed systems allowing mapping of the algorithm components across the computing fabric without use of currently fashionable terms tied to specific products in the market. In §4 a simulation of a cloud computing cluster is done to demonstrate and calculate the savings in processing time that RITA provides. The cycle accurate discrete event simulator OMNeT++[6] is used to capture the highly-overlapped message transmission lifetimes to measure message latency.

[6] https://omnetpp.org/

This work produced the following contributions to the literature:

- A definition, mathematical basis, and usage for RITA.
- A unique delayed-differential equation formula for heterogeneous processor cost functions.
- The new syntax and semantics for a RITA description language.
- New WorkFlow Control Pattern (WCP) and Decision-Point (DP) mapping algorithms for auto-partitioning applications across heterogeneous processing environments.
- A new ontology for WCP and DP for RITA distributed flow-control for a semantic network.
- Creation of decision-point algorithm definitions and DP scoring method to assign algorithms to CE types.
- A new extension of the Lotka-Volterra equations allowing predictive value assessments for establishment of steady state, maximal messaging in a RITA system.

1.4 Prior Publications

As part of writing teams for several papers on topics in the cloud computing domain and applications using cloud computing technology, my team members and I published eight papers developing concepts focused on cloud computing or technology that would be of benefit to cloud computing. In §1.4.1 the focus of this paper was on predicting the price of processing time of cloud processing offered by EC2 by Amazon using their auction system. In §1.4.2 a more in-depth treatment of the neural network from §1.4.1 was done for the MLP and RNN models used. In the paper in §1.4.3 the focus was on optimization of time- and value-based flow control for intensive processing applications. Resource contention-aware scheduling for cloud systems is the focus of three papers described in §1.4.4, §1.4.5, and §1.4.6. In §1.4.7 drug discovery by virtual screening was done for GPU enabled local and cloud systems and a cost comparison done to evaluate when a cloud instance should be employed for the drug discovery modeling. In conclusion, a technical report in §1.4.8 describes an alternative computing architecture not currently used for cloud computing or HPC but would be an asset in heterogeneous systems. The following subsections detail the ideas in prior work related to the motivation in §1.1.

1.4.1 Applications of Neural-based Spot Market Prediction for Cloud Computing

In the paper by Wallace, Turchenko, Sheikhalishahi, et.al. [WTS+13] the authors see cloud computing as the convergence of ideal characteristics of various distributed computing technologies. As cloud and grid computing systems are created they reuse known economic models; but these systems have wide variances due to many variables in both operation of the system and hosting and execution of client applications. These variances invalidate some of the preconceptions of existing economic models for time sharing systems. These systems use different business models that create new markets for these non-traditional time sharing systems providing profitability to the system owner and offered at a unit price that is attractive to users. The system must compete as a commodity system that is sufficiently profitable. This price-point is dependent on time-of-day use, number of concurrent users, and equipment and facilities capitalization cost.

Economic benefits of cloud and grid adoption are the main drivers as shown a study by Armbrust, et.al. in [AFG+09]. Initially, cloud providers had only a fixed price for their service offerings [CBA+05, WLLD05]. As cloud systems grow larger and are partitioned into more unique configurations, this fixed price method becomes inefficient when total demand is much lower than data center capacity. This leads to under-use of the system so cloud providers need an incentive mechanism to encourage users to submit more jobs. When total demand rises over data center capacity, it is desirable to provide an incentive to users to reduce their demand through raising per–unit costs, decreasing performance, or decreasing system availability.

In 2009 Amazon created a spot price system to auction (i.e. sell) to a highest bidder its unused data center capacity. The spot price mechanism for EC2 shares many similarities with the standard uniform price auction mechanism. The spot price charged for a request may fluctuate depending on the supply of, and demand for, spot instance capacity. Spot prices are a tuple of {*maximum price per hour the user wishes to pay for an instance type, the region desired, and the number of spot instances to run*}. If the maximum price bid exceeds the current spot price, the job(s) will run until termination by the user or the spot price increases above the user set maximum price. The cost of spot instance hours are billed based on the spot price at the start of each hour an instance executes. If the user spot instance is interrupted in the middle an of hour of instance use (because the spot price exceeded the user maximum bid price), the user is not billed for that partial hour of spot instance use. However, if the user terminates the spot instance a charge occurs for the partial hour of use [Ama14].

Market driven resource allocation has been applied to grid computing environments [CBA+05, WLLD05] and has recently been adopted by cloud computing. The auction-based resource allocation mechanism in the cloud spot market causes the price of services to be dynamic. Spot market pricing has been done in the electric energy industry [DC01] and is essential for power systems planning and operation. An electricity costing model does not have a mechanism to store electricity as it can not store its service while a cloud system can, thus the floor of the electricity model can be much lower than that of a cloud system as electricity can not be stored in sufficient quantities to keep its floor higher. The alternative is to restrict generation and loose the currently produced power. In a cloud model the system can be made idle, almost instantly, and await a price point when it would be profitable to operate. For both the cloud market and the electricity market, accurate forecasting is very important for both production and consumption of commodities like compute resources and electricity in order to optimize their buying and selling decisions.

In this paper, we demonstrated neural network calculation methods for predicting spot prices. This prediction method would be useful to users of for bidding on spot price system instances from cloud providers. In the literature there are neural network based techniques to forecast electricity spot prices. In [DC01], neural network techniques based on short-term load forecasting is presented to predict short term spot price in the Australian national electricity market. In [JTB11], Javadi, Thulasiram, and Buyya wrote on the characteristics of Amazon spot instances (SIs). We did a comprehensive analysis of SIs based on one year price history in four data centers of Amazon's EC2 by analyzing different types of SIs Amazon offers in terms of spot price and the inter-price time (time between price changes) and determined the time dynamics for spot prices for hour-in-day and day-of-week sample sets. Moreover, [JTB11] proposed a statistical model that fit well with these two data sets. The statistical models are based on a mixture of Gaussian distribution, with three or four components, and are able to capture spot price dynamics as well as the inter-price time of each SI and the model exhibits a good degree of accuracy under realistic working conditions.

For our experiments, we used 3842 spot price data points for seven months starting December 2009 and ending June 2010 (215 days). This averages to 17 data points per day and our analysis showed that was best to fulfill a short-term prediction as the trend of data could change unpredictably fast, therefore making long-term predicting models miss the change. In our model, the data gathering interval averages 1.3 hours. This provides sufficient time to re-train our prediction model and account for the latest input data received from the previous time period.

We choose the multi-layer perceptron (MLP) with the "moving simulation mode" as it is a good short-term prediction method that has re-training and it allows capturing the last significant data available from the previous step of the prediction system that is continually updating and improving its performance [RL97]. The successful usage of the moving simulation mode in [TBDSG11] showed that it is not necessary to choose a large window since a larger window would include older data that makes the NN re-training less effective. Similar to work in [TBDSG11], we chose twenty values as the size of the moving simulation window. The MLP consists of three or more layers (input, output, one or more "hidden" layers). The nodes are non-linearly activating. Each node in one layer connects with a certain weight to every node in the following layer. Learning occurs by the perceptron changing connection weights after each datum is processed based on the variance output compared to expected results.

The MLP architecture of 5-10-1 was chosen along with the Box-Jenkins method [BJ70] forming the MLP training set. We chose five input neurons because it is sufficient within the twenty input data points of the moving simulation window with ten neurons of the hidden layer being enough to provide a good generalization and predicting ability of the model for MLP training on twenty input vectors and one output neuron as we are predicting the price for one step ahead. The neurons of the hidden and output layer use a sigmoid activation function. On each step of the moving simulation mode, the MLP is trained to reach the sum-squared training error 10^{-5} with 10^{6} minimum number of training epochs.

In the data set from Amazon EC2, the real and predicted spot prices for historical data about spot prices of the "medium" class of cloud instances based on Linux and Windows instance configurations were used as data input to the MLP system. The MLP model used provided a very good representation of the real trend. The numerical analysis of the predictions in Table 1.1 shows the high accuracy of the proposed approach as the monthly average relative prediction errors do not exceed 5.6% for the *m1.linux* data and 6.4% for the *m1.windows* data. The average relative prediction errors for the whole testing period of six months are 3.3% and 3.7% respectively for *m1.linux* and *m1.windows* data. For our purposes a prediction result is an outlier when its relative prediction error is more than 10%. For out prediction results we had 155 (about 4.0% of the total results) and 188 (about 4.9% of the total results) outliers for the *m1.linux* and the *m1.windows* experiments respectively.

We consider our prediction results as being very good in the context of a one-step prediction system as peaks of the spot price are predicted. This is due to using a moving simulation mode which re-trains the MLP on each prediction step.

Experiments	Avg. relative prediction error(%)		Num.& Percent of outliers (Rel. Predict. Err. >10%)	
	m1.linux	m1.windows	m1.linux	m1.windows
Dec.2009(266)	4.4	3.5	12 (4.5%)	16 (6.0%)
Jan.2010(556)	2.6	3.4	5 (0.9%)	25 (4.5%)
Feb.2010(556)	4.0	6.4	25 (4.5%)	28 (5.0%)
Mar.2010(663)	2.6	2.6	2 (0.3%)	10 (1.5%)
Apr.2010(564)	1.7	2.3	5 (0.9%)	7 (1.2%)
May 2010(637)	2.0	3.6	10 (1.6%)	49 (7.7%)
Jun.2010(595)	5.6	3.9	96 (16.1%)	53 (8.9%)

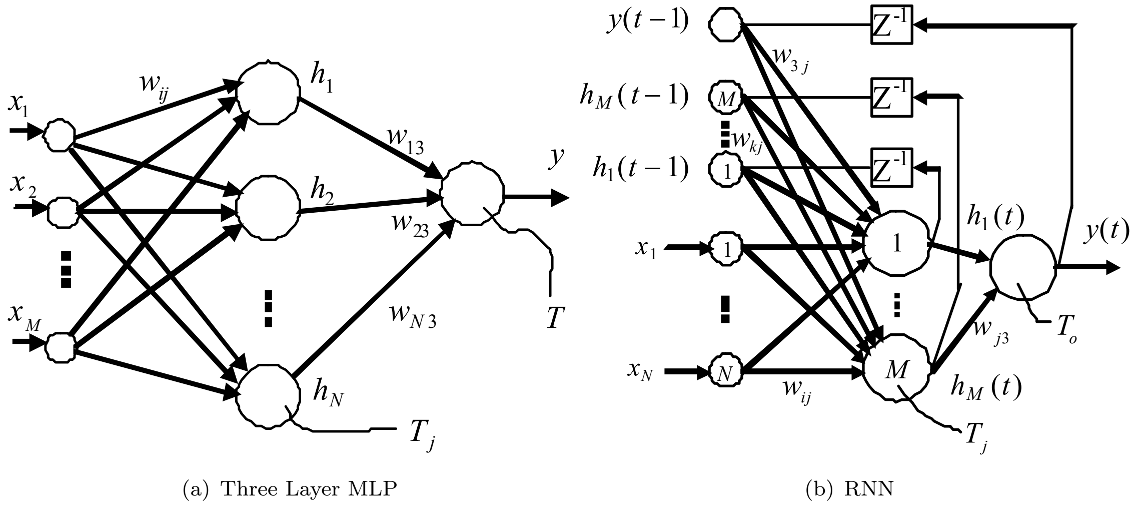
Table 1.1: Numerical analysis of MLP prediction results

The conclusion of our paper was that the experimental simulation modeling results for the Amazon EC2 spot instances showed high correlation accuracy of the proposed approach given that the average relative prediction error does not exceed 4% and the number of outliers were less than 5% for the total number of the prediction results showing that neural networks are well suited for prediction and are useful for users bidding on spot instance services.

1.4.2 Spot Price Prediction for Cloud Computing Using Neural Networks

In the paper by V. Turchenko, V. Shultz, I. Turchenko, R. Wallace, et.al. [TST+14], this work expanded and provided more depth on the neural network model from [WTS+13]. Using the same data as in [WTS+13] the expansion on prediction of the spot prices used two standard models of neural networks: a multilayer perceptron (MLP), Figure 1.1(a), and a recurrent neural network (RNN), Figure 1.1(b); the Haykin model for MLP and the Boyacioglu and Baykan model for RNN were used. These are well known and well researched models providing the required level of accuracy as described in [TST+14].

New in this paper, as compared to [WTS+13], is a "Middle Term" prediction mode. Taking into account the long simulation time of the computational experiment, this paper provides a middle-term prediction using 88 and 176 data points from the December 2009 to June 2010 data set as training data using two NN models (an MLP 5-10-1 and a RNN 5-10-1 NN layering) with reverse connections from both hidden and output layers as shown in Figures 1.1(a) and 1.1(b). Both models use adaptive and constant learning rates. The constant learning rates were 0.5/0.5 for the hidden and output layers for the MLP model and 0.1/0.1 for the RNN model. Both models are trained to reach the sum-squared training error of 10^{-5} with 5×10^5 training epochs. The training time of one middle term prediction experiment took about 30 seconds using the MLP model and 45 seconds using the RNN model for 88 input data points and about 60 seconds using the



(a) Three Layer MLP

(b) RNN

FIGURE 1.1: MLP and RNN state transition models

MLP model and 90 seconds using the RNN model for 176 input data points. All middle term prediction experiments were executed on an AMD Phenom II x 4 956 processor at 3.4GHz with 4GB of RAM. The total computational time for the entire experiment for middle term prediction was about 4 hours.

As the paper shows, the MLP and RNN models provide accurate prediction results for the majority of cases. For both of the 88 and 176 input training data sets the prediction results are a bit less accurate for the December 2009 and the June 2010 time periods for the fifth prediction day. The results did show good prediction for neural networks using the middle term prediction of cloud system spot prices.

The paper concludes that predictive models based on artificial neural networks for short term and middle term prediction methods of future spot prices for cloud computing have low error margins and that the models, based on standard multi-layer perceptron and recurrent neural network architectures, performed well. For prediction actions, the moving simulation mode approach to remove old historical data for neural network re-training improved prediction accuracy of the model. The experimental results on the Amazon EC2 spot instances showed high prediction accuracy with the approach used. For the short term prediction mode, the average relative prediction error is less than 4% and the number of outliers (a relative prediction error of more than 10%) is less than 5% for the total number of prediction results. For the middle term prediction mode, the average relative prediction error is in the range of 2.2—4.6% and the maximum relative prediction error is in the range of 5.1—17.8%. The obtained experimental results show that neural networks are well suited for this kind of prediction and are very useful for users bidding on spot instance services. Prediction of spot prices from other cloud service providers using neural networks will potentially be a future direction of such research.

1.4.3 Regulated Condition-Event Matrices for Cloud Environments

In the paper by Wallace, Martin, and Vázquez-Poletti, et.al.[WMV14] the authors introduce the Regulated Isomorphic Temporal Architecture (RITA) which provides time-coordinated methods of control not dependent on a single time domain and an explicit separation of temporally based event processing from computations. RITA separates temporal event processing from computation providing a functional programming [7] style for developers in a familiar language that can integrate with existing procedural code without working in multiple coding paradigms requiring extensive "glue code" [8] to allow one paradigm to work with another. This paper introduced a guarded condition-event system that provides a regulated, isomorphic temporal architecture that has an explicit separation of event processing and computation with constructs allowing integration of time-aware events for multiple time domains found in Cloud or existing distributed computing systems.

RITA is a run time service in the PaaS layer. The advantages of RITA in the PaaS layer are:

- Preventing application unbounded priority inversion by reducing the communication interrupts that lead to this problem.
- As an event propagator it is designed for partitioned, communicating processes in a "share all," "shared partial," or a "shared nothing" environment supporting parallelism without heavy use of synchronization primitives for publication/subscription (pub/sub) systems or message passing interfaces.
- High performance worker threads for "scatter gather" configurations can use the RITA bifurcation of communication and computation to improve performance with a high performance PaaS IPC.
- Canonical event forms define the data communication needed for a distributed application making all communication explicit and drastically reducing communication traffic loads and errors.

As cloud computing systems have processes executing autonomously and independently communicating with each other their communication can quickly increase to a level that decreases system throughput. Communication currently in use depends on uniform, monolithic communication mechanisms following a pub/sub methodology for tight-cluster, grid, or single systems. These systems are dependent on a single homogeneous time domain that requires an inordinate amount of effort handling latency issues

[7] A programming paradigm that treats computation as the evaluation of mathematical functions and avoids state and mutable data

[8] Code that does not contribute any functionality but serves solely to "glue together" different parts of code that would not otherwise be compatible

in a monolithic time domain for federated, distributed cloud systems. This latency can cause side-effects due to imperative code managing events. Events are either missed and must be resent, if possible, or the imperative code induces more latency by direct queue management from the application code.

Declarative coding in RITA allows side-effect free conditions that are separate from the imperative application code. Using RITA it is possible create a series of processes that can be (a) event sensitive by using only the canonical event forms (i.e., spike, set-at, and transitional), (b) temporally sensitive, based on local time domain and triggered by canonical events, and (c) value sensitive through delta value comparisons. This results in each RITA cell being semi-autonomous so unexpected behavior can either be ignored or can trigger a recovery in RITA processing cells as required. RITA is designed to work in a distributed and autonomous environment. A discussion of cloud computing and its dependencies on networks [Bir12] observes that the issues of network traffic are becoming limiting factors in cloud computing systems. RITA reduces unnecessary IPC traffic and improves application performance as described in [WMH84, Wal00] and documented in [WMV14]. Other DEBS do not have the clean demarcation between the declarative and imperative portions of an application and the ability to naturally enforce functional programming in a non-functional specific language which is why Lisp, Scheme, and Haskell have never really surpassed C, C++, Java, and other imperative languages in commercial popularity.

1.4.4 Autonomic Resource Contention-Aware Scheduling

In this paper, Sheikhalishahi, Grandinetti, Wallace, et.al. introduce the concept that complexity in computing systems introduces issues and challenges causing poor performance and high energy consumption [SGWVP15]. The authors provide a metric definition and model of resource of contention that can be used for high performance computing workloads. The metric is used in scheduling algorithms for high level of resource management. In the paper an autonomic resource contention-aware scheduling approach is developed for various layers of the resource management stack. The metric definition allows the relationship between distributed resource management layers to have a measure allowing optimization and reducing resource contention.

The authors describe a new job state called “Prepending.” A job in this state is accepted, but not immediately queued, for execution after acceptance. This technique balances, or minimizes, the resource contention that can occur by keeping those jobs that increase the contention for resources, out of execution scheduling for a period of time. When jobs in the “Prepending” state transition to the Pending Execution state,

they are added to a wait queue where resources are allocated for the job by the scheduler. To keep track of the weighting for all jobs in different stages — currently running jobs, scheduled jobs, waiting jobs, and "Prepending" jobs — three dictionary-based data structures were designed to hold the data: 1)Monitoring Data, 2)Schedule Monitoring Data, and 3)Monitored Jobs. The autonomic algorithm uses the following resource management components:

- *Queue Mechanism.* This is the entry point of a job into the scheduler. If a job passes through admission control policy it can be accepted and is marked as Pending or "Prepending."
- *Scheduling Function.* In every scheduling cycle, the "Prepending" list is checked for jobs that can be changed to the Pending state, thus they will be added to the wait queue, so the scheduler can allocate resources to that job.
- *Job Scheduling.* When the precise schedule for a job is determined, the Monitored Jobs data is completed specifying which nodes are allocated to the job, and what parts of a job are scheduled for each of the allocated nodes.
- *Job Completion.* Once a job completes, or is canceled, and no longer requires the acquired resources this algorithm updates the monitoring data.

The following assumptions are used for the algorithms and models:

- Let $j.stresson$ be resources which job j is stressed by (i.e. resources in high demand or low supply)
- Let $j.resReq$ be resource requirements of job j in terms of capacity
- Let $JS(t,n)$ be the scheduled jobs on physical host n at time t

At the beginning of every scheduling cycle the scheduling function looks at all "Prepending" jobs in search of jobs which can be transitioned to the Pending state. If the inequality formula 1.3 holds for all stressed (*stresson*) resource types for a job, then the job state is transitioned to Pending which updates *SchedMonitoringData*. The details are presented in Algorithm 2.

$$Util[resType] + JobResReqs[resType]/Total[resType] <= 1 \tag{1.1}$$

$$\begin{aligned} MonitoringData[resType] + JobResReqs[resType] \\ <= autoCoef * Total[resType] \end{aligned} \tag{1.2}$$

$$SchedMonitoringData[resType] + JobResReqs[resType] \\ <= autoCoef * Total[resType] \tag{1.3}$$

In Algorithm 1, the inequality formulas, 1.1 and 1.2, are used to determine the initial actions for a job. Formula 1.1 and then 1.2 are executed in order. *Total* indicates the total capacity of each resource for a site, and *autoCoef* is introduced as an autonomic parameter. According to the use of each resource under consideration, *autoCoef* may take one of the three values as 1, 2, and 4 for high, moderate, and low loads, respectively. These inequalities and functions are developed intuitively to determine the initial state of a job.

Algorithm 1: Queue(Job j)

```
1  begin
2      MonitoredJobs[j.id] ⇐ empty dictionary
3      flagCounter ⇐ 0
4      forall the r such that r ∈ j.stresson do
5          MonitoringData[r] ⇐ MonitoringData[r] + j.resReq[r]
6          MonitoredJobs[j.id][r] ⇐ j.resReq[r]
7          Util[r] ⇐ getUtilization(r, currentTime)
8          Total[r] ⇐ getTotal(r)
9          if Util[r] ≤ 1 − (j.resReq[r]/Total[r]) then
10             flagCounter ⇐ flagCounter + 1
11         if flagCounter = len(j.stresson) then
12             j.state ⇐ Pending
13             forall the r such that r ∈ j.stresson do
14                 SchedMonitoringData[r] ⇐
                   SchedMonitoringData[r] + j.resReq[r]
15         else
16             flag ⇐ False
17             forall the r such that r ∈ j.stresson do
18                 if MonitoringData[r] + j.resReq[r] > autoCoef * Total[r] then
19                     flag ⇐ True
20             if flag = True then
21                 j.state ⇐ Prepending
22
23             else
24                 j.state ⇐ Pending
25                 forall the r such that r ∈ j.stresson do
26                     SchedMonitoringData[r] ⇐
                       SchedMonitoringData[r] + j.resReq[r]
```

Algorithm 2: ScheduleCycle1(Time t)

```
1  begin
2      forall the j such that j.state ∈ Prepending do
3          flag ⇐ True
4          forall the r such that r ∈ j.stresson do
5              if SchedMonitoringData[r] + JobResReqs[r] > autoCoef * Total[r]
               then
6                  flag ⇐ False
7                  break
8              if flag = True then
9                  j.state ⇐ Pending
10                 forall the r such that r ∈ j.stresson do
11                     SchedMonitoringData[r] ⇐
                       SchedMonitoringData[r] + j.resReq[r]
```

When a job has a precise schedule as determined by the scheduler then *MonitoredJobs* is populated specifying the host, or hosts, where a job is scheduled. This is done by creating a second level dictionary key with the physical host identification in *MonitoredJobs*. The third level dictionary has a flag to model transitioning between full or not full states of a physical host creating third-level dictionary keys for each *stresson* resource assigned and the amount of resource consumption needed by a physical host for the scheduled job. The algorithm is presented in Algorithm 3.

Algorithm 3: ScheduleJob(Job j, Time t)

```
1  begin
2      forall the pnode such that pnode ∈ vmrr.nodes.values() do
3          MonitoredJobs[j.id][pnode] ⇐ {}
            ▷ After job j gets scheduled, its resource requirements are
           held in the vmrr structure.
4          MonitoredJobs[j.id][pnode]['flag'] ⇐ True
5          forall the r such that r ∈ j.stresson do
6              MonitoredJobs[j.id][pnode][r] ⇐ vmrr.resReq[pnode][r]
```

1.4.5 A Multi-capacity Queuing Mechanism in Multi-dimensional Resource Scheduling

In an expansion on the paper in §1.4.4, Sheikhalishahi, Wallace, Grandinetti, et.al. wrote in [SWG+14] that the complexity of computing systems introduce issues and challenges causing poor performance and high energy consumption. In this paper we defined and modeled resource contention metrics for high performance computing workloads as a performance metric in scheduling algorithms at the highest level of resource management to address the main issues in computing systems. We then proposed a novel autonomic resource contention-aware scheduling approach for various layers of resource

management establishing the relationship between distributed resource management layers and optimizing the resource contention metric. Our simulation results confirmed our approach.

From the scheduling viewpoint, the core issues of poor performance are high idle time (i.e. low processor usage), overloading, and resource contention. Energy efficient computing was also considered as a factor to reduce high energy consumption. These core issues are related to each other and may be more important for future high performance computing (HPC) systems. Resource contention is widely recognized as having a major impact on the performance of computing systems and distributed algorithms. Applications running simultaneously on adjacent cores of a multicore processor may experience reduced performance due to an increased miss rate in the lowest level caches. Regardless, performance metrics commonly used to evaluate scheduling algorithms in resource management systems do not take into account resource contention because researchers are more interested in improving the conventional, well-known performance metrics of utilization, application makespan scheduling, and latency. On the contrary, we consider addressing the resource contention issue will implicitly address poor performance, high energy consumption issues, application makespan scheduling, and latency [FSS07, ZBF10].

In scheduling, we expect users to specify a stress factor for their jobs. This user provided information and user behavior are used in the design of contemporary parallel systems schedulers [SF09]. If users do not specify this attribute, they can at least specify the HPC class — such as compute intensive, data intensive, and so on, for their workload. This information can be used to induce stress factors.

Our approach in this paper was inspired by autonomic computing. An autonomic scheduling approach could exploit this reference architecture to make various decisions in different components. Queue mechanisms based on job characteristics information is provided by an information service, other jobs in the queue, and a grouping – or affinity – mechanism that could be implemented to reduce resource contention among jobs. In our autonomic scheduling model, we take into account the interaction of low-level components of resource management, such as the core scheduler information, with the higher level components of resource management and the front-end components such as admission control, pricing strategy, and queuing mechanisms.

In our approach the scheduler makes decisions in the queuing mechanism of whether to put a new incoming job in the wait queue by setting its state to a Pending state immediately, or wait through additional scheduling cycles until the necessary conditions are establish by setting job's state to our new state *Prepending* — as opposed to *Pending*

— according to the system state from the core scheduler information about resources, jobs, and applications.

The key information to be exploited at higher levels of autonomic scheduling are the status of resources (fully used or not), the resources usage, and how jobs are scheduled. In this work, we introduce a new job state of *Prepending*. This is the state of jobs which are accepted as jobs for the system, but are not immediately queued after their acceptance. This technique balances resource contention by minimizing the impact of these jobs by having the jobs wait and not be immediate scheduled for a job scheduling period thus reducing the contention for resources. When *Prepending* jobs transition to a *Pending* state, after evaluation of their resource needs, they are added to the wait queue where the scheduler allocates resources to them and the job is now *Pending*. To keep track of the weight of all jobs in different stages: currently running jobs, scheduled jobs, waiting jobs, and *Prepending* jobs, we design three dictionary-based data structures as follows:

- *MonitoringData*. This data structure has a key for each resource type, and the corresponding key value reports the current consumption of the resource type. It only reports about consumption of stress resource types, because stress consumption contributes to resource contention metric. *MonitoringData* contains information for all available jobs in the system regardless of their state, that is, currently running jobs, scheduled jobs, waiting jobs, and *Prepending* jobs.
- *SchedMonitoringData*. This data structure is the same as *MonitoringData* by tracking resource usage of all other jobs but *Prepending* jobs.
- *MonitoredJobs*. This represents the total capacity needs for a job. After a job becomes scheduled, this data structure then represents how the capacity needs of a job are satisfied by what portion of job capacity needs are satisfied by contributing physical hosts in the distributed system.

TABLE 1.2: HPC characteristics distribution in workload trace

HPC Char.	Overall Weight	CPU Memory	CPU	Memory	I/O	Net-in Net-out	Net-in	Net-out
Compute intensive	$\frac{1}{2}$	$\frac{6}{10}$	$\frac{4}{10}$	0	$\frac{1}{2}$	$\frac{4}{6}$	$\frac{1}{6}$	$\frac{1}{6}$
Data intensive	$\frac{1}{6}$	$\frac{3}{10}$	0	$\frac{7}{10}$	$\frac{1}{2}$	$\frac{4}{6}$	$\frac{1}{6}$	$\frac{1}{6}$
Memory intensive	$\frac{1}{6}$	0	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{4}{6}$	$\frac{1}{6}$	$\frac{1}{6}$
Comm. intensive	$\frac{1}{6}$	0	0	0	$\frac{1}{2}$	1	0	0

We used workload archives from the Parallel Workloads Archive [Fei10] as job traces for our simulation experiments. In general, there is no workload archive in the Parallel Workloads Archive representative of HPC job characteristics, which resources they put

stress on, and other information we need in our model and algorithm for capacities of net-in, net-out, and IO resource types of jobs. We synthetically generated these parameters through uniform distributions. At first, a uniform distribution specifies HPC characteristic of a job, then according to HPC characteristic, we have at most three uniform distributions to select resource types of a group, that is, one for CPU memory, CPU, and memory group of resources, another one for net-in net-out, net-in, and net-out, and one for IO resource. The details are presented in 1.2. We conducted a number of experiments according to the following configuration parameters covering settings of autonomic algorithm, policies, and the number of cores per physical host:

- Autonomic algorithm. If the autonomic algorithm is enabled, there is an AUTO term in the initial part of configuration name.
- Workload traces. We used workload traces derived from SDSC Blue Horizon, SDSC DataStar, and KTH IBM SP2 of the Parallel Workloads Archive. We altered these derived traces.
- Systems multi-instance type CPUs of 2, 4, 8, and 16 cores per physical node.
- Host selection policies. We explored two green host selection policies as described in [SLG11]. In our paper, we use these two policies, GREEN1 and GREEN2, in our configurations to evaluate their behavior with autonomic scheduling approach.
- Host selection policies. We explored two green host selection policies in [SLG11]. In this paper, we use these two policies, that is, GREEN1 and GREEN2 in configurations to see their behavior with autonomic scheduling approach.

GREEN1 is the simplest policy. It calculates resource contention among the scheduled jobs on a physical host at time t and then, measures the consolidation score of a physical host for a job being scheduled. Tentative time t is determined by the scheduler as a possible time to schedule a job. In this policy, only the schedules and reservations on a physical host at time t will participate in consolidation score, and it simply ignores the job time horizon where in the future, there may be changes in reservations and schedules.

GREEN2 policy is multidimensional for the run time horizon of a job where it considers the future reservations and changes of a physical host to calculate resource contention. This policy seems to be more precise as it divides future time into time steps in which each time step keeps the status of a physical host unchanged. At the beginning of each time step the consolidation score of a physical host regarding a job is determined, and then, it is multiplied by the duration of that time-step (end of time-step minus start of time-step). Finally, the summation of all these values over the job run time is the final consolidation score of the physical host regarding the job being scheduled.

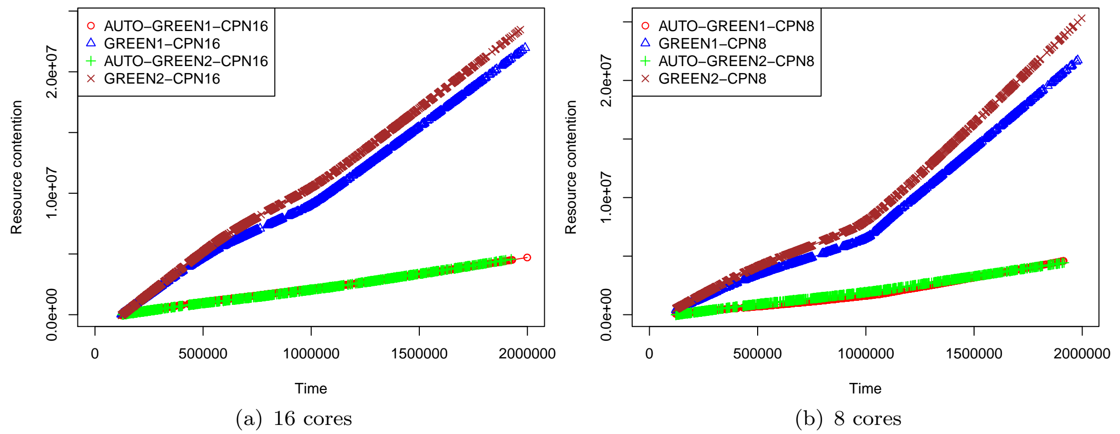
In addition to the variable parameters, we have fixed parameters of aggressive back-filling strategy as the packing mechanism and cloud paradigm as the computing paradigm used in all configurations. Thus, the scheduling function periodically evaluates the wait queue, using an aggressive back-filling algorithm to determine whether any job can be scheduled. We only study the cloud paradigm where the requested run time of jobs is precise and accurate, unlike the HPC paradigm in which it is an estimation. We consider systems at sites with more than one core per physical host to create an environment that can have resource contention. Our naming convention for a configuration has three parts. The first part is "AUTO" if autonomic algorithm is enabled, otherwise it would be omitted. The second part presents host selection policy of a configuration, that is, "GREEN1" or "GREEN2" used in our experiments. The third part shows the number of cores per physical node of a site under experimentation starting with the "CPN" term followed by the number of cores per physical node format. For instance, "AUTO-GREEN1-CPN8" is a configuration with autonomic algorithm enabled, GREEN1 as the host selection policy with eight cores per physical system.

Our experiments explored the impact of autonomic algorithms on the following metrics:

- Completion time. The time from the start of the trace to when the last job request is completed, measured in seconds.
- Resource contention. This value, measured in seconds, is a variance comparison metric and is not an exact time for contention for a resource.

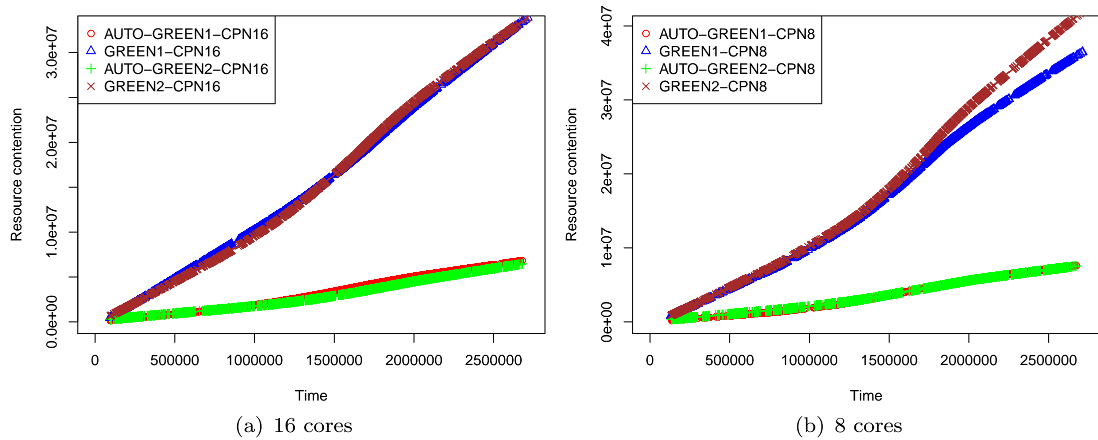
For each workload trace we group resource contention and graph over time based on our experimental number of cores per physical node.

Figures 1.2(a), 1.2(b), Figures 1.3(a), 1.3(b), and Figures 1.4(a), 1.4(b) demonstrate the graphs for BLUE1, BLUE2, and DS traces, respectively. We observed that for all cases autonomic scheduling approach outperforms non-autonomic with large improvements. In total, the results demonstrate that establishing the relationship at the resource management layers as shown by the resource contention metric leads to less resource contention resulting in improved performance. Autonomic scheduling resulted in an average of 4.8 times better improvement.



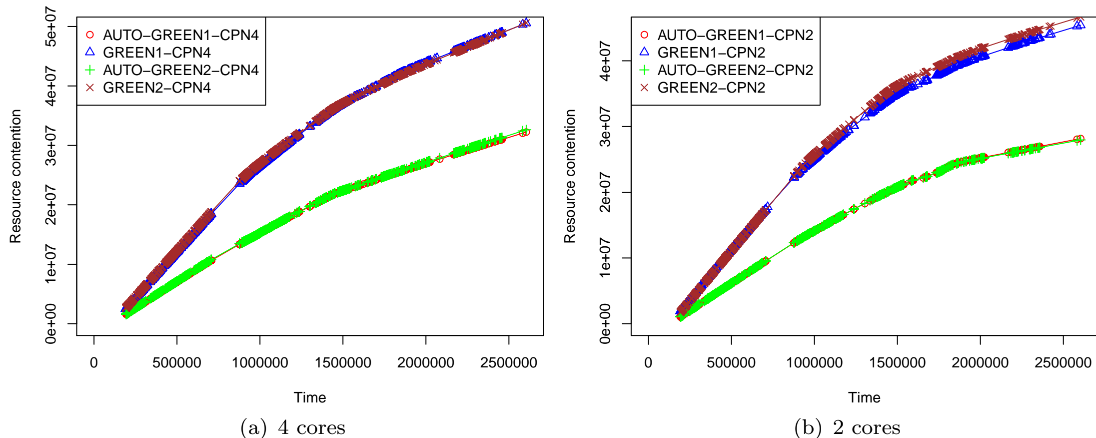
(a) 16 cores (b) 8 cores

FIGURE 1.2: BLUE1: Resource contention graph for 16 and 8 number of core processors. Lower resource contention is better.



(a) 16 cores (b) 8 cores

FIGURE 1.3: BLUE2: Resource contention graph for 16 and 8 number of core processors. Lower resource contention is better.



(a) 4 cores (b) 2 cores

FIGURE 1.4: DS: Resource contention graph for 4 and 2 number of core processors. Lower resource contention is better.

1.4.6 A Multi-Dimensional Job Scheduling

Sheikhalishahi, Wallace, and Grandinetti, et.al. were invited to expand the paper in §1.4.5 for inclusion in a special edition of the Future Generation Computer Systems journal [SWG+15]. In this paper a statistical analysis was done for the work-load traces from the prior papers on this topic. For each experiment, for each job, time data was collected. t_a is the arrival time, or time when the job request is submitted; t_s si the start time of the job; and t_e is the time the job ends. At the end of an experiment, the following metrics were computed:

- *Waittime*: This time is computed as $t_s - t_a$; the time a job request must wait before it starts running with units in minutes.
- *Slowdown*: If t_u is the time the job would take to run on a dedicated physical system, the job's slowdown is $(t_e - t_a)/t_u$. If t_u is less than 10sec, the *Slowdown* is computed using t_u set to 10sec.

The optimization of these two metrics is a minimization problem. The authors analyzed simulation results for each experiment based on mean and standard deviation measures. In order to compare the two policies, Multi-Capacity Bin Packing (MCBP) and back-filling queuing policy (BKFL), MCBP is normalized to BKFL results as: $MCBP/BKFL$. This transforms a policy into a value allowing better numeric presentation and objective comparison.

In general, better results, a smaller σ, were found for both MCBP and BKFL, however in terms of standard deviation the *Waittime* metric was higher for the MCBP policy, while *Slowdown* produces a smaller deviation but on average the results were better

for *Waittime* and *Slowdown*. There was more discrepancy on *Waittime* values for the MCBP policy with respect to the BKFL policy but the data were clustered better for the *Slowdown* metric using the MCBP policy. This observation implies that, in total, there is better scheduling with the MCBP policy than with the BKFL policy. Using the MCBP policy jobs are allocated to the system faster showing that the MCBP policy outperformed the BKFL policy.

1.4.7 A Performance/Cost Model for a CUDA Drug Discovery Application on Physical and Public Cloud Infrastructures

Guerero, Wallace, Vázquez-Poletti, et.al. wrote on the topic of moving a CUDA[9] application from a local system to a Cloud Computing environment[GWVP+13] detailing a price-performance model. We experimented with a virtual screening (VS) application that aids drug discovery research by predicting how ligands interact with drug targets. The *BINDSURF*[SLPSCG11] application is a fast and efficient blind-VS methodology for the determining protein binding sites depending on the ligand. It is used for fast unbiased pre-screening of large ligand databases by using the parallel architecture of GPUs to compress execution time. With our model it is possible to determine the best infrastructure use for execution time and system costs for any given problem solved by BINDSURF. Conclusions obtained from our study can be extrapolated to other GPU based virtual screening methodologies.

Large system clusters are adopting these relatively inexpensive and powerful GPU devices as a way of accelerating computationally-intensive parts of the applications. One of the current supercomputers, Titan, located at the DOE Oak Ridge National Laboratory in Tennessee, USA[SDS14], is equipped with AMD Opteron Processors and the latest generation of nVidia K20x GPUs. GPUs have a great impact on the power consumption of the system. A high-end GPU may well increase the power consumption of a cluster node up to 30%. This is a critical concern especially for very large data centers, where the cost dedicated to supply power to such computers represents an important fraction of the total cost of ownership (TCO)[FWB07].

Reducing power consumption in these large installations is now becoming an urgent concern as several governments (e.g., US, British) are creating taxes targeting facilities that consume too much electricity. For instance, some of the more well known data centers on the Internet, such as Google and Facebook among others, consumed about 0.5% of the overall electricity in the world during 2005. When electricity needed for cooling and power distribution is also considered that number increases up to 1% [Koo08]. The

[9] Compute Unified Device Architecture is a GPU architecture developed by nVidia corporation.

research community is also aware of this issue and it is making efforts in developing reduced-power installations. For instance, the Green500 list [FC14] shows the 500 most power efficient computers in the world. In this way we can see a clear shift from the traditional metric FLOPS to FLOPS-per-watt.

Virtualization techniques provide significant energy savings through enabling greater resource use by sharing hardware resources among several users. This reduces the amount of a particular device needed. Virtualization is being increasingly adopted in data centers because of this shared use and reduced infrastructure cost. In particular cloud computing is an inherently energy-efficient virtualization technique where services run remotely in a ubiquitous computing cloud providing scalable and virtualized resources[Hew08]. Thus peak loads can be moved to other parts of the cloud and the aggregation of resources provide higher hardware use[BGDG+10]. Public cloud providers offer their services in a "pay-as-you-go" fashion, and provide an alternative to local system infrastructures. This alternative to local system infrastructures only becomes real for a large data amounts and long execution times.

In our experiments we used *BINDSURF* VS calculations for direct prediction of binding orientations. We used three different ligands that represented chemical diversity of large compound databases. We refer to them as ligands *A*, *B* and *C*. Ligand *A* is a blood clotting co-factor recently discovered by Leo [LSZ09]. Ligand *B* and ligand *C* have been extracted from their Protein Data Bank complexes with the respective identifiers **2byr** and **3p4w**. In the binding orientation docking calculations we used 5, 10, 50, 500, 5000 and 50000 Monte Carlo steps. An optimal value for the steps parameter does not exist for all different ligand types we used — *A*, *B* and *C* — so therefore we performed a small number of VS calculations for short Monte Carlo steps (5, 10, and 50) to obtain qualitative information about potential hot-spots in the surface screening approach for millions of different ligands. In other situations we might be more interested in obtaining accurate predictions for a smaller set of ligands and use higher number of Monte Carlo steps (500, 5000, and 50000).

The local architecture we used to perform the experiments is described in Table 1.3(a). It is an Intel Xeon E5620 CPU with 4 cores running at 2.4GHz, 16GB of memory and two nVidia Tesla C2050 graphics cards.

The cloud infrastructure we used is one offered by Amazon through its Elastic Compute Cloud services[10]. As BINDUSRF is coded using CUDA, the Cluster GPU instances were the only possible choice. The specifications of the GPU provided by Amazon EC2 are displayed on Table 1.3(b).

[10] http://aws.amazon.com/ec2/

Table 1.3: Platforms System Specifications.

(a) Local Machine

Proc.:	Intel Xeon E5620@2.4Ghz
Memory:	16GB
2xGPU nVidia Tesla C2050	
GPU:	GF100
Memory Size:	3072 MB
Memory Bandwidth:	144 GB/sec
Stream Processors:	448
Max Power Draw:	238 W

(b) Amazon EC2

Proc.:	2xIntel Xeon X5570@2.93GHz
Memory:	22GB
2xGPU nVidia Tesla M2050	
GPU:	GF100
Memory Size:	3072 MB
Memory Bandwidth:	148.4 GB/sec
Stream Processors:	448
Max Power Draw:	225 W

As a public cloud provider, Amazon charges per hour of use. Each "Quadruple Extra Large" instance (which provided GPUs) deployed on the US-East Region costs $2.1 per hour[11]. We compare both local and cloud models for BINDSURF processing 6,000 different ligands. Each BINDSURF simulation has 5,000 Monte Carlo steps. This is the maximum number of steps we have empirically evaluated. Several assumptions are taken in order to compare those models. They are:

- A machine from the local infrastructure costs $8,159.55
- The amortization period of each of these machines is 3 years
- The kW-h price is that of Spain[12]: $0.1352
- The energy consumption in idle mode for a machine from the local infrastructure is 245 W-h
- The facility cost per machine per year in the local infrastructure is $12,000.
- The administrator salary is $3,300/Mo. and each administrator is assigned to 100 machines from the local infrastructure.
- The cluster GPU instances from Amazon were launched from the US East region data center with a cost of $2.10/Hr.

The cloud model is compared to different percentages of local infrastructure use ranging from 40% to 100%. In the local infrastructure, the costs become stabilized from 100 machines upward. The system administrator salary represents a rate for administering 100 machines. From this point forward the cost is linear for the local infrastructure. Although the number of machines used in the experiments and the administrators needed to maintain those machines are increased, the execution time of the targeted application decreases. Considering an average-high usage of the local infrastructure (nominally 60%-70%), the cloud infrastructure is a good solution for ligand type A. With ligand C the result occurs but only in certain cases. The processing of ligand type B should be moved to the cloud only when an average local usage is 40% and only in very specific cases.

[11] http://aws.amazon.com/ec2/pricing/
[12] http://www.statista.com/statistics/13020/electricity-prices-in-selected-countries/

We found it noteworthy that Amazon charges per hour of use and rounds up to the next whole hour for partial hour use. Thus, if the execution time of an application is 1.1 hours, Amazon will charge for 2 hours. As more machines are added to the resource pool, with the execution time equally distributed among them, it is more likely to have idle machine hours. This fact is reflected in different behaviors of *BINDSURF* when executing different types of ligands. In our case, ligand type *B* is the most affected by the rounding method.

Focusing on the physical infrastructure we provided a detailed cost model that considered a wide variety of elements and factors such as energy consumption, administration cost, and machine facility costs. This work concludes with establishing a "break-even point" for use of *BINDSURF*. We provided detailed comparisons of execution of the same application on the two infrastructures generating a performance/cost model for each. We concluded that the machine usage per year of the local infrastructure should be quite high — ranging between 50% to 100% — in order to be profitable, otherwise cloud computing is a more cost effective alternative than local computing. Cost calculations are different between local and cloud infrastructures as the variability in charging caused by the partial-hour upward rounding (the ceiling cost per hour) is not reflected in the local system price, and thus, cloud infrastructures are highly affected by execution time due to this rounding calculation.

1.4.8 Consideration of the TMS320C6678 Multi-Core DSP for Power Efficient High Performance Computing

As part of a team at Oak Ridge National Laboratory investigating non-traditional computing elements for high performance computing to be considered for the advanced computing initiative Wallace, Vacaliuc, Clayton, et.al. in [WVC+11] investigated heterogeneous computational architectures. Current computational architectures using accelerators [BDH+10b] require extensive system knowledge and esoteric programming methods. These systems are composed of commodity processors integrated with Field Programmable Gate Arrays (FPGA) and Graphics Programming Units (GPU) or both types. A new Digital Signal Processor (DSP) architecture may be a viable alternative to the FPGA/GPU and avoid the performance problems associated with integrating accelerators into computer systems. We investigate the Texas Instruments TMS320C6678 multicore DSP which demonstrates equivalent power, cost efficiency to the best accelerators available today, and has an identical programming paradigm as multicore general purpose CPUs.

Heterogeneous multiprocessor systems represent the leading edge of HPC systems. These systems take advantage of different types of computing hardware by assigning computation tasks to the most appropriate hardware type. A common example is the hybrid CPU/FPGA combination. Recent breakthroughs in the use of GPUs as computation nodes have expanded the range of processors that could be used in a target hardware system. With both the FPGA and GPU as accelerators, very high computational efficiencies have been observed [BDH+10b]. Not all codes are suited for FPGAs or GPUs. Programming FPGA-based processors is difficult, and quite foreign, for application programmers because the available tools are designed for hardware logic synthesis [Hem09]. Although GPU programming is based on C/C++ languages such as CUDA or OpenCL, achieving good performance is challenging because data structure and access patterns must be rearranged. These obstacles render both FPGAs and GPUs unsuitable for many applications [GBL10]. With the TMS320C6678 multicore DSP, it is now possible to have high computational efficiency in a processor that can participate in a large percentage of the source code [CHB+09]. This is achieved without dependencies on out-of-line code such as in FPGA and GPU systems.

A survey of state-of-the-art processors in Table 1.4 highlights important attributes of the most power-efficient processors currently available. The table is sorted by watts per giga-FLOP (W/GF) from lowest to highest. Other values are operational clock frequency, *Freq(MHz)*; giga-FLOP for double-precision and single-precision, *GFLOPS(DP/SP)*; Thermal design power, *TDP* in watts, is the maximum amount of power the chip package can dissipate; double- and single-precision efficiency versus a nVidia Fermi GPU, *DP Eff.* and *SP Eff.* respectively.

Given the lower initial cost, lower operational costs — DSP devices have a lower price point than GPUs — lower power requirements, and the ability to take existing C/C++ code bases that work with OpenMP communication software we were able to have CRAY computer consider designing a system board. We were able to demonstrate the power and ease of programming of the TI TMS320C6678 multicore DSP. Further development of the CRAY system board was not funded by Oak Ridge National Laboratory as this was a competitive design to the TITAN GPU-based system board design.

Table 1.4: Potential Ultrascale Processors

Device	Cores	Freq (MHz)	GFLOPS (DP/SP)	TDP	Cost ($US)	W/GFLOP (DP/SP)	DP Eff. vs Fermi	SP Eff. vs Fermi
AMD Cypress HD5870 (40nm)	1600	850	680/2720	188	370	0.28/0.07	166%	333%
TI TMS320C6678-1250 (40nm)	8	1250	40/160	17	200	**0.43/0.11**	108%	216%
nVidia Fermi M2050 (40nm)	448	1150	515/1030	238	2500	0.46/0.23	100%	100%
Intel i7-2715QE (32nm)	4	2100	67/134	45	1000	0.67/0.33	69%	69%
IBM Power7 (45nm)	8	4000	256/256	200	Unk.	0.78/0.78	59%	29%
AMD Opteron 6164 HE (45nm)	125	1700	82/163	65	872	0.80/0.40	58%	58%
Fujitsu SparcVIIIfx (45nm)	8	2000	64/128	58	Unk.	0.91/0.45	51%	51%

1.5 Structure and Flow of Thesis Chapters

In this overview I have described the motivation, summary of contributions, background and summarized my prior publications that focused on cloud computing. The remainder of this thesis will discuss the theoretical background in §2 for application execution optimization referring to topics already discussed in §1.4.1 to §1.4.8 in this overview. In §3 the novel work of mapping algorithms to computational elements will be presented demonstrating the model of computation and its allocation across federated, distributed systems comprised of homogeneous and heterogeneous compute elements. In §4 simulation and analysis of the mapping algorithms from §3 are described and the results evaluated against the goal determining the optimal mapping of execution units across cloud-deployed computational platforms. The thesis concludes in §5 with a summary of this work, conclusions, and a discussion of future research in this area.

Chapter 2

Theoretical Background

2.1 Overview

ALGORITHM decomposition and the creation of specific code portions that execute on either primary or ancillary computing devices has been around since the introduction of the popular 8087 co-processor, developed by Intel in 1980, which used programmed I/O or DMA[1] to access the functionality of the co-processor. And, indeed, prior to 1980, large system developers, such as Digital Equipment Corporation, developed the DECSystem-10 [BKHH78] which used algorithm decomposition to create programs for true physically parallel execution. Having used such a system early in my career as a computer scientist, my opinion is that systems have been miniaturized, software tools have become much better, IPC is now a commodity product, and is usually a normal layered product; but algorithm decomposition and handling events has remained a current area of research.

To understand algorithm decomposition it is important to understand the impact that complex event and distributed event-based systems have on the decomposition process. In §2.2 and §2.3 a brief survey of these event systems and languages are discussed. With this understanding it is then possible to explore the RITA's formal model, temporal logic and how that is expressed in the RITA language notation presented in §2.4.

[1]Intel Component Data Catalog 1980, Intel catalog no. C-864/280/150K/CP, pages 8-21, 8-28

2.2 Complex Event Systems (CES)

COMPLEX Event Processing (CEP) is a concept that has arisen through discrete event simulation, database development and programming languages such as Esper and other Event Query Languages (EQL) that are used to create a CES. Most CES currently available are now part of commercial offerings from Oracle, IBM, Tibco, and other vendors [dCRN13]. In [Bui09] Bui describes and gives examples of STREAM, Borealis, AMiT, ruleCore, SASE+, Esper, Cayuga, Drools, and XChangeEQ EQL languages. Each of these languages are based on query language semantics which is not a core concept of algorithm decomposition and, being such, these languages are not explored. Eckert in [EBB+11] categorizes EQL languages as:

1. languages based on composition operators. Also known as composite event algebras or event pattern languages,
2. data stream query languages which are based on Structured Query Language (SQL),
3. production rules,
4. finite state machines, and
5. logic languages.

The first, second and fifth categories are languages explicitly developed for specifying event queries. The third is way to use the existing technologies of production rules to implement event queries. The fourth approach is a known graphical technology.

Commercial activity was preceded by research projects in the 1990s. Leavitt published an article on a CEP language project, *Rapide*, at Stanford University directed by David Luckham [Lea09]. During the same time period there were three other research projects: Infospheres in California Institute of Technology, directed by K. Mani Chandy; Apama in University of Cambridge, directed by John Bates; and Amit in IBM Haifa Research Laboratory, directed by Opher Etzion. The commercial products that followed were dependent on the concepts developed in these and later research projects.

The CEP community started a series of event processing symposiums organized by the Event Processing Technical Society, and later by the ACM DEBS conference series[2]. An output of the event processing community was the event processing manifesto [CEvA11] from the second Dagstuhl seminar on event processing. The goal of the seminar was creation of a comprehensive document explaining event processing and its relation to

[2] http://www.debs.org/

other technologies and to suggest future work in terms of standards, challenges, and shorter-term research projects. Of the research topics identified, *a*) probabilistic events, *b*) provenance (i.e. event precedence), *c*) event context, *d*) function placement with optimization, and *e*) consistency are relevant to my thesis. Additional topics identified in the seminar for near-term research require application of neural-network technology. There is no direct correlation between models of computation and allocation methods for distributed heterogeneous compute elements and events and actions as, *f*) goal-directed reaction, *g*) retraction, *h*) prediction with speculation, and *i*) adaptive event processing. are attributes of database applications which are outside the focus of my thesis. The manifesto had a final set of topics for near-term research including access control, authenticity (while semantically different, highly overlaps provenance), and privacy. These are capabilities of a security system which is outside the focus of my thesis.

2.2.1 Probabilistic Events

Making a distinction between actual events and notification of events subsequent to observation is important since any observation can be perturbed. Therefore event notifications can be viewed as describing captured event data with a level of certainty or confidence. This concept has been widely explored in the context of databases. Such confidence can also be associated with predictions of future events; thus, the metric is either continuous, discrete, or a probability density function. Modeling such events benefit from probability theory and stochastic analysis using tools like PRISM by Kwiatkowska, et.al. from Oxford University [KNP11]. For allocation of processing to heterogeneous systems stochastic analysis and simulations, Shestak, et.al. [SSMS08], have shown the probability that the performance of a system, β, in the interval of $[\beta_{min}, \beta_{max}]$, which is the acceptable range of possible variation in system performance, has a *"robustness metric"*, θ (an artificial metric); and $\theta = \mathbb{P}[\beta_{min} \leq \psi \leq \beta_{max}]$ where ψ is a unit-less numeric value of the randomness (i.e. uncertainty) of a system with its stated execution performance. Thus, θ provides a comparative measure of the probability that a system will satisfy a given quality of service (QoS) where $\theta = 1.0$ is unity for the robustness metric. In [SMS14], Smith et.al., build on this concept for developing a non-zero probability of addressing the timing constraints of a system by maximizing the probability that the QoS is achieved through path re-linking and local search within a genetic algorithm. In [TLL+11], Tang et.al. describe a stochastic heterogeneous earliest finish time (SHEFT) scheduling system for precedence constrained tasks in a parallel application with random tasks processing time and communication time on grid computing systems that minimize the makespan.

Each CEP method has liveness and safety as a goal so that a system has utility as without this you have a quiescent, or dead-locked, system that is without utility. In mapping applications the allocation mapping must have a concept of which computing elements have sufficient "robustness" so that events are processed without undue latency. This type of evaluation is a heuristic of an element in a cluster/cloud system that helps determine which, and how many, components of a mapping are assigned to the computing elements. This is most definitely a scheduling issue and the mapping algorithm must take scheduling into account which will be examined in Chapter 3.

2.2.2 Event Precedence

Knowing event precedence is important in understanding the flow of data in a distributed application. It has been felt that this area has been under-addressed so far in [CEvA11]. This area is related to probabilistic event management and event origin is an attribute that affects architecture choice and partitioning of algorithms across compute elements. Historically, this has been known for some time [BWF+96] and some auto compile time work has been done [FK97, FTL+02, CSJN05].

By using event precedence it is possible to use fundamental theory of events in concurrent and distributed systems for reasoning about causality and mechanisms for identifying logical or vector clocks [RS96] which is, of course, dependent on the seminal idea of *Lamport timestamps* [Lam78] and is the basis of RITA event management for CEP where the concept in Lamport's work of "happened-before" — ergo, "$\alpha \rightarrow \beta$" in Lamport's notation — is codified by the event ordering of "$\beta \leftarrow \alpha$" in RITA with multiple event ordering being explicit as needed, thus "$\beta \leftarrow \alpha \leftarrow \gamma \leftarrow \delta \leftarrow \ldots$" where the resultant, β, is a function of all the prior events and their interactions. It is possible in RITA to have such an explicit ordering of events and have a predictable network of precedence as shown in Figure 2.1 where event B has such a precedence. As systems would be hard-pressed to track all event relations and store all events ever observed during application execution to allow any event to carry its entire genealogy it is important to have events as closed, directed graphs. This concept will be explored more in in §2.4.8 where we discuss the temporal logic model.

2.2.3 Event Context

Real-world events are most commonly associated with time and space dimensions, mirroring the most common inquiries about such events, namely *when* and *where*. True distributed systems, as compared to clusters, typically do not have synchronized time which motivates a discussion about logical and vector clocks and applications, as just

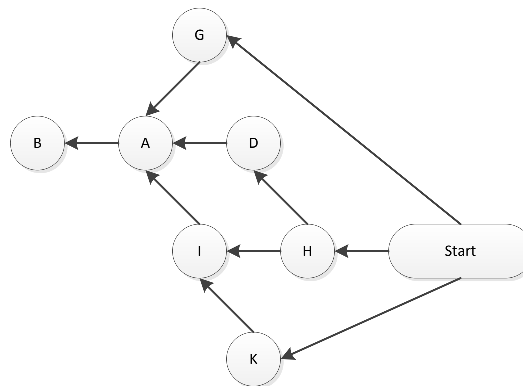


FIGURE 2.1: Network of Event Precedence

discussed in §2.2.2, would be more sensitive to which software component created a given event.

Luckham has a well regarded text on the subject [Luc02] and describes event context and event pattern matching with regard to his language, *Rapide*. The Esper language manual[3] has a description of context as:

> Context-dependent event processing occurs frequently: For example, consider a requirement that monitors banking transactions. For different customers your analysis considers customer-specific aggregations, patterns or data windows. In this example the context of detection is the customer. For a given customer you may want to analyze the banking transactions of that customer by using aggregations, data windows, patterns including other EPL constructs. In a second example, consider traffic monitoring to detect speed violations. Assume the speed limit must be enforced only between 9 am and 5 pm. The context of detection is of temporal nature. A context takes a cloud of events and classifies them into one or more sets. These sets are called context partitions. An event processing operation that is associated with a context operates on each of these context partitions independently. *(Credit: Taken from the book "Event Processing in Action" by Opher Etzion and Peter Niblett.)*
>
> A context is a declaration of dimension and may thus result in one or more context partitions. In the banking transaction example there the context dimension is the customer and a context partition exists per customer. In the traffic monitoring example there is a single context partition that exists only between 9 am and 5 pm and does not exist outside of that daily time period. In an event processing glossary you may find the term event processing agent. An EPL statement is an event processing agent. An alternative term for context partition is event processing agent instance. Esper EPL allows you to declare contexts explicitly, offering the following benefits:
>
> - Context can apply to multiple statements thereby eliminating the need to duplicate between statements the context dimensional information.
> - Context partitions can be temporally overlapping.
> - Context partitions provide a fine-grained life cycle that is independent of the life cycle of statement life cycle.

[3] http://www.espertech.com/esper/

Of note, RITA is novel as it defines event context for processing — which is defined in a temporal setting — without extant grouping and artificial syntax. RITA captures this via the guarded condition event matrix.

2.2.4 Function Placement

Placing functions across physical resources, i.e. computing elements, requires breaking-down processing into elementary operations and placing them on physical and logical entities capable of hosting event processing agents. Improvement is needed in dynamic placement strategies with proactive behavior, based on fluctuations in application load and load distribution. Current models and systems react to spikes in activity, at best, and event processing operations are considered static and do not support instance-adaptive or speculative event processing. Current work from Microsoft on the Dandelion heterogeneous compiler [RYC+13] and the IBM Liquid Metal Project Lime heterogeneous compiler [ABB+12] provide exceptionally interesting tools for mapping algorithms to heterogeneous computational elements. These tools will be further discussed in §3 for algorithm decomposition and mapping.

2.2.5 Consistency

Current literature shows several approaches for managing events by considering shortest-path and other metrics to ensure low-latency [SRHZ14, LZG+14, ADR14, CLB14, KMS14]. With this, different event processing agents are seen as combining the same events in different sequences. This leads to observers of the same complex events seeing apparently contradictory outcomes which can trigger conflicting reactions. This makes replicating event sequences difficult. Several solutions use manual deployment proxies that multiplex complex events and give ordered output with other techniques that have application in only special cases. From the literature, these methods shift the issue to an intermediary software layer than providing a general solution.

2.3 Distributed Event Based Systems (DEBS)

In §2.2 complex event systems topics are discussed and, as noted, the DEBS conferences and event processing community were moved to DEBS systems. In this section a survey of the research and industrial systems are given as a background of what have become understood systems and systems in practice.

In [RSS07] Hermes, Gryphon, Siena, Esper, Borealis and Aurora (now just Borealis), and AMIT are discussed. One of the first distributed content-based pub/sub systems was the Scalable Internet Event Notification Architecture (Siena) which supported restricted event patterns, but not a complete pattern language. Distributed pub/sub architectures Hermes, Gryphon (now part of IBM MQ Event Broker), and Siena only provide primitive events. Hermes is a pub/sub system of network event brokers decoupling publishers and subscribers. This is in contrast with Cambridge Event Architecture extensions to middleware for closely coupled components. Hermes uses XML for event transport while allowing standard programming languages such as Java for typed-event programming in end systems for distributed, event-based middleware architecture making use of a typed event model, object-oriented programming languages. Further, it has routing algorithms for avoiding global broadcasts and fault tolerance mechanisms.

Esper was a research project, but is now a product from ExperTech[4], and is an Open Source event stream processing solution for analyzing event streams. Esper supports conditional triggers on event patterns, event correlations and SQL queries for event streams. It has a lightweight processing engine and is currently available under GPL license [RSS07].

Borealis is a second-generation distributed stream processing engine developed at Brandeis University, Brown University, and MIT. Borealis inherits core stream processing functionality from Aurora [CcC+06] and distribution functionality from Medusa[SZS+03]. Borealis modifies and extends both systems to provide advanced capabilities commonly required by stream processing applications.

AMIT (now an IBM e-business Management Service offering) is an event stream engine providing high-performance situation detection mechanisms. AMIT has a sophisticated user interface for modeling business situations based on the following four types of entities: events, situations, lifespans and keys.

In [MFP06] Java Event-Based Distributed Infrastructure (JEDI), Rebeca notification service, Cambridge Event Architecture (CEA), Elvin a notification service, READY event notification service, and Narada Brokering project research projects are discussed. Also discussed are the commercial JMS pub/sub systems: IBM MQ, TIBCO, and Oracle.

The JEDI project does not seem to be active since 2001. JEDI was a distributed content-based pub/sub system. It has tuples of name/value pairs called event parameters. JEDI used event dispatchers in a tree structure with routing performed hierarchically. Subscriptions propagated upwards in the tree with state maintained at the event dispatchers. Events propagated upwards and followed downward branches when

[4] `http://www.espertech.com`

encounter a matching subscription. Hierarchical routing obviated advertisements to restrict the propagation of subscriptions.

Rebeca was a PhD thesis demonstration of implementing a pub/sub interface with a simple event system for distributed notification comparable to Siena and JEDI. Rebeca was based on a formal specification that defines the intended behavior of the notification unambiguously, as does RITA. Rebeca had extensible data and filter model. Rebeca was designed to support various routing algorithms with visibility control through use of scope for notifications.

The CEA, JEDI, Siena, Hermes, Gryphon, Esper, Rule-Core, and AMIT Research projects – and industrial solutions – work on event stream processing (ESP) and complex event processing (CEP). These two approaches address processing large amounts of events delivering real-time communication, allowing closed loop decision making, and continuous data integration [RSS07].

The Cambridge Event Architecture (CEA) was created in the early 1990s to address the emerging need for asynchronous communication in multimedia and sensor-rich applications. It introduced the publish-register-notify paradigm for building distributed applications allowing simple extensions of synchronous request/reply middleware (CORBA) with asynchronous publish/subscribe communication. This research project is maintained by the University of Cambridge Opera group as an archived project.

Elvin is a notification service for application integration and distributed systems. It features a security framework, internationalization, and "pluggable" transport protocols, and has been extended to provide content-based routing of events. Events are name/-value pairs with a predicate-based subscription language. Elvin has a source quenching mechanism where event publishers request information from event brokers about subscribers currently interested in their events. If there are no subscribers, publishers can stop publishing events which reduces computation and communication overheads. This is a very similar construct in RITA, with a notable exception, as RITA publication is self limiting based on event data being significant, not based on available listeners. Elvin was sponsored by The Distributed Systems Technology Centre (DSTC) which was supported by the Australian Government's Cooperative Research Centre (CRC) program. DSTC, a CRC, completed its operations on 30 June 2006.

The READY event notification service, developed at AT&T, introduced event zones to partition components based on logical, administrative, or geographical boundaries and to delimit the visibility of events. Boundary brokers connect zones and control the communication between them, and may enforce security policies on connected clients. Although similar to scoping, zones resemble more the domain idea of CORBA as it

mainly addresses control on the physical routing network; the engineering aspect is lacking. For instance, in READY a component belongs to exactly one zone so that there is only a two-level hierarchy. This project has been dormant since 2000.

The Narada Brokering project, sponsored by the University of Indiana, provided a unified messaging environment for grid computing, which integrates grid services, JMS, and JXTA. It is JMS compliant and supports a distributed network of brokers as opposed to the centralized client/server solution advocated by JMS. The JXTA specification is used for peer-to-peer interactions between clients and brokers. Events can be XML messages that are matched against XPath subscriptions by an XML matching engine. The network of brokers is hierarchical, built recursively out of clusters of brokers. Every broker has complete knowledge of the topology, so that events can be routed on shortest paths following the broker hierarchy. In general, there is the additional overhead of keeping event brokers organized hierarchically, which can be costly. Dynamic changes of the topology are propagated to all affected brokers. This project has been dormant since 2009.

2.4 RITA Theory

RITA has seminal portions of its constructs based on the influence of queueing theory [GSTH08], using Hoare logic for partial correctness of programs and Communicating Sequential Processes (CSP) [Hoa85], temporal theory from Kröger [KM08], with schema formalism from the Z notation codified by Spivey [Spi92].

2.4.1 Events and Event Propagation

We will start with a definition of event propagation. Events, ϵ, are a tuple of $\{\lambda, \delta\}$ where λ is *Name* and δ is *Data*. Propagation is controlled by input of ϵ to an *Action*, α, which may transform δ which modifies the event state:

$$
\begin{aligned}
&\epsilon_{\{\lambda,\delta\}} \rightarrow \alpha \rightarrow \epsilon_{\{\lambda',\delta'\}} \quad \therefore \\
&\forall \epsilon : \alpha(\epsilon) \Rightarrow \epsilon'
\end{aligned}
\tag{2.1}
$$

Events, by themselves, can be informative regardless of λ or δ. The arrival of an event may be considered as actionable and thus an event with the tuple $\{\lambda = \varnothing, \delta = \varnothing\}$ is therefore valid in the event system. This construct is very similar to the $c.v$ tuple in CSP. While Hoare does not indicate that $c.v$ must have values, one may safely assume such as

channel connections are critical in his concept of communication. The advancement from Hoare's system to RITA is the concept that communication in a channel is named and, *ipso facto* the event, ϵ, has a name and ϵ can travel along any channel. This is a semantic improvement as the λ (c, channel, in CSP) and δ (v, value, in CSP) may be null but the ϵ will still have semantic meaning due to its existence, or not, which is respectively a discrete member of the set $\{1, 0\}$. Another semantic advancement from CSP in RITA is that a channel may exist or may not exist in RITA, as only a named communication mechanism providing transport for information, so therefore a connection to a channel is not needed for an event to exist.

Having given a description of event constructs it is now possible to combine these individual events into a coherent construct. Thus given the equation in 2.1, a sentence, σ, is a sequence of actions which are connected by events. A sentence has a start state A, with an event initialized to $\varnothing$; An event is sent to an action where a modification, ϵ', occurs; it is then input to subsequent actions, where a modified ϵ leaves an action, and thus the event propagates through the system. The final event state, ϵ^n, is deduced as being the terminal event state, Ω.

$$\sigma : \{\vDash A(\epsilon_{\{\varnothing,\varnothing\}}) : \epsilon \rightarrow \alpha_i \rightarrow \epsilon' \rightarrow \alpha_{i+1} \rightarrow \epsilon'' \ldots \alpha_n \rightarrow \epsilon^n \vdash \Omega\} \tag{2.2}$$

This sentence is the basis for event sequences in RITA and becomes more interesting when multiple events enter a single action. With algorithm decomposition, theoretical inputs to α can be truly in parallel with instantaneous arrival leading to race-conditions. With a set of independent actions and events, true parallel action and execution is possible. Deriving from equation 2.2, and joining each sentence, σ_n to a singular action, α_1 we can write:

$$\sigma_i \rightarrow \alpha_1 \parallel \sigma_{i+1} \rightarrow \alpha_1 \parallel \ldots \sigma_n \rightarrow \alpha_1 \tag{2.3}$$

graphically this would be:

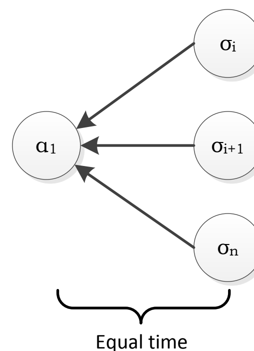


Figure 2.2: Parallel Event Arrival

Electrically, this would be an allowable construct. In circuit design logic this physical construct would be an error and design rules would be employed as described in [GSA90] for multiplexing of signals, registers, channels, memory, or buffers so there would be a sequence of unordered events entering an action sequentially (see CSP Concurrency in [Hoa85]). In order to show the isomorphic relationship, the set Θ is established based on equation 2.2 so that we can have R as the strict relation "$\rightarrow$" be a *transitive* relationship where $a\ R\ b$ and $b\ R\ c \rightarrow a\ R\ c$ for every $a,\ b,\ c \in A$ and be *asymmetric* where $a\ R\ b \rightarrow \neg(b\ R\ a)$ for all $a,\ b \in A$. Adding the temporal operators[5] so that in comparison to the expected partial ordering relationships shown as, "$a \leq b$" or "$a \subseteq b$," the relationship is now shown as classical logical form using a Kripke structure [Kri63] with the temporal relationship of $\mathcal{K}_i(\bigcirc \epsilon) = \mathcal{K}_{i+1}(\epsilon)$. Also the form, $A \rightarrow \Diamond B$, is a typical formula used for time-forward projection. Given a sequencing of events (as actions do not propagate) we can now show the set, partial order, and temporal order of events:

$$\Theta : \{\epsilon \ldots \epsilon^n\} \quad \texttt{Set definition.} \tag{2.4a}$$

$$\sigma \longleftrightarrow \{\epsilon \in \Theta \mid \epsilon \preccurlyeq \sigma\} \quad \texttt{Partial order.} \tag{2.4b}$$

$$\epsilon \rightarrow \Diamond \epsilon \quad \texttt{Temporal order.} \tag{2.4c}$$

Equation 2.4b shows a partial order and with every partial order having the "same shape;" thus we have an isomorphic collection ordered on temporal attributes.

In theoretical discussions event transitions are assumed to be instantaneous and occur in zero time. This is not a practical modeling technique, expeditious yes, but not practical; nor is it, in actuality, true especially in a distributed system where transmission latency can be long and erratic, e.g. a packet switched network transmission, both in a macro sense on a system to system transmission medium (a system NIC using CAT 5e, Inifiniband, PCIe, etc.) and even with network-on-chip (NoC) routers using a slot-ring guaranteed hard latency [HKKB13], therefore time must be accounted for in event transitions as well. Efforts have been made to reduce such latency to asymptotic to zero by specialty hardware design. For example, the D.E. Shaw company created their Anton system [DGM+11] with extensive optimizations but even with such optimization there are limits to how much latency can be reduced.

Given that latency can not be eliminated, determining which portions of an algorithm can, or can not, be distributed across computing elements is controlled by a latency cost function. While any processing can be distributed it is the cost of that distribution, measured in time, that is minimized in a latency cost function. Minimum latency has a

[5] $\bigcirc$, $\Box$ and $\Diamond$ known as *next-time*, *always* (or *henceforth*), and *sometime* (or *eventuality*) operators, respectively. Formulas $\bigcirc A$, $\Box A$, and $\Diamond A$ are read "next A," "always A," and "sometime A."

direct relationship with processing throughput. In RITA, there are explicit time relationships for guarded algorithm activation and in order to have proper temporal attribution it is necessary to model event propagation so latency can be evaluated for algorithm components and a decision be made on distributing an algorithm. Using gather-scatter or MapReduce techniques have been codified for years, but no temporal evaluation is currently known for general codes. From my work developing standards on integrated circuit electronics, VHSIC Hardware Description Language (VHDL)[IEE88, IEE09] and Property Specification Language (PSL) [IEE12], I have seen that there has been significant work in the electronics industry on understanding temporal issues in design but there has been few temporal systems developed for algorithm decomposition. With the advent of grid/cloud/cluster computing resources becoming numerous, there has been some recent efforts in this area.

Current work from Microsoft on estimating distributed event latency using continuous queries using LINQ, Esper, or StreamSQL demand long periods (weeks to month) of data collection to train the Maximum Cumulative Excess (MACE) [CGB+11] model. This model, while shown to be very accurate, uses a form of stochastic calculation for processes within a cluster without DAG consideration. Of note is the work done by Ferguson, et.al. from Microsoft on the Jockey system [FBK+12] which focuses on service level objectives (SLO) as a new concept over service level agreements (SLA) which have been the industry standard of service time expectation by contract. In contrast with SLA, SLO makespan times are derived from contractual agreements to ensure that missing a makespan finish will not be financially detrimental to the business. With the final output often being the output products of pipelined processes, a makespan finish time on the final output leads to many internal deadlines for processes that are included in the final output makespan time. As such, many internal deadlines are "soft" so that a finish time of two hours instead of one is undesirable, but does not cause a loss of revenue or financial penalty whereas in a SLA, an internal process that does not finish as per contract would cause a penalty. Thus in a single cluster running a large number of concurrent processes where some have no deadline, some have a soft deadline, and some have very strict deadlines the use of weighted fair sharing would not map latency objectives for each of type of deadline onto an appropriate processing stream. By directly specifying a utility function to indicate the deadline and importance of a process alleviates this problem. With Jockey, as with MACE, training for mapping from empirical data must still be done. The improvement of Jockey over MACE is that Jockey uses the compiler intermediate form DAG for code developed for the Microsoft Cosmos system using the Dryad software stack [IBY+07] or SCOPE (Structured Computations Optimized for Parallel Execution) declarative language for parallel and distributed programming. Of the two, Dryad is more interesting as it is a general-purpose distributed execution engine for coarse-grain

data-parallel applications. A Dryad application combines computational "vertices" with communication "channels" forming a data-flow graph. Dryad runs the application by executing the vertices of this graph on a set of available computers, communicating as appropriate through files, named pipes, and shared-memory FIFO buffers.

Latency considerations strongly influence the choice and performance of network algorithms, such as routing and flow control. Modeling this requires that simplifying assumptions be made as extensive, realistic modeling can make meaningful analysis extremely difficult due to the lack of true measurements; or as with the MACE and Jockey systems, extensive empirical training data must be available for any meaningful, accurate prediction. In doing this simplification, several queueing theory methods have been developed for this purpose [GSTH08]. These models do provide a basis for adequate delay approximations as well as statistically qualitative results. Looking at only the point-to-point mensuration for latency, Bertsekas and Gallager [BGH92] describe four components contributing to latency:

- Processing delay: the delay between the time the packet is correctly received at the head node of the link and the time the packet is assigned to an outgoing link queue for transmission, with an addition of delays introduced at DLC and physical layers.
- Queuing delay: the delay between the time the packet is assigned to a queue for transmission and the time it starts being transmitted. During this time, the packet waits while other packets in the transmission queues are transmitted.
- Transmission delay: the delay between the times that the first and last bits of the packet are transmitted.
- Propagation delay: the delay between the time the last bit is transmitted at the head of the link and the time the last bit is received at the tail node. This is proportional to the physical distance between transmitter and receiver.

Little's Theorem

Other basic concepts include Little's theorem [LG08] that states that the average number of customers in the store, L, is the effective arrival rate, λ, times the average time that a customer spends in the store, W, or put simply as: $L = \lambda W$ and this is an accepted, simplified estimation method that has been proven [Jew67, Eil69] to hold true for a number of queueing systems and thus can be used for measuring event arrivals with events queued, L, and event arrival rate being λ. Simplified models to a large extent can cover the basic behavior of the communication network as applied to load frequency control. These models are largely based on exponential arrival rates as this allows for several simplifications in quantifying the waiting time in queues and, for a dedicated communications network, the assumption that a Poisson distribution models the arrival rate is an accepted simplification technique.

2.4.2 Latency

Network communication latency for the current web-oriented, query-based, database access systems offered by Google, Amazon, Microsoft, Oracle, et.al. have several partial makespan models based on empirical data. In contrast, this work uses a less rigid, "mechanical," model and a more flexible organic model of latency. Most latency models are focused on networking and computing elements using a unit-time measurement for easily measured time of input to a unit, processing in a unit, and time of output from a unit. When measuring software components this networking and computing element mensuration concept is used, but this may not calculate the actual latency. Complex distributed and federated software used in cloud and distributed systems have dependencies on the latency of prior events for current event latency causing a propagation that is not a direct stimulus-response. This is because not all stimuli are actionable causing delay in propagation as there is a temporary quiescence in the system. This variable latency is due to decision logic comparing inputs and algorithmically evaluating the importance of the data.

2.4.2.1 Markov Models

The Markov model, particularly the hidden Markov model, has been shown to be a good technique for solving prediction problems where empirical data and prior state with current state probabilities are used to predict future states. Both Markov and hidden Markov processes are stochastic and do not have to be continuous. Xie, et.al. [XHTH13] have used Hidden Markov models to present approaches to predicting network traffic — called a "nested hidden semi-Markov model" — which includes a nested latent semi-Markov chain and one observable discrete stochastic process and has shown through a second work [XHX+13] a novel hidden Markov modeling method for an algorithm driven, two layer hidden Markov model for prediction of arrival of network traffic. Of interest is the 2011 dissertation work by Caravagna at Universitá di Pisa [Car11] where recognition of delay differential equations as being a better method and basis for a non-Markovian process algebra with delay stochastic simulation algorithms, based on a well-known stochastic simulation algorithm, are combined with delay differential equations to improve simulation performance to handle random variables. The stochastic basis and non-continuous behavior of Markov processes are not advantageous for RITA modeling. RITA modeling is not based on probabilistic occurrence, but is based on delay differential equations, which are the preferred method of calculation of latency.

2.4.2.2 Delay Differential Equations

To regulate variable latency, RITA models actionable stimuli uses delay differential equations (DDE) to account for the propagation time of events. These equations are a different form of ordinary differential equations as the derivative at any time depends on the solution at prior times. A description of the DDEs used begins with the simplest constant delay equations. The derivative with respect to time ($\dot{x} = \frac{dx}{dt}$) is used, by convention, giving the general form for the DDE:

$$\dot{x} = \mathbf{f}(x(t), x(t-\tau_1), x(t-\tau_2, \ldots, x(t-\tau_n)) \tag{2.5}$$

where the time delays, τ_i, can be positive constants (which are fixed, discrete delays), state dependent delays (the τ_i's depend on x), or distributed delays where the right-hand side of the differential equation is a weighted integral over past states. DDEs provide a more realistic evaluation of time for distributed assumptions over traditional point-wise modeling assumptions. As a simple example, start with a given DDE of $\dot{x} = -x(t-1)$ and suppose that we have $x(t) = f_{i-1}(t)$ over some interval $[t_{i-1}; t_i]$ and over the interval $[t_i; t_{i+1}]$, by separation of variables and using Myshkis **method of steps** we derive the instantaneous value of $x(t)$ by:

$$\int_{f_{i-1}(t_i)}^{x(t)} dx' = -\int_{t_i}^{t} f_{i-1}(t'-1)dt' \tag{2.6a}$$

$$\therefore \quad x(t) = f_i(t) \tag{2.6b}$$

$$= f_{i-1}(t_i) - \int_{t_i}^{t} f_{i-1}(t'-1)dt' \tag{2.6c}$$

As with other types of equations, we derive a lot of insight from a stability analysis of the equilibria. An equilibrium point is a point in the state space for which $x(t) = x^*$ is a solution for all t. Thus, for a DDE of the form shown in equation 2.5, the equilibrium points must satisfy

$$f(x_1^*; x_2^*; \ldots; x_n^*) = 0 \tag{2.7}$$

and where a system is described by one differential equation, or a system of differential equations, the equilibria can be estimated by setting all derivatives to zero. Understanding the stability of a system is required as regulated processing is is only effective when there is no unstable processing due to resource exhaustion. DDEs are primarily used in the study of biological systems and as such there has been work on several forms of DDEs and techniques to find if a system described by a DDE is stable [LS11, May73, Gop92]. These models typically study the relationship between codependent populations of consumers (predators) and producers (prey). What is desired in these biological systems is

a steady state where any oscillation in the respective populations return to their equilibrium values so there is no "crash" (extinction) in either population. This is also true for distributing processing across multiple computing elements. The mapping from a biological system to a software, hardware, or both, system would be where the software application is the "predator" and the system resources are the "prey." For distribution of algorithms across computing elements system stability is needed to ensure that processing does not stall due to resource exhaustion, this is the "crash" state mapped from biological systems.

Our treatment of stability theory starts with the Logistic model that states:

$$\frac{dN}{dt} = \gamma N \left(1 - \frac{N}{K}\right) \tag{2.8}$$

where γ is the Malthusian parameter (rate of maximum population growth, see C.1), N is the population density, K is the carrying capacity of the population (the maximum sustainable population) and the equilibria is $\frac{dN}{dt} = 0$. Using the Logistic model for one variable as the base equation, detecting the stability of models with several variables requires solving systems of differential equations. Consider a predator-prey model with two variables: density of prey and density of predators. Dynamics of the model is described by a system of two differential equations:

$$f(t)\begin{cases} \frac{dH}{dt} = f(H,P) \\ \frac{dP}{dt} = g(H,P) \end{cases} \tag{2.9}$$

This is the 2-variable model in its general form. Here, H is the density of prey, and P is the density of predators. The first step is to find equilibrium densities of prey (H^*) and predator (P^*)

$$f(x)\begin{cases} f(H^*,P^*) = 0 \\ g(H^*,P^*) = 0 \end{cases} \tag{2.10}$$

Which leads to making the model linear at the equilibrium points, $H = H^*, P = P^*$, and then estimating the Jacobian matrix:

$$\mathbf{A} = \begin{vmatrix} \frac{\partial f}{\partial H} & \frac{\partial f}{\partial P} \\ \frac{\partial g}{\partial H} & \frac{\partial g}{\partial P} \end{vmatrix} \tag{2.11}$$

The eigenvalues of matrix A are then estimated with the number of eigenvalues equal to the number of state variables.

For equilibrium to be established the criteria of the resultant eigenvalues is:

1. If the real parts of <u>all</u> eigenvalues are *negative*, then the equilibrium is stable,
2. If <u>at least one</u> eigenvalue has a *positive* real part, then the equilibrium is unstable.

Eigenvalues have the same meaning as the slope of a line in phase plots. Negative real parts of eigenvalues indicate a negative feedback and thus it is important that all eigenvalues have negative real parts because if one eigenvalue has a positive real part then there is a direction in a n-dimensional space in which the system will not tend to return back to the equilibrium point, and consequently it is unstable.

Another model of stability is Ricker's model. This model is a discrete-time analog of the Logistic model:

$$N_{t+1} = N_t \, e^{\left[r\left(1-\frac{N_t}{K}\right)\right]} \tag{2.12}$$

Finding the equilibrium of the population density N^* is the solution to

$$N^* = N^* \, e^{\left[r\left(1-\frac{N^*}{K}\right)\right]} \tag{2.13}$$

This equation is obtained by substituting N_{t+1} and N_t with the equilibrium population density N^* in the initial equation. By inspection, the roots are: $N^* = 0$ and $N^* = K$. The first equilibrium, $N^* = 0$, is of no interest as there is no population. The estimate of the slope $\frac{df}{dN}$ at the second equilibrium point is:

$$\frac{df}{dN_t} = \left(1 - \frac{rN_t}{K}\right) e^{\left[r\left(1-\frac{N_t}{K}\right)\right]} \tag{2.14}$$

Applying the condition of stability: $-1 < (1 - r) < 1$, thus r is in the range: $0 < r < 2$; and Ricker's model has a stable equilibrium of: $(N^* = K \iff 0 < r < 2)$.

If a discrete time model has more than one state variable, then the analysis is similar to that done in continuous-time models. The first step is finding the equilibria. Second, make the model linear at the equilibrium state (estimate the Jacobian matrix). And third, estimate eigenvalues of this matrix. The only difference from continuous models is the condition of stability. Discrete-time models are stable (asymptotically stable) if and only if all eigenvalues lie in the circle with the radius = 1 in the complex plain.

Most DDEs do not have analytic solutions, so it is generally necessary to resort to numerical methods for solutions. Because the solutions have discontinuous derivatives at the knots[6] it is necessary to be careful when using numerical methods designed for ordinary differential equations (ODE) with DDEs. ODE integration generally assumes that at least the first few derivatives are continuous but this can go badly wrong at

[6] a knot K such that $\{K : K \subset \mathbb{S}^3\}$ which is homeomorphic to the circle $\mathbb{S}^1$

the knots. A best-practice is checking the results of numerical integration of DDEs by reducing the size of the time step and checking at least a few results with a second integration method and, if the results are in reasonable agreement, then the result can be safely used. A "reasonable agreement" is a result that is smaller than the **local truncation error** (LTE) solution using the Euler method.

Discussion of DDEs can result in a thesis topic by itself. It is not the intent of this thesis to provide an extensive discussion on the derivation and theory of DDEs but provide sufficient theory demonstrating that DDEs allow a forward-projection mechanism of latency for predictive allocation of software across computing elements without empirical training data. Automated DDE solvers exist[7] and are used to create latency cost functions given a polynomial for historic time.

2.4.2.3 Latency Cost Functions

Latency cost functions are nominally split between "hardware" and "software" domains. In highly integrated, multicore systems the assignment of latency between these two domains becomes fluid and mixed due to components being dependent on a combination of hardware and microcode instructions (i.e. software). This complexity is exacerbated by multicore devices having smaller form factors than in prior history resulting in more transistors per unit area with higher clock speeds where these dense systems have to have control software to support the more advanced and integrated systems. The interface of software and hardware becomes even more critical in these highly integrated systems requiring simultaneous construction of both the microcode software and hardware. When done correctly, multicore devices are easier to use, software is more easily added, and the hardware is more adaptable to new uses such as GPUs becoming GPGPUs.

Embedded systems demonstrate this due to multicore platforms having a heterogeneity of computing elements which require mapping applications efficiently on the available hardware components but also to determine which hardware components are necessary to satisfy the overall design objectives. This hardware and software co-design effort assumes it is possible to explore a large design space to find the best software mapping on the hardware. Such a design space can be huge especially when reconfigurable components are available that can support run time reconfiguration. An older study in 2008 by Sandia National Laboratory describes integration of hardware and software components for tight integration for future supercomputers using multicore processors [PKL08]. A 2013 survey by Ogras and Marculescu [OM13] addresses the current NoC integration issues which will be explored in §2.4.2.3.2.

[7] MATLAB toolbox `dde23`, R-Project library `deSolve`.

Focusing on a general model, such as Little's theorem discussed in §2.4.1, with its simplifications can provide sufficient fidelity for algorithm decomposition and distribution given that the decomposition is over a stable and sufficiently large distribution allowing the simplification to be statistically significant. Arriving at this simplified model for the complexity of latency is the topic of the following subsections.

Only considering hardware, latency cost functions are based on major latency components listed in Table 2.1 which excludes storage operations. While storage operations are important, as they relate to latency, there has been a constant stream of papers over the past twenty years on storage operation latency. This amount of literature is expected as storage latency is, by far, the largest latency for systems [RW94, UAM01, KRM08, CGS09, BVF+12, LKV14]. Storage latency is treated as a separate, additive latency cost for a system and is not integrated as a part of the RITA latency calculation for event propagation. It is separately modeled. Figure 2.3, shows this relationship. Storage components are modeled with CART-MARS[8] methods [LFSZ12a, LFSZ12b].

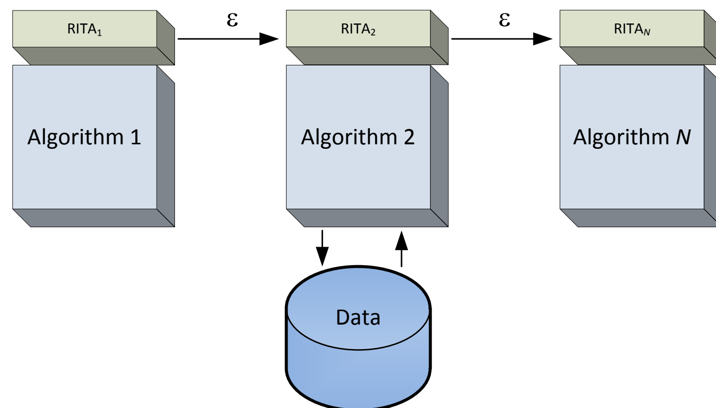


FIGURE 2.3: Modeling of processing components as they relate to event propagation Storage I/O is not considered in the cost model as RITA does has no direct control of algorithm I/O operations.

[8]CART: Classification And Regression Trees. MARS: Multivariate Adaptive Regression Splines

Table 2.1: Major Latency Components

(a) Processor Type

Graphical Processing Unit (GPU) – are built for very regular throughput workloads such as graphics, dense matrix-matrix multiply, and any data parallel operation. GPUs are notoriously poor at handling branching code
General Purpose Processor, Single core CPU (GPP) – Built for general logic, calculation, DMA, and subsystem control. GPPs are ill-suited for parallel operations
Digital Signal Processors (DSP) – are designed specifically to measure, filter and compress continuous real-world analog signals. DSPs can compete with GPUs for processing, but are not able to have as many threads active as are GPUs
Field Programmable Gate Arrays (FPGA) – have large resources of logic gates and RAM blocks to implement complex digital computations with very fast I/Os and bidirectional data buses. These systems are seen as flexible, fast ASIC-like elements
Many Integrated Core (MIC) – especially good as "embarrassing" parallel processing computation using SIMD. MIC suffers from poor programming practices that treat the MIC chip as a GPP
Heterogeneous System Architecture (HSA) – Combined CPU & GPU on a single die with high speed common memory. This is a design initially from AMD. nVidia has NVlink as an alternative

(b) Network

Network serialization – includes signal modulation and data framing (packetizing), size of transmitted packets (varies with link bandwidth)
Network processing delay – gateways, firewalls, switches, and routers determine filtering, encapsulation, MTU fragmentation. Network technology sophistication determines processing latency
Propagation delay – Data travels at a speed of approximately 4.76 microseconds per kilometer in copper (70% speed of light), ergo a 100 kilometer one-way propagation delay in the cable is 0.476 milliseconds;round trip; almost one millisecond
Inherent router and switch delay – is the time to shift packets from an ingress port to an egress port for the data-unit's destination address which is Layer 3 for router or Layer 2 for switch
Queuing delay – different ingress ports are heading to the same egress port concurrently. This resource contention is, called "head-of-line blocking," can lead to substantial latency. test

(c) Cache & RAM

L1 cache is accessed on every instruction cycle as part of the instruction pipeline and is broken into separate instruction and data caches
L2 cache is shared between one or more L1 caches and is often much larger than L1 cache. L2 cache is designed to minimize the miss penalty (the delay incurred when an L1 miss happens)
L3 caches are optional and, if present, specific to the design of the chip. L3 cache is mainly a benefit to large multiprocessor servers. Intel uses the L3 cache for inter-core communication in Nehalem and later CPUs. AMD uses a crossbar and only has L3 cache in parts like the FX which are derived from server dies
Off chip random access memory (RAM) which is accessed by the mainboard specialty bus designed for low latency RAM data retrieval

(d) Multi-core Network-on-Chip

Store-and-Forward (SAF) – Each router stores an entire packet in its buffer, and then it forwards the packet to the next node
Virtual-Cut through (VC) – Each router stores only fractions of a packet (flit)[a] in its buffer, and then it forwards the fractions to the next node
Wormhole (WH) Switching – The most common. Uses small buffers to store at least a header flit in each hop where each router can forward flits of a packet before receiving the entire packet

[a] flow control digit; smallest unit of information recognized by the flow control method.

2.4.2.3.1 Processor Type, Cache, RAM A search of the literature retrieves multitudes of bottom-up processing calculations for each of the major latency components in Table 2.1. Each component, especially processor type, cache, and RAM, have vendor specific, proprietary models; thus bottom-up models only have true utility when performing program tuning to a specific architecture supported by a specific environment.

In Table 2.1(a) we can see that the generally accepted processor types of GPU, GPP, FPGA, and DSP are augmented by specialized processor architectures of MIC and HSA. It is these specialty processors that are supplanting the generally accepted processor types found in cloud environments. This is due to the large MIC[9] and HSA processors reducing cost, latency, and power consumption. Industry is producing these HSA chips across the board to capture the cloud data center market. These vendors and their shipping, or near-term shipping, HSA products are listed in Table 2.2. Other HSA vendors exist (i.e. Tilera[10]), with products not being as much for cloud environments as they are mostly used for embedded, signal processing, or other specialty operations.

Table 2.2: Current Heterogeneous System Architectures (HSA)

Vendor	HSA Product
AMD	Kaveri APU, AMD's CPU + ATI GPU combination announced January 2014. Carrizo HSA 1.0 compliant announced November 2014, Plus "hUMA" Heterogeneous Uniform Memory Access, but the definition of UMA and NUMA are poor thus making the novelty of the term moot.
INTEL	Xeon E5-FPGA hybrid chip, announced 6 June 2014.
ARM	ARM/FPGA combination with the Xilinx Zynq processor, announced July 2013.
nVidia	NVLink — NVLink will only operate within a single chassis so clustering GPU accelerated machines will not be able to take advantage of the increased bandwidth using NVLink. It seems likely that interconnect vendors like Mellanox will produce interface cards that enable IB-NVLink transfers so communication between NVLink enabled nodes can bypass the PCI bottleneck. Announced March 2014.
IBM	nVidia and IBM will use NVLink and IBM's future versions of its Power CPUs. Announced March 2014.

In Table 2.1(c) the memory structures specific and tightly bound to the processor types in Table 2.1(a) provide shared data between heterogeneous processors (see Table 2.2). These have now taken on mass storage characteristics via development by Hewlett Packard of a "memristor," first theorized by Chua in 1971[Chu71]. While this device is not yet manufactured in quantity[11], it acts like a passive, nonvolatile, nonlinear resistor and can act as non-volatile solid-state memory with greater data density than traditional

[9]Intel Xeon Phi Co-processor 7120X is 16GB, 1.238 GHz, with 61 cores (60 usable)

[10]Tilera seeks to have a 100 core chip for cloud computing since 2009, currently it is shipping a 72 core chip, TILE-Gx72

[11]http://www.theregister.co.uk/2014/06/11/hp_memristor_the_machine/

disk with access times similar to DRAM. Such a memristor would replace both RAM and mass storage and further condense computing element components – and reducing off-chip resources in the process – reducing variables, but increases the complexity of the latency cost function by pushing RITA from the IP network into the chip NoC domain.

2.4.2.3.2 Network and Network-on-Chip In past studies on Internet Protocol (IP) the sources of latency are network interconnect and software protocol[12] [13] [ZKK12]. Significant latency for TCP/IP comes from the protocol software interface. In the protocol, endpoints are assumed to be completely asynchronous and are assumed to be unaware of each other. When a message arrives at an end point, the computing element must process an interrupt and software must discover the application that must process the new message via protocol stacks. After discovery, the application must be context switched and data is copied into the applications buffer before the message can be processed. Additionally, other overhead can occur resulting in several significant sources of variation. For example, a Network Interface Card (NIC) may use interrupt moderation to absorb interrupt processing overhead for a batch of packets. This technique can artificially add latency without regard for the latency sensitivity of a packet within the batch. The latency of IP networking is large for event processing. The next larger value for latency is storage operations as noted above. In an HPC environment multiple interconnect controllers and protocols are used. Of the top 500 supercomputers, 45% use Infiniband and 25% use Gigabit Ethernet[14]. Additional usage is shown in Figure 2.4.

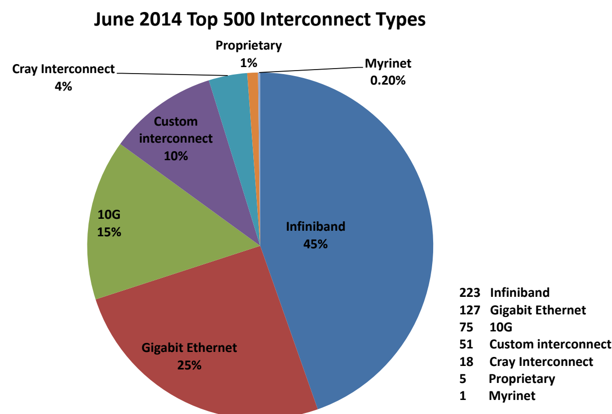


FIGURE 2.4: June 2014 Top 500 Interconnect Types

Traditional TCP/IP can be quite inefficient when transferring data. For example, when sending a file over a network there are four data copies and four CPU time-consuming mode switches between user space and kernel space. Currently there are

[12] http://queue.acm.org/detail.cfm?id=2071893
[13] https://www.usenix.org/legacy/publications/login/2008-10/openpdfs/walker.pdf
[14] http://www.top500.org/

three very high speed transports used in high performance computing that support a high speed method of network transport for remote direct memory access (RDMA): (i) The Internet Wide Area RDMA Protocol (iWARP) created in 2007 by the Internet Engineering Task Force (IETF) RFCs 5040–5042 and since 2007, extended RDMA through RFCs 6580, 6581, and 7306; (ii) An alternative is Infiniband protocol (IB), supported through the InfiniBand Trade Association comprised of a steering committee of companies: HP, IBM, Intel, Mellanox, Oracle, QLogic and System Fabric Works founded in 1999; (iii) The third is RDMA over Converged Ethernet (RoCE) which is a non-routable data link layer protocol allowing RDMA IP network rather than IB. These three transport stacks are shown in Figure 2.5.

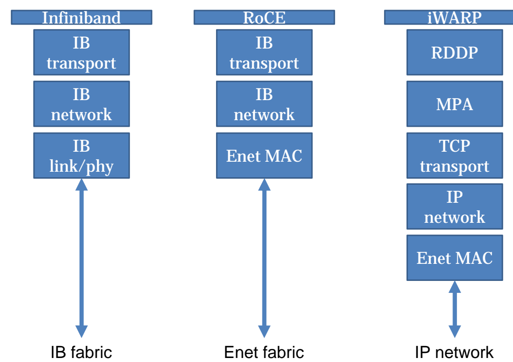


FIGURE 2.5: Three Transport Protocols

2.4.2.3.3 General Latency Equations Regardless of interconnect, the system to system networking sources of latency do follow general processing rules. Using data published by Siemens[15] in the following bullet-list and equations form the basis for calculation of IP networking latency for this work:

1. Store and Forward Latency (L_{SF}):
   Store and forward refers to the basic operating principle of an Ethernet switch. The term is descriptive of its actual operation: the switch stores the received data in memory until the entire frame is received. The switch then transmits the data frame out the appropriate port(s). The latency this introduces is proportional to the size of the frame being transmitted and inversely proportional to the bit rate as $L_{SF} = \frac{FS}{BR}$ where L_{SF} is the store and forward latency, FS is the frame size in bits, and BR is the bit rate in bits per second. For the maximum size Ethernet frame (1500 bytes[16]) at 100 Mbps the latency is $120\mu s$. For comparison, the minimum size frame (64 bytes) at Gigabit speeds has a latency of just 0.5 μs.

[15] http://w3.siemens.com/mcms/industrial-communication/en/rugged-communication/Documents/AN8.pdf

[16] Ethernet II framing, a.k.a."DIX" for DEC, Intel, and Xerox; the major design participants.

2. Switch Fabric Latency (L_{SW}):

   The internals of an Ethernet switch are known as the switch fabric. The switch fabric consists of sophisticated silicon that implements the store and forward engine, MAC address table, VLAN, and CoS, among other functions. The fabric introduces delay when executing the logic that implements these functions. The nominal switch fabric latency on products is 5.2 μs.

3. Wireline Latency (L_{WL})

   Bits transmitted along a fiber optic link travel at about two-thirds the speed of light, c, thus $L_{WL} = \frac{Distance_{meters}}{MediaSpeed_{seconds}}$. When very long distance Ethernet links are deployed, this delay can become significant. The one way latency for a 100km link works out to

   $$L_{WL} = \frac{1 \times 10^5 m}{(0.67 \times (3 \times 10^8 m/s))} \approx 500\ \mu s$$

   Note that for the distances involved in local area networks, this delay becomes trivial compared with the other contributions to latency.

4. Queuing Latency (L_Q)

   Ethernet switches use queues in conjunction with the store and forward mechanism to eliminate the problem of frame collisions that used to exist on broadcast Ethernet networks. Queuing introduces a non-deterministic factor to latency since it can often be very difficult to predict exact traffic patterns on a network. Class of Service (CoS) introduces a priority scheme to Ethernet frames to help mitigate queuing latency. It is a best-effort service, however, and cannot guarantee quality of service, since multiple frames at the highest priority level must still be queued relative to one another. Another consideration is that if a lower priority frame has already started transmission, then that frame must be completed before the switch may begin transmitting the higher priority frame. Calculating with absolute certainty the worst case latency for any Ethernet frame can be challenging. It requires detailed knowledge about all sources of traffic on the network. Specifically, one must know the maximum frame size transmitted by any device, the CoS priority of frames, and the time distribution and rate of frames. In an arbitrary communications network, little of this information is known and some assumptions have to be made. For a network with no traffic load, the queuing latency for a frame will be zero. For a loaded network, assume that the likelihood of a frame already in the queue is proportional to the network load. The average queuing latency can then be estimated as

   $$L_Q = (NetworkLoad) \times L_{SF(max)}$$

   where L_Q is the average latency due to queuing, Network Load is the fractional load relative to full network capacity and $L_{SF(max)}$ is the store and forward latency of a full-size (1500 byte) frame. For example, a network with 25% load would have

an average queuing latency of:

$$L_Q = 0.25 \times (12000bits/100Mbps) = 30\mu s$$

5. Total Worst-Case Latency Calculation (L_{TOTAL})
The latency sources described above are duplicated for every switch that an Ethernet frame must traverse on its journey from source to destination. Hence the general calculation for worst-case latency in a switched Ethernet network is expressed as:

$$L_{TOTAL} = \sum_{switches}(L_{SF} + L_{SW} + L_{WL} + L_Q) \text{ or}$$
$$L_{TOTAL} = (L_{SF} + L_{SW} + L_{WL} + L_Q) \times N_{switches}$$

where each contribution to latency is considered separately for each switch in the path. The calculation may be simplified considerably only considering the case where one traffic source has a high priority and is infrequent enough so that multiple frames of that type need never be queued at any switch in the network. Given this, the worst-case queuing latency is exactly the maximum sized frame in each switch in the path. The worst-case latency then simplifies to:

$$L_{TOTAL} = [(\frac{FS_h}{BR}) + L_{SW} + (\frac{FS_{MAX}}{BR})] \times N_{WSwitches} + L_{QWL(total)} \quad (2.15)$$

where FS_h is the size of the high-priority frame being considered (bits), $N_{WSwitches}$ is the store and forward combined with switch fabric latency, and $L_{QWL(total)}$ is the queuing and wireline latency due to the cumulative wire-line distance from transmitter to receiver.

2.4.2.3.4 NoC Network Interconnects A NoC depends on simpler networking devices than those of an IP network where event processing is highly effected by interconnections. The most common forwarding strategies for NoC interconnects are listed in Table 2.1(d). The design space for multicore processors is vast due to designs having complex problems in achieving the best architecture subject to a set of design constraints. With such a variety of core implementations, interconnect types, topologies, cache hierarchies, and memory management policies design choices have a power series in options in terms of the number and type of configurations to allow more cores and memory to fit into the die area. The complexity of the search space makes simulation-driven exhaustive exploration of all design points prohibitively expensive. An alternative is to decrease the number of data points for consideration by doing intelligent search, e.g., leveraging the methods of machine learning [KK08] or use design of experiments [SVL07]. While these methods may help in designing multicore chips, these methods do no result in predictive formulas facilitating software allocation to computing elements.

The microelectronics industry continues to push below the 28nm barrier demonstrated by a 7nm, and smaller, push by IBM[17] producing MIC GPPs, GPUs, and heterogeneous multi-core processors that are denser and have higher speed computing elements. Table 2.1(d) lists the three most common forwarding strategies for NoC interconnects. With such small feature sizes, and the ability to have well-developed NoC router cell designs, the design space for multicore processors becomes vast due to complexities of on-die space management for achieving the densest and most operationally efficient components. NoC implementations become a "best use" balance of a vast array of interconnect types, topologies, cache hierarchies, and memory management policies. Design choices become a power series in terms of the number and type of configurations to allow more cores and memory to be placed on the chip die area [OM13].

Memory traffic for MICs produces a cyclic latency dependency with the memory subsystem substantially affecting overall system performance. Nitkitin, et.al. [NdSPC13], developed an analytic method to estimate the performance of highly parallel MICs with hierarchical interconnect networks. Portions of Nitkitin's work rely on a fixed-point methods to account for the cyclic latency which is akin to DDEs. The utility of the method put forward by Nitkitin is the use of fixed-point methods for application to non-differentiable functions. A portion of Nitkitin's work has a model for power estimation that is outside the scope of this work and is not used. A more specific modeling technique using SystemC[18] for cycle-accurate evaluation of mesh, torus, and fat-tree NoC topologies was done by Weichen Liu, et.al.[LXW+11] using their Multi-Constraint System-Level (MCSL) benchmark suite for eight industry applications. Liu's work is specific for system design of SoC systems, and thus is not applicable to general solutions as is Nitkitin's work, which is more applicable to the scope of this thesis.

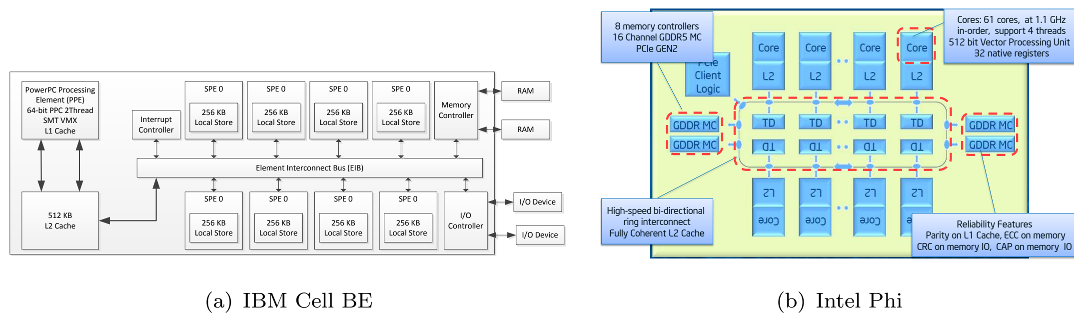
A queuing method of quantifying NoC latency has been put forward by Qian, et.al. [QJB+14]. In this work, channel waiting times in the router links are estimated using a generalized GE/G/1/K queuing model[19] handling "bursty" traffic (traffic that is quiescent and has random high input)and dependent arrival times with general service time distributions. DDEs depend on uniform functions and this modeling technique can help simulation of such "bursty" throughput. The queuing model does have a 13% error margin for the traffic patterns evaluated while providing about 70 times speedup compared to simulation. The authors goal was to provide a faster than simulation method. The utility of this method is that it can be used as a first level discriminant to expunge any assignments that have extremely high latency.

[17] http://www.cnet.com/news/ibm-spends-3-billion-to-push-the-far-future-of-computer-chips/

[18] IEEE 1666 http://www.accellera.org/

[19] General Exponent/General Job size/Number of servers/Number of places in system

Modeling heterogeneous computational devices that have multiple layers of parallelism, communication, and memory access with hierarchical characteristics is difficult to model. Li, et.al., [LZFD10] have extended PlogP to mPlogP (memory access PlogP). The mPlogP model comes from PlogP (Parameter LogP) by Kielmann, et.al., in 2000 [KBV00] and PLogP is an extension to the Culler, et.al., LogP model from 1993 [CKP+93]. The mPlogP model extends PLogP by replacing the static parameter L with dynamic parameter l, importing the parameter m to model the calculation, and adding another level on top of the conventional level to model behaviors of computational cores. In this way the model handles multi-level parallelization of the heterogeneous multi-core computer. To analyze parallel usage of heterogeneous multi-core computers, Li's model analyzes communication and memory access at different levels using memory access overhead to quantify calculation. The model states that it is capable of predicting the behavior of every part of applications guiding parallel optimization. While Li did find a strong correlation with the IBM Cell BE architecture (Figure 2.6(a)), the generalization of this algorithm to current multi-core architectures is questionable as MIC processor interconnect architectures (Figure 2.6(b)) are very different than the Cell BE architecture. The utility of Li's work is explicitly accounting for multicore memory access.



(a) IBM Cell BE

(b) Intel Phi

FIGURE 2.6: Comparison of IBM Cell BE™ and Intel Phi™ Block Diagrams

Another method of handling latency is to optimize the physical switch routing paths that data must take on a MIC chip. While this is a computing device optimization technique, it does have implication in allocating software to computing elements. Part of the allocation depends on a description of the device latency to be used in the cost function. The description must be sufficient to describe a base latency cost. This is not an obvious metric. As a tangible example, a comparison of the Intel Core i5 and i7 Lynnfield architectures only differ in two relevant performance areas, processor speed and hyperthreading. In Figure 2.7 the two processor family speed steps are compared. The figure shows the overlap or close values of the speed steps between the two families. With such identical or close processor speeds, the only true discriminant is hyperthreading. In the i5 Lynnfield there is no hyperthreading in the four core configuration. In the i7 Lynnfield there are two threads per core giving eight threads of processing for the i7

(four cores × two threads). With these cores, the efficiency of the NoC between devices becomes the true discriminant.

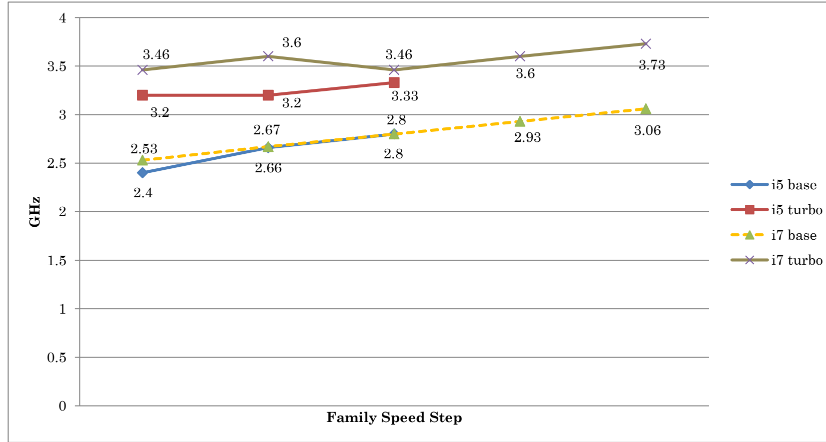


FIGURE 2.7: Comparison of Intel i5 and i7 Lynnfield family processor speed steps

Seiculescu, Rahmati, et.al. [SRM+13] use a worst-case latency algorithm for SoC topologies developed by Rahmati, Murali, et.al. [RMB+09]. They extended it with an iterative, modified Dijkstra's algorithm solving the single-source shortest path problem which adds cost to graph edges that have contention with higher priority (i.e. hard latency constraint) switch routing paths. This cost addition is used to provide floor-planning optimization for ASIC cells so time critical data flows do not compete for a switch with other flows. This latency reduction is for chip design and is not part of the scope of the research. The concept is important through. As mentioned, knowing the characteristics of a device is important in allocating software to computing elements. The work by Rahmati has utility for this work as it allows calculation of an upper bound where buffer depth of a switch is calculated as the sum of all buffers between the arbitration points of two consecutive switches. Since no traffic regulation is assumed in the model, the worst-case latency is achieved when all buffers are full and when the packet of a flow loses arbitration to all other flows that it can contend with. Under these assumptions, the upper bound on delay for a flow is given by Equation 2.16

$$UB_i = ts_1 + ts_2 + \sum_{j}^{h_i} u_{ij} \tag{2.16}$$

Where (i) UB_i is the upper bound delay for a packet of the ith traffic flow in the network to traverse the NoC, (ii) ts_1 and ts_2 represent the packet creation and ejection times which are constant, (iii) the sum adds the contribution of the worst-case interference at every hop, (iv) u_{ij} is the interference of other flows on flow i at switch j from the path of the flow for which the upper bound delay is calculated; Or as Dara Rahmati, wrote… "The time needed for packet i to go from the input buffer of switch j to the

input buffer of switch $j+1$," and (v) the number of hops, h, on the path of flow, i, is denoted by h_i.

In Cassidy and Andreas [CA12] a model is provided for a realistic, parallel processor program execution where processing cannot be perfectly divided across the number of parallel processors N. Rather, only some portion of the algorithm can be made parallel (the parallel fraction, F_p), while the remaining portion is executed sequentially (the serial fraction, F_s). Their model takes into account the *Meta-class* of instructions, indicated by M, time to execute t_i, and the probability density of the instructions $p(t_i)$. Where $p(t_i)$ is:

$$p(t_i) = \frac{Q_i}{\sum_{i=0}^{M-1} Q_i} = G_i \tag{2.17}$$

where Q_i is the quantity of instructions with the ith delay. To keep the equations clean $p(t_i)$ is replaced by G_i for use in the joint delay (J_D) equation (2.20). The serial fraction (S_D) executes with an expected delay as shown in Equation 2.18, while the parallel fraction (P_D) executes with an expected delay as shown in Equation 2.19. If there are an arbitrary number of parallel and serial portions, the serial and parallel results are joined reducing Equation 2.20 to Equation 2.21.

$$S_D = \sum_{i=0}^{M-1} p(t_i) t_i \tag{2.18}$$

$$P_D = \frac{1}{N} \sum_{i=0}^{M-1} p(t_i) t_i \tag{2.19}$$

$$J_D = \frac{F_p}{N} \sum_{i=0}^{M-1} Gp_i P_{Di} + F_s \sum_{i=0}^{M-1} Gs_i S_{Di} \tag{2.20}$$

$$J_D = \sum_{j=0}^{K-1} \frac{F_j}{N_j} \sum_{i=0}^{M-1} G_{ij} D_{ij} \tag{2.21}$$

Where, (i) The number of parallel processors is N, (ii) M are classes of instructions, (iii) K is the number of levels of parallelism, and (iv) Gp_i and Gs_i are the instruction distribution fractions with delays Dp_i and Ds_i for the parallel and serial portions of the algorithm, respectively. The serial fraction of the algorithm is where $\sum_{j=0}^{K-1} F_j = 1$ and where $N_j = 1$.

The calculations from Cassidy and Andreas depend on classes of instructions described by Waite and Goos in 1984 [WG84], revisited in 1995, where they describe four general classes of instructions for all architecture types:

> (1) Computation: Implements a function from n-tuples of values to m-tuples of values. The function may affect the state. Example: A divide instruction whose arguments are a single-length integer divisor and a double-length integer dividend, whose results are

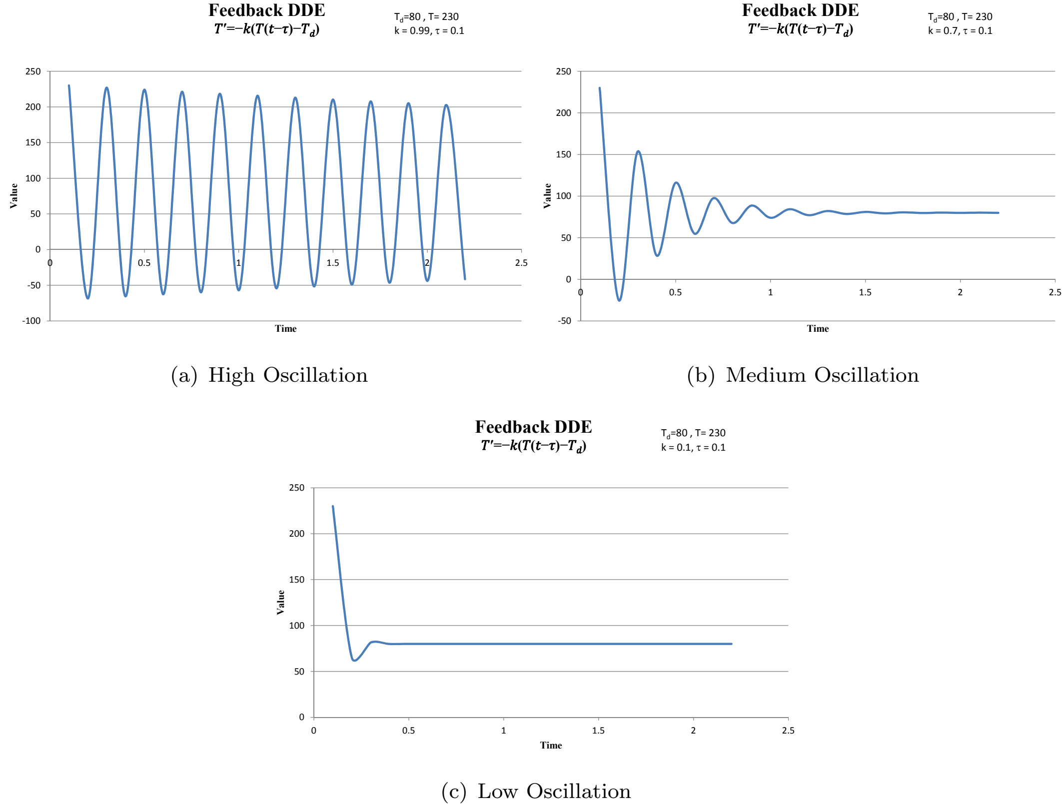
> a single-length integer quotient and a single-length integer remainder, and which may produce a divide check interrupt. (2) Data transfer: Copies information, either within one storage class or from one storage class to another. Examples: A move instruction that copies the contents of one register to another; a read instruction that copies information from a disc to main storage. (3) Sequencing: Alters the normal execution sequence, either conditionally or unconditionally. Examples: A halt instruction that causes execution to terminate; a conditional jump instruction that causes the next instruction to be taken from a given address if a given register contains zero. (4) Environment control: Alters the environment in which execution is carried out. The alteration may involve a transfer of control. Examples: An interrupt disable instruction that prohibits certain interrupts from occurring; a procedure call instruction that updates addressing registers, thus changing the program's addressing environment.

Combining Waite and Goos work classes of instructions with the classic work by Flynn [Fly66] that lists the definition of the four classes of system we begin to have a method of recognizing general codes and mapping to architectural types. Flynn's four classes are: (i) Single Instruction Stream-Single Data Stream (SISD), (ii) Single Instruction Stream-Multiple Data Stream (SIMD), (iii) Multiple Instruction Stream-Single Data Stream (MISD), and (iv) Multiple Instruction Stream-Multiple Data Stream (MIMD). With the continued development of SoCs, these four original classes have been augmented with tiled multiprocessor system-on-chip (MPSoC) platforms [WJM08, DWH14]. Since the early 1990s several ASICs for specialized processing existed and the Lucent Daytona™ chip as the first recognized cell-based MPSoC general purpose processor. The MPSoC platforms have been in use long before commercial multicore chips. One of the more popular multicore chips became common circa 2006 with development of the Core Duo 2™ chip from Intel. Both MPSoC and multicore chips are now in common usage. Many applications for MPSoCs are not single algorithms but systems of multiple algorithms. The type of computations performed at different portions of the application can vary widely: types of operations, memory bandwidth and access patterns variations argue for heterogeneous architectures.

2.4.2.4 RITA Latency Cost Function

The RITA latency cost function is comprised of elements from Equations 2.15 (latency for traversal of LANs), 2.16 (latency for traversal of a NoC), and 2.17 (latency for serial and parallel portions of code). The RITA latency cost function captures the meta-values of the latency elements. Attempting to capture the device specific latency elements varies by vendor and specific topology resulting in a cumbersome single cost function. To alleviate this cumbersomeness a parametric method identifying computing elements in a topology is recommended and is discussed in §3.3.2. As an example, to calculate the actual L_{TOTAL} the equation would require specific values for the $L_{WL(total)}$ variable that changes by vendor, by cable quality and type, by line segment geometry, and by

routing equipment vendor for each network route. Such specific knowledge is quantified by books written on the best practices and approximations for designing optical networks [Alw04, Vac05]. A software system should achieve a steady-state operation, executing on a steady-state hardware system. Thus transients should have a form that dampens oscillation through optimization tuning as illustrated in Figure 2.8.



(a) High Oscillation

(b) Medium Oscillation

(c) Low Oscillation

FIGURE 2.8: Levels of Oscillation based on Initial Conditions. Varying the magnitude of k, the corrective operation.

Equation 2.22 is the RITA DDE cost function that is derived by integrating time over all Latency, $\mathcal{L}$, by summing Network Latency, $\mathcal{L}_{\mathcal{N}}$, on-chip Switch Latency, $\mathcal{L}_{\mathcal{S}}$, and the sequential and parallel fractional portions of Program Latency, $\mathcal{L}_{\mathcal{P}}$.

RITA DDE Cost Function

$$\mathcal{L}' = \int_{(t-\tau)}^{t} \left[\left(\mathcal{L}_{\mathcal{N}}(t-\tau) + \mathcal{L}_{\mathcal{S}}(t-\tau) + \mathcal{L}_{\mathcal{P}}(t-\tau) \right] dt \qquad (2.22)$$

Where $\mathcal{L}_{\mathcal{N}} = L_{TOTAL} = [(\frac{FS_h}{BR}) + L_{SW} + (\frac{FS_{MAX}}{BR})] \times N_{WSwitches} + L_{QWL(total)}$; (2.15)

Where $\mathcal{L}_{\mathcal{S}} = UB_i = ts_1 + ts_2 + \sum_{j}^{h_i} u_{ij}$; and (2.16)

Where $\mathcal{L}_{\mathcal{P}} = J_D = \sum_{j=0}^{K-1} \frac{F_j}{N_j} \sum_{i=0}^{M-1} G_{ij} D_{ij}$. (2.21)

Using Equation 2.22 it is now possible to evaluate λ as a Gamma function for arrival of data and using a worst case $\mathcal{L}_{\mathcal{N}}$ as a ceiling, $\lceil W \rceil$, which is the waiting (latency) time, W, for Little's Theorem $L = \lambda W$. A ceiling function is used to only count whole transactions in the system as counting partial transactions is not a sound method for true throughput, truncating any fractional part.

Latency calculated for $\mathcal{L}_{\mathcal{N}}$ and $\mathcal{L}_{\mathcal{S}}$ accounts for channel capacity as expressed in the Shannon–Hartley theorem for channel capacity, C, in bits/second; where the highest upper bound on a coherent data transmission rate without error correcting of low bit error rate data sent with an average signal power, S, in watts or $(\text{volts})^2$ for an analog channel with additive white Gaussian noise of power, N, in watts or $(\text{volts})^2$; and where, B, is the bandwidth of, C, in Hertz thus:

Shannon-Hartley Theorem

$$C = B \log_2 \left(1 + \tfrac{S}{N}\right)$$

Specifically this is included in the physical media calculations for line speed in $\mathcal{L}_{\mathcal{N}}$ and flow contention (a "noisy" switch) in $\mathcal{L}_{\mathcal{S}}$.

The traditional approach of using queueing theory for designing quasi-optimal routing and flow control is: *a*) formulating steady-state queueing models for a network and then deriving an expression for a suitable performance measure in terms of the queueing model, *b*) optimizing mathematical programming to achieve a steady-state with adaptation done by varying incoming changing traffic, network topology, or the routing and flow control parameters. This approach assumes static loading conditions during each updating period allowing the network to attain steady state thus the network goes through a series of steady-state periods. The queueing models can be complex for dynamic networks and can be difficult to program and maintain. As an example, the Chapman-Kolmogorov differential equation is used to describe time-dependent state probabilities of a finite capacity M/M/1 queue with time-varying average arrival and service rates:

$$\tfrac{dp^{\kappa}(t)}{dt} = \lambda^{\kappa-1}(t)p^{\kappa-1}(t) - \mu^{\kappa}(t)p^{\kappa}(t); \kappa \in \{0, 1, 2, \ldots, K\}$$

where $p^{\kappa}(t)$ is the probability of κ units in the system (for both queue and service time t), $\lambda^{\kappa}(t)$ is the average arrival to the queue if there are κ units in the system, and $\mu^{\kappa}(t)$ is the average service rate with κ units in the system. K is the capacity of the system. This model of the queueing system is known to have an analytically difficult solution with the time-varying coefficients even if the arrival rate and service rate are constant — $\lambda^{\kappa}(t) = \mu^{\kappa}(t) = \mu$ — for a steady-state equilibrium. As a rule, the exact transient analysis of the M/M/1 queue and the transient behavior of $p^{\kappa-1}(t)$, requires an infinite sum of Bessel functions as shown in Asmussen [Asm87], Cohen [Coh82], Cox and Smith [CS61] and a survey by Tripathi and Duda [TD86] thus making queueing analysis a method that will not be used in RITA latency analysis.

2.4.3 Informal Event Model

Up to this point, the discussion has been on event theory, event propagation theory, and mathematical modeling of latency in order to form a foundation for a model of communication between federated, distributed systems. In this subsection, the discussion continues with the introduction of the RITA event model based on a canonical, minimal set of transformations that can be combined to create complex event interactions. The three canonical event forms used are “Spike,” “Set-at,” and “Transitional.” Each form occurs over a Δt for the local event space for each canonical event. In the model, time is defined as infinitely divisible, countable, and continuously and monotonically increasing, thus any incremental units of measurement are only sample points of infinitely countable time. This allows a base unit of time to be set by the computing element device clock signal. For example, in Intel i7 systems, the invariant Time Stamp Counter (TSC) is used which is the ratio between the invariant TSC frequency and the base clock which can be converted to a sampled ratio for a dynamic frequency estimate for each sampling period [Int14]. For nVidia GPU systems the `cudaEventCreate()`, and `cudaEventRecord()` allows kernel events to be measured, but does require a `cudaEventSynchonize()` call as well [NVI14]. For almost all FPGAs the use of counters is done at the MHz rate of the clock or else a system `clock()` call in a “soft” core FPGA.

This time definition allows discretized sampling at the granularity of the computational element and also simplifies the definition of an event “lifetime.” It is a common misconception that events must happen asymptotic to the speed of light: c [20]. This is not so. Real-time events happen in the time it takes for the task makespan “wall clock”[21] time to happen. This may truly be a microsecond, or several minutes, of wall-clock time.

Any perceived event lifetime is defined by the latency of its occurrence and its subsequent observance. This latency is graphically shown in Figures 2.9, 2.10, and 2.11 which illustrate the canonical forms by using electrical wave form graphs illustrating that one, or more, events causing a state change has some interval of time in which an event state is not known (as is true in voltage changes in circuits[22]) and, as in the physical world, time continues to increment, albeit by an infinitesimal amount, during an instantaneous state change. It is these cumulative Δt values that need DDEs for accurate modeling.

It is important to initially discuss the difference between a *set-at* event and a *transitional* event. While the wave forms used to illustrate the canonical events may appear the same as shown in Figures 2.10 and 2.11, note the definite differences in the time

[20] 2.99792458×10^8 ms^{-S} (exact)

[21] The time humans perceive, as opposed to simulation time (compressed), computational time (stuttered — only execution time counted, i.e. wait time excluded)

[22] As is shown in VHDL where signal values can be ’U’,’X’,’Z’,’W’,’L’, and ’H’ and not just ’0’ and ’1’

element t where, as is explained in Subsection 2.4.3.2, a *set-at* event can transition only once during the lifetime of the system; whereas a *transitional* event can oscillate between states σ_1 and σ_2 at any frequency greater than one. This temporal difference is very important in constructing a system of events without ambiguity as to the semantics of the intent of a state change.

The remainder of this Section builds the logical foundation of event semantics by describing the temporal mechanics of the canonical event forms. This is then further detailed in §2.4.5 where each form is formally described, cumulatively producing a basis for the RITA notation shown in §B which is enforced in the run-time environment by following these formally defined event forms and temporal specifications. This formalism is the core of the RITA concept; which is an advancement over the current state of the art and practice.

2.4.3.1 Spike Event

The spike event form in Figure 2.9 occurs at a Δt asymptotic to zero. Figure 2.9 has prior time shown as time in the past until event execution, $-\infty \cdots t_n$, where t_n is the "now" time of the event occurring that causes state change from σ_1 to σ_2 for a subdivision of time t_n shown as $\Delta t_{n_{0..1}}$. At the end of this asymptotic to zero time, the state changes back to the original σ_1 state for time $t_{n+1} \cdots \infty$. Events of this form are considered periodic "heartbeat" events. This event form can be used for counted threshold limits, keep-alive notification, or request for service. Typically this event is a transport layer UDP transmission. This event form can not be queued and has no lifetime.

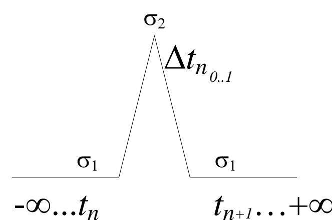


FIGURE 2.9: "Spike" Event Form.

2.4.3.2 Set-At

The "set-at" event form in Figure 2.10 is a permanent state change to a new state. State σ_1 remains unchanged until an event occurs and a permanent state change to state σ_2 occurs at the next time increment t_{n+1} from all time past for all time future. Past and future time are as described in §2.4.3.1. This event form can be used for one-time events such as system initialization and system termination. As shown in Figure 2.10, this is not the transitional event form as there is no ability for a transition from state σ_1 to return after transitioning to state σ_2. This event form has an infinite lifetime and can be queued.

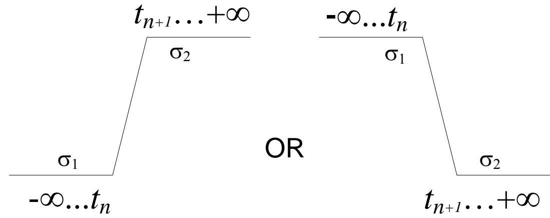


FIGURE 2.10: “Set-At” Event Form.

2.4.3.3 Transitional

The transitional event form in Figure 2.11 occurs over some period of measurable time with an event-sensitive entity perceiving a state change which separates this event form from the “spike” event form. Past and future time is as described in §2.4.3.1. This event form is semantically dense as each increment of time can result in a different state. The semantic meaning is compounded based on the frequency of state change and for how many sampling intervals of t the state remains constant. This event form has a limited lifetime, albeit the transition between $\sigma_1 \rightarrow \sigma_2 \rightarrow \sigma_1$ can be as long as the system needs it to be, thus it can appear as a set-at event form, but it is not because it can revert while set-at can not revert to its σ_1 state. Queuing of this event form requires prioritized time-based queuing that can be dynamically edited as higher priority events occur or as queued events expire or both.

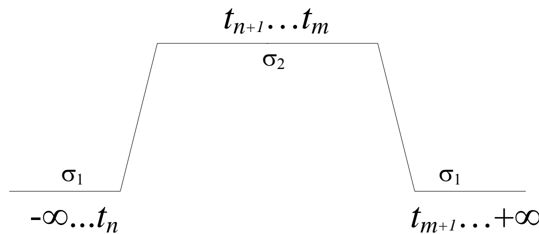


FIGURE 2.11: Transitional Event Form.

2.4.4 Event Form Use

Building on the informal event model, we now discuss how to make use of these canonical event forms to manage event propagation. The notation for events and actions change here to indicate that the event form use is being described as a processing event flow. RITA processes events as shown in Figure 2.12.

Given one or more events E, conditions C, guards G, and actions A, the system has $1 \cdots n$ inputs to a precondition-event matrix evaluating events against specific guards. Guards are a proper subset of conditions ($G \in C$). If the guard evaluates to *True*, the action step comprised of $1 \cdots n$ conditions on that event, $C_1(E) \ldots C_k(E)$, are evaluated. The results of the action step are $0 \cdots n$ outputs, based on the evaluation of condition

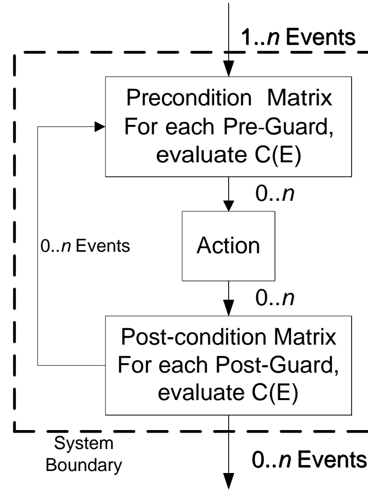


FIGURE 2.12: Event Processing

vectors, to a post-condition event matrix. If the post-condition event guard matrix evaluates to *True*, the output action data – now seen as input event data – is input to another precondition-event guard matrix. In actual use, the definition of guards for precondition and post-condition matrices are compressed in the event evaluation chains so that the trace,

$$\begin{aligned} G_{pre}(E) \rightarrow A \rightarrow G_{post}(E') \rightarrow \\ G_{pre}(E') \rightarrow A \rightarrow G_{post}(E'')\cdots \end{aligned} \tag{2.23}$$

has the post-condition guard being the precondition for the next action step:

$$G_{pre} \rightarrow A \rightarrow G_{post_{pre}} \rightarrow A \rightarrow G_{post_{pre}}\cdots \tag{2.24}$$

As shown in Equation 2.24 each event system can be chained to other event systems.

2.4.5 Formal Event Model

Building on the informal description given in §2.4.3, a definition for the formal event specifications and formal condition-event matrix is given. For each of the three event forms the formal definition of the *precondition* and *post-condition* are formally defined. The action step, as shown in Equation 2.24 above, is not formally defined. The precondition allows an event to initiate an action step (calculation of application data) in the system. A post-condition dampens output from the action step by suppressing output that does not evaluate to *True* in the post-condition. Only post-conditions create new events, or propagation of existing events, in the system. Guards and conditions evaluate specified application data as being time, summation, or delta critical.

Time critical information is data which must reach the computational process at fixed times usually expressed as some delta time. Summation critical information is data that is summed to form a value that triggers an event. Delta critical information is data that must be significantly different from its previous value in order to trigger an event.

2.4.5.1 Spike Event Form

The spike event form has the precondition specification of:

$$\exists E : \forall C_{pre}(E) \rightarrow True \tag{2.25}$$

There exists an event such that a precondition on the event will evaluate to *True*. The spike event form has the post-condition specification of:

$$\begin{aligned} &\exists E : \forall C_{pre}(E) \rightarrow True \ \therefore \\ &\exists A(E) \Rightarrow \exists E' \ni C_{post}(E') \rightarrow True \end{aligned} \tag{2.26}$$

Equation 2.26 asserts the precondition and then states therefore there exits an action for the event implying that there exists some output event E' such that the post-condition C_{post} evaluates to $True$.

2.4.5.2 Set-At Event Form

The set-at event form has the precondition specifications based on an initial condition value:

$$\begin{aligned} &\exists E : \forall C(E)_{(-\infty \ldots t_{n-1})} \equiv False; \\ &\exists A_{t_n}(E) : C_{pre}(E) \rightarrow True \langle \forall \Delta t \rangle \end{aligned} \tag{2.27}$$

There exists an event where, under all conditions, E result in $False$. At some time t_n there exists an action, A, where the precondition for the event results in $True$ over the sequence of all increments of time. Temporally this is:

$$\Box \left[\Box P(E) \Rightarrow \Box P(E') \right] \tag{2.28}$$

This states that there is always an implication of $P(E)$ to $P(E')$ for all predicates of E and E'.

The set-at event is a terminal state transition. This means that once the set-at event form has happened, it is an individuate action and the post-condition is *NULL* as an external source to the entire system is required to change the state once the event occurs to transform the set-at event state.

2.4.5.3 Transitional Event Form

The transitional event form has the precondition specification of:

$$\exists E : C_{pre}(E) \rightarrow True\langle \Delta t_1 \cdots \Delta t_n \rangle \in \forall \Delta t) \tag{2.29}$$

There exists an event where the precondition results in $True$ for a subset of all time. The post-condition is specified as:

$$\begin{aligned} &\exists C_{pre}(E) \rightarrow True \therefore \{\exists A(E) \mid True\langle \forall \Delta t_{t_1 \ldots t_n} \rangle\} \\ &\Rightarrow \exists E' \ni C_{post}(E') \rightarrow True \end{aligned} \tag{2.30}$$

There exists a precondition on an event evaluating to $True$; therefore there exists an action for the event such that it is $True$ comprehended over a sequence of time implying that there exists an output event for the post-condition that evaluates to $True$.

2.4.6 Condition-Event Matrix

The RITA condition-event matrix is comprised of a system of three matrices that are evaluated by row; expressed as:

$$\begin{bmatrix} G_1 \\ \vdots \\ G_n \end{bmatrix} \bullet \begin{bmatrix} C_1(E_1) \cdots C_k(E_1) \\ \vdots \\ C_n(E_n) \cdots C_{n,k}(E_n) \end{bmatrix} \bullet \begin{bmatrix} op_{1,1} \cdots op_{1,k-1} \\ \vdots \\ op_{n,1} \cdots op_{n,k-1} \end{bmatrix} \rightarrow \begin{bmatrix} R_1 \\ \vdots \\ R_n \end{bmatrix} \tag{2.31}$$

Given Equation 2.24, the chaining of systems of condition-event matrices requires that guards be part of preconditions only. As shown in Equation 2.31, a vector, V, is defined as V: $(C_n(E_n) \cdots C_k(E_n))$, a "row." The condition vector is comprehended, post guard evaluation at a single time t, across the condition resultants with the operators $\wedge$ "`and`," $\vee$ "`or`," $\oplus$ "`xor`," and $\neg$ "`not`" expressed as:

$$C_1(E_1)\ op_1\ C_2(E_1)\ op_2\ C_3(E_1) \cdots op_{k-1}\ C_k(E_1) \tag{2.32}$$

The resultant vector, R, if $True$ indicates that the output event E' be propagated to the vector of guards in the next condition-event matrix. The separation of logical control occurs using the G_n, C_n, and op_n components. Internal to C_n components is where data control dispatch calls occur to application code. If a condition position is empty in the matrix the postfix op_n is considered $NULL$. This allows for sparse condition vectors.

2.4.7 Explicit Time

All guard and condition expressions for RITA have time expressed in delta or summation time. These expressions use a restricted subset of C language syntax. Allowed constructs are *IF-THEN*; *IF-THEN-ELSE*; *SWITCH* where alternatives must be inclusive of all values passed by the Guard condition; relational operators for computable numerical values; no pointers or locally declared storage is allowed; and only boolean stack values are allowed as return values and must be of the form *<condition_name>_TRUE* or *<condition_name>_FALSE*. As the *<condition _name>* is known to the system, the postfix *TRUE* or *FALSE* is a name decoration that RITA enforces.

Vector variable data that represent time values relies on three time functions ensuring that the condition-event matrix does not have to rely on user manipulated time functions. The *evaluate_time*() function can be used in the Guard or Condition code. Its function signature is: *int evaluate_time(<vector variable>,[BEFORE | AFTER | NOW], <delta>);*. The *create_time*(), and *get_time*() functions can only be used in code external to the event engine. The event engine system time is uniform throughout evaluation of the guard and its condition vector.

The *evaluate_time*() function compares the vector variable time value with the precision set for the particular system based on the ability of the computer architecture to support high precision time such as the High Precision Event Timer (HPET) hardware timer used in commodity PC systems Windows/Intel chip sets since 2005 [Int04]. In 2013, work on providing nanosecond time resolution with HPET library has been demonstrated [FSH13]. Additionally, precision hardware timers are available from vendors such as Symetricom (http://www.symmetricom.com).

The resultant of the *evaluate_time()* function is *TRUE* or *FALSE*. The special enumeration values of **BEFORE**, **AFTER**, and **NOW** are used to determine the comparison of the system time with the vector variable time. The **NOW** enumeration must have a decimal delta of zero. The delta value is a numerical constant that is a positive floating point value with the precision needed and supported by the system hardware timer.

The *get_time()* function is used to examine the time value set in the *create_time()* function. Assignments to vector variables are allowed. Vector variables can be used in multiple event matrices. Chaining of event matrices ensures that the event value is modified in a deterministic method, without race conditions, and that the tuple of $\{event, value\}$ is propagated through-out the system of condition-event matrices resulting in a final event consumption. If a user causes race conditions by having multiple matrices modify a vector variable in parallel, the pre-processor will alert the user. These

restrictions ensure that each condition-event matrix executes in the same sequence given the same event sequences and event values.

2.4.8 Temporal Logic Model

Having discussed events, latencies, event models, and the condition-event matrix, the final part of RITA theory is a discussion of temporal logic constructs. Quoting Lamport, "What good is temporal logic?" his answer, "Temporal logic is a good method for specifying and reasoning about a concurrent program."[Lam83], is the basis for this section.

As discussed in §2.2.3 (event context) and in §2.4.1 (event propagation), the question of *when* an event occurs is crucial to evaluation of *where* a computational process is mapped to accommodate latency affects for that mapping. RITA theory is developed for concurrent, distributed systems so the two properties that must be satisfied are:

- **Safety** — guarantees that nothing bad happens in the system
- **Liveness** — guarantees that something good eventually happens

Every distributed system has some form of safety and liveness and systems vary with some systems having better safety and liveness than others. Properly functioning systems are either atomically or eventually consistent. Atomic consistency guarantees that operations will occur without interrupt and are expected to function at or near the speed of the computing elements and physical network that comprise the system. This can appear as an "instantaneous" event even though there is some latency (see §2.4.3 for definition of time) so atomicity is not really instantaneous, but is a guarantee of isolation between concurrent processes with "succeed-or-fail" semantics; an event either successfully changes the state of the system or it has no effect at all. Eventually consistent systems are used in distributed computing to achieve a stated high availability. This high availability must be tempered with the proviso that data does not always have a consistent state after operations. Such systems state that if no update is made to a given datum for a period of time (the Δt for the period varies by system) that *eventually* access to that datum will consistently return the last written value [Vog09] and that the system is said to have converged [PST+97]. Eventual consistency can increase the complexity of distributed software applications as eventual consistency is a liveness guarantee and does not make safety guarantees.

2.4.8.1 Deadlock Prevention

RITA uses temporal logic to describe the order in which things must happen rather than the actual times at which they happen. This is why §2.4.7 discusses **BEFORE**, **AFTER**, and **NOW** and only allows *create_time*(), and *get_time*() functions as external input to the condition-event matrix for real system time values.

In a survey of distributed deadlock detection algorithms by Elmagarmid in 1986[Elm86] he lists the three quintessential methods for handling deadlocks: *a*) prevention, *b*) avoidance, and *c*) detection. This older text limits its review to the years 1981-1986 and misses the "Banker's Algorithm" by Dijkstra in 1968[Dij68]. An update to the running time of Dijkstra's algorithm is discussed in Elleithy's work from 2010[LGH10] where Elleithy, et.al. improved Dijkstra's algorithm from $O(dn^2)$ for d resources and n processes to $O\left[\left(n + \sum_{1\leq j\leq d} M_j\right) d\right]$ where M_j is the total number of resource units and, when d and M_j are constants, the running time is $O(n)$. Another survey of deadlock algorithms done in 1989 by Singhal[Sin89] revisits most of the algorithms reviewed by Elmagarmid. In a review of the literature up to the present, only variations on the original methods have been found, and these variations are database oriented which is not germane to this work. Of interest is recent work on deadlock detection for NoC implementations[JH11, ADMX+12, BS13].

RITA does have safety properties where an assertion such as, "The program must respond to an input within ten milliseconds." is expressed in Guards and Conditions. The liveness of RITA is expressed by having event flows that do not deadlock based on Google's Go language methods [SI09]. Go is the implementation language for RITA. RITA also uses features of the Valgrind methods without binary implementation [NS07]. In RITA event flows can terminate, but not deadlock and quiescence in an event flow is not a deadlock as systems can be awaiting input, sometimes indefinitely, but this is not a deadlock as a next action can be taken when input arrives.

2.4.8.2 Temporal Constructs

RITA is a regular language in that there is a series of systems $\mathcal{S}$ connected by events ϵ that can be pumped produce a new sequence of ϵ's. The pumping lemma $\mathcal{R}_\rho$ for RITA is expressed as:

$$
\begin{aligned}
&\texttt{Given RITA is a regular language: } \mathcal{R} \\
&\mathcal{R} : \exists \epsilon_\rho \geq 0 && \texttt{(pumping length)} \\
&s = \epsilon_x \epsilon_y \epsilon_z \ , \ i \geq 0 \ : \ \epsilon_x {\epsilon_y}^i epsilon_z \in \mathcal{R} && \texttt{string } s \\
&|\epsilon_y| > 0 \\
&|\epsilon_x \epsilon_y| \leq \rho
\end{aligned}
\tag{2.33}
$$

The event forms are now rewritten in temporal form for LTL [KM08, Gal08] with $\mathcal{S}_C$ being the condition.
The spike event form:

$$\Box\epsilon \in \mathcal{R}_\rho \rightarrow \Box\mathcal{S}_C \rightarrow \Box\epsilon \tag{2.34}$$

The set-at event form:

$$\Box\epsilon \in \mathcal{R}_\rho \rightarrow \Diamond\epsilon \wedge \mathcal{S}_C\, \mathcal{U}\, \mathcal{S}_{C'} \tag{2.35}$$

Where C holds *until* ($\mathcal{U}$) ϵ where C' has to hold at the current or a future time and C does not have to hold any more. The transitional event form:

$$\Box\epsilon \in \mathcal{R}_\rho \rightarrow \Diamond\epsilon \wedge \mathcal{S}_C \rightarrow \Diamond\epsilon \tag{2.36}$$

Next, a set of Z schemas are now derived showing safety and liveness of message processing in the form used by Duke, et.al.[DHR88]. Initial variables of Tag and $exptag$ are given:

$Tag\ :\ \{0,1\}$
$exptag = 1$

State is a trivial schema containing the above variables and is not shown. The following schemas are needed to build the transaction schema, $TransAck$.

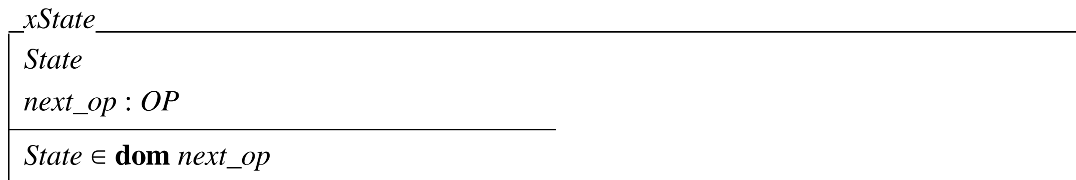


FIGURE 2.13: Extended State Z schema

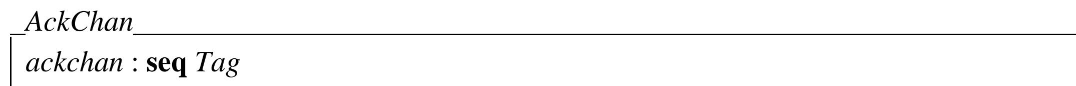


FIGURE 2.14: ACK Channel Z schema

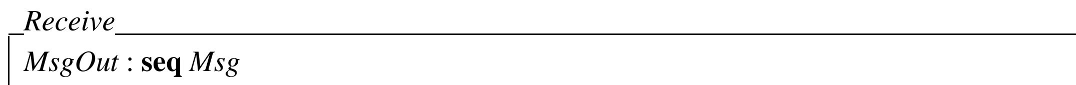


FIGURE 2.15: Receive Z schema

TransAck

$\Delta AckChan$
$\Xi Receive$

$ackchan' = ackchan \frown \langle 1 - exptag \rangle$

FIGURE 2.16: Transaction ACK Z schema

From this, a sequence history from initial states and consecutive states is shown related by $next_op$. Schema $Init$ is an initialization of values of the $State$ schema. OP is an operation on States, incorporating $TransAck$.

$$OP \subseteq State \leftrightarrow State$$

History

$history : \mathbf{seq}_\infty\, xState$

$State_1 \in \{Init\}$
$\forall i : \mathbf{dom}\, history - \{1\} \bullet State_{i\text{-}1} \quad next_op_{i\text{-}1} \quad State_i$

FIGURE 2.17: Operation and History Z schema

Here $\mathbf{seq}_\infty$ is our pumping lemma. The system can never deadlock because the operation $TransAck$ is always active. Using temporal logic notation the $History$ schema can be rewritten as:

History

$history : \mathbf{seq}_\infty\, xState$

$State_1 \in \{Init\}$
$\Box(State\ next_op\ \bigcirc State)$

FIGURE 2.18: Operation and History Z schema

To show correctness, safety is shown as messages not being lost, duplicated, or permuted as shown by the following theorem and lemma

Theorem:

$$History \Rightarrow \Box(MsgsOut \subseteq MsgsIn \subseteq MsgsForTrans)$$

Lemma:

$$History \Rightarrow \Box(buf = \langle\, \rangle \Rightarrow MsgsIn = MsgsOut)$$
$$\Box(buf \neq \langle\, \rangle \wedge MsgsIn = MsgsOut \Rightarrow serial = 1 - exptag)$$
$$\Box(MsgsIn \neq MsgsOut \Rightarrow MsgsIn = MsgsOut \frown (msgof(\mathbf{head}\, buf))$$
$$tagof(\mathbf{head}\, buf) = exptag)$$

This states that $MsgsOut$ is behind $MsgsIn$ by at most the messages in buf. Liveness is shown by guaranteed delivery of messages in the Z-notation and temporal-notation $Progress$ schemas.

Progress

History

$\forall i : \mathbf{dom}\ history \bullet$
$\quad MsgsIn_i \subseteq MsgsIn_{i+1} \land MsgsOut_i \subseteq MsgsOut_{i+1}$
$\forall i : \mathbf{dom}\ history \bullet$
$\quad MsgsOut_i \subset MsgsForTrans \Rightarrow \exists j : \mathbf{dom}\ history \bullet$
$\qquad j \geqslant i \land MsgsOut_j \subset MsgsOut_{j+1}$

FIGURE 2.19: Liveness of messages Z schema

Progress

History

$\Box(MsgsIn \subseteq \bigcirc MsgsIn \land MsgsOut \subseteq \bigcirc MsgsOut)$
$\Box(MsgsOut \subset MsgsForTrans \Rightarrow \Diamond(MsgsOut \subset \bigcirc MsgsOut))$

FIGURE 2.20: Liveness temporal Z schema

Chapter 3

Algorithm to Computational Architecture Mapping

3.1 Overview

ARCHITECTURES of computing devices have been the boon and bane of every compiler ever written. Instruction set architectures, memory access channels, and the attendant I/O, cache, and bus circuitry have driven development of compilers to either take advantage of the physical architecture or to work around it. When mapping algorithms onto heterogeneous architectures, having knowledge of the state machine for the architecture that executes the software instructions is crucial. With some CISC systems there are uninterruptible instructions that read and write memory with one instruction and when used with an operand type that is accessible in a single memory operation, each instruction provides an atomic read-modify-write sequence. Some RISC systems have no single instruction that performs such so an atomic read-modify-write operation is only possible through a sequence of instructions that have load-locked, store-conditional instructions. These types of instructional differences require exacting knowledge of the hardware to map software modules to computational elements.

With increasing clock frequency and decreasing silicon feature sizes (22nm is now common, with predictions for 7nm[1]), modern CPU designs have upper performance limits for single processor systems. Limits are caused by a "**memory wall**," where memory latency increases and bandwidth is insufficient causing processors to lack enough instructions and data to continue computation seamlessly causing high cache misses

[1] Brian Krzanich, Intel CEO from 2013 Intel Developer Forum keynote presentation. See page 53 for IBM's prediction.

and processors will effectively be always stalled waiting on memory; A "**power wall**," where the trend is consuming exponentially increasing power with a factorial increase in clock frequency leading to smaller feature sizes which pose material physics difficulties which effect manufacturing, system design, and deployment; And the instruction level parallelism wall, "**ILP wall**," where it is difficult finding sufficient parallelism in a single instruction stream to keep a high-performance single-core processor busy [HP11, ABD+09]. A solution to these "walls" is increasing the number of processors to achieve a high computational throughput needed for physical simulation (especially simulations for fluid dynamics), multimedia content creation, and financial modeling. This is a common architecture employed in computer clusters in the TOP500[2] supercomputer list. Mapping algorithms across such computing elements requires knowing which computing pattern applies to a computing element and an ontological knowledge of the parallel computing environment supporting that environment. This ontological knowledge allows best resource use and adaptability to changes in the environment.

3.2 Partitioning Patterns

Computing Elements

To map algorithms to appropriate machine architectures — which are referred to as computing elements in this thesis as not all computation is done by a CPU — requires an understanding of patterns of computation that can recognized, or indicated by source language pragmas, so proper executables can be created and loaded on the correct computing elements and this requires an ontology. This thesis uses the work by Spyns [SMJ02] and reduces it to an operational definition of ontology:

Ontology Definition

> An computing ontology consists of relatively generic knowledge of computer-based resources that represent agreed upon domain semantics that can be reused by different kinds of applications or tasks.

3.2.1 Berkeley "Dwarfs"

In a report from Berkeley[ABC+06], a detailed listing of thirteen classes of algorithms — also known as "dwarfs" — are listed that provide an accepted set of numerical methods for scientific computing. In Table 3.1 the original seven classes are listed. As a pun, Colella named these seven numerical methods as "dwarfs" from Walt Disney's, "Snow White and the Seven Dwarfs." While a colorful story, the true meaning that Colella had in mind was that truly useful computational patterns are not invented but "mined" from successful software applications as shown in Figure 3.1. Colella described these in his remarkably well know but unpublished presentation, "Defining Software Requirements for

[2] `http://www.top500.org/`

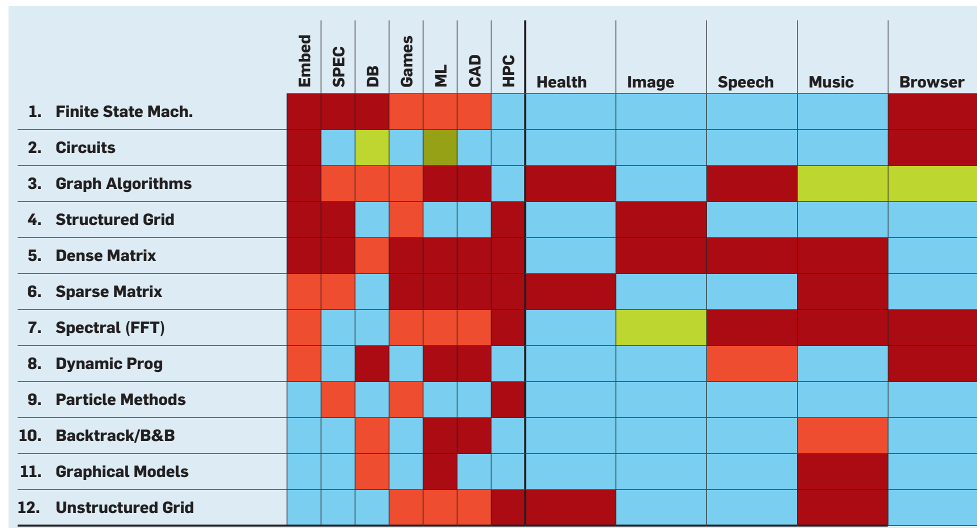
Scientific Computing."[3] Colella developed this list while at Lawrence Berkeley National Laboratory in 2004. Membership for algorithms in each class is defined by similarity in computation and data movement. The dwarfs are specified at an abstraction level allowing reasoning about their behavior across a broad range of applications. Programs that are members of a particular class can have different implementations. The underlying numerical methods may change over time with each dwarf defining the underlying patterns that will persist through generations of changes.

Dwarf	**Description**
1. Dense Linear Algebra (e.g., BLAS, ScaLAPACK, or Mathworks MATLAB)	Data are dense matrices or vectors. (BLAS Level 1 = vector-vector; Level 2 = matrix-vector; and Level 3 = matrix-matrix.) Generally, such applications use unit-stride memory accesses to read data from rows, and strided accesses to read data from columns
2. Sparse Linear Algebra (e.g., SpMV, OSKI, or SuperLU)	Data sets include many zero values. Data is usually stored in compressed matrices to reduce the storage and bandwidth requirements to access all of the nonzero values. One example is block compressed sparse row (BCSR). Because of the compressed formats, data is generally accessed with indexed loads and stores
3. Spectral Methods (e.g., FFT)	Data are in the frequency domain, as opposed to time or spatial domains. Typically, spectral methods use multiple butterfly stages, which combine multiply-add operations and a specific pattern of data permutation, with all-to-all communication for some stages and strictly local for others
4. N-Body Methods (e.g., Barnes-Hut, Fast Multipole Method)	Represented by a regular grid; points on grid are conceptually updated together. It has high spatial locality. Updates may be in place or between 2 versions of the grid. The grid may be subdivided into finer grids in areas of interest ("Adaptive Mesh Refinement"); and the transition between granularities may happen dynamically
5. Structured Grids (e.g., Cactus or Lattice- Boltzmann Magnetohydrodynamics)	Represented by a regular grid; points on grid are conceptually updated together. It has high spatial locality. Updates may be in place or between 2 versions of the grid. The grid may be subdivided into finer grids in areas of interest ("Adaptive Mesh Refinement"); and the transition between granularities may happen dynamically
6. Unstructured Grids (e.g., ABAQUS or FIDAP)	An irregular grid where data locations are selected, usually by underlying characteristics of the application. Data point location and connectivity of neighboring points must be explicit. The points on the grid are conceptually updated together. Updates typically involve multiple levels of memory reference indirection, as an update to any point requires first determining a list of neighboring points, and then loading values from those

[3] Presented at the 2005 Salishan Conference on High Speed Computing, Salishan Lodge, Oregon, USA

Dwarf	Description
7. Map Reduce (Monte Carlo)	This dwarf was originally called "Monte Carlo", after the technique of using statistical methods based on repeated random trials. The patterns defined by the programming model MapReduce are a more general version of the same idea: repeated independent execution of a function, with results aggregated at the end. Nearly no communication is required between processes.

TABLE 3.1: Original Seven Dwarfs



[ABD+09]

FIGURE 3.1: Dwarf Application Areas
Prevalence for pattern in that application shown by color: Red=High; Orange=Moderate; Green=Low; Blue=Rare.

The Berkeley study went on to define six more dwarfs as shown in Table 3.2 based on the following criteria:

- Combinational Logic generally involves performing simple operations on very large amounts of data often exploiting bit-level parallelism. For example, computing Cyclic Redundancy Codes (CRC) is critical to ensure integrity and RSA encryption for data security.
- Graph Traversal - applications must traverse a number of objects and examine characteristics of those objects such as would be used for search. It typically involves indirect table look-ups and little computation.
- Graphical Models - applications involve graphs that represent random variables as nodes and conditional dependencies as edges. Examples include Bayesian networks and Hidden Markov Models.
- Finite State Machines - represent an interconnected set of states, such as would be used for parsing. Some state machines can decompose into multiple simultaneously active state machines that can act in parallel.

- Machine Learning – statistical machine learning to make sense from the vast amounts of data now available due to faster computers, larger disks, and the use of the Internet to connect them all together.
- Database Software – Using the metrics of MinuteSort, Hashing, MapReduce, and BLAST (Basic Local Alignment Search Tool)[AGM+90]

New Dwarf	Description
8. Combinational Logic (e.g., encryption)	Functions that are implemented with logical functions and stored state
9. Graph traversal (e.g., Quicksort)	Visits many nodes in a graph by following successive edges. These applications typically involve many levels of indirection, and a relatively small amount of computation
10. Dynamic Programming	Computes a solution by solving simpler overlapping sub-problems. Particularly useful in optimization problems with a large set of feasible solutions
11. Backtrack and Branch+Bound	Finds an optimal solution by recursively dividing the feasible region into sub-domains, and then pruning sub-problems that are suboptimal
12. Construct Graphical Models	Constructs graphs that represent random variables as nodes and conditional dependencies as edges. Examples include Bayesian networks and Hidden Markov Models
13. Finite State Machine	A system whose behavior is defined by states, transitions defined by inputs and the current state, and events associated with transitions or states

[ABC+06]

TABLE 3.2: Additional Dwarfs

The “dwarfs” are used to categorize specific pattern computing codes. They provide a framework for reasoning about these codes. The thirteen dwarfs focus on scientific computing and are considered to be single function application kernels, in fact replacing many system libraries with specialized system-specific software used with auto-tuning for a specialized system. Dwarfs are best understood as computational patterns providing the computational interior of the structural patterns discussed earlier. By analogy, the structural patterns describe a “factory” physical structure and general workflow. Application programmers spend extremely long hours modifying software enhancing performance for specific computer architectural features. As these features are different between platforms, software tuned for one platform usually has performance problems when ported to another platform; Problems such as a chip’s ISA, 32 or 64 bit instruction word, and the size and availability of L1, L2, and L3 cache. Auto-tuning software helps programmers automate labor-intensive and error-prone process of tuning and porting a specific software application “factory.” Domain-specific auto-tuners such as ATLAS[WD98] for dense linear algebra, OSKI[VDY05] for sparse linear algebra, FFTW[FJ05] and SPIRAL[XJJP01] for signal processing have been successful in producing highly-optimized architecture-specific codes for these classes of dwarf patterns.

While these patterns can be identified, they are not easily moved from between physical systems and thus dynamic allocation is not advantageous. They are discussed here to identify that some specific algorithms can not be moved to a new computing element via a general mapping process. They are, in fact, "islands" that are "bridged" to in federated computing applications using network communications with other mapped algorithms that build a larger computational mapped architecture. Movement of data from these "islands" is the primary activity with these systems being data stores for data query or query-response processed data retrieval.

3.2.2 Work-flow Patterns

Moving on from "islands" of software to more transportable software modules, a highly cited work providing a taxonomy of work-flow patterns was created by W.M.P. van der Aalst in 2003 [vdAtHKB03] and was extended and revised in work by Russell in 2006 [RHvdAM06]. Russell's and van der Aalst's work adds to the ontological base of "dwarfs" by describing operational perspectives of modules so code can be considered as operation types, and these types can be used to map to computational elements.

- The **control-flow perspective**, or **process perspective**, describes activities and their execution ordering through different constructors, which permit flow of execution control, e.g. sequence, choice, parallelism and join synchronization. Activities in elementary form are atomic units of work, and in compound form modularize an execution order of a set of activities
- The **data perspective** layers business and processing data on the control perspective. Business documents and other objects which flow between activities, and local variables of the work-flow, qualify in effect pre- and post-conditions of activity execution.
- The **resource perspective** provides an organizational structure anchor to the work-flow in the form of human and device roles responsible for executing activities
- The **operational perspective** describes the elementary actions executed by activities, where the actions map into underlying applications. Typically, (references to) business and work-flow data are passed into and out of applications through activity-to-application interfaces, allowing manipulation of the data within applications.

By understanding control flow dependencies between activities in a workflow decomposition into patterns follows. From Russell's revision, he extended van der Aalst's twenty patterns to forty-three patterns codifying them into workflow control patterns

(WCP) ordering them from simplest to most complex as shown in Table 3.3. In Table 3.4 the novel decision point mapping created for this work is shown and used in §3.4.

Number	**Pattern**	**Description**
WCP-1	Sequence	An activity in a workflow process is enabled after the completion of a preceding activity in the same process
WCP-2	Parallel Split	The divergence of a branch into two or more parallel branches each of which execute concurrently
WCP-3	Synchronization	The convergence of two or more branches into a single subsequent branch such that the thread of control is passed to the subsequent branch when all input branches have been enabled
WCP-4	Exclusive Choice	The divergence of a branch into two or more branches. When the incoming branch is enabled, the thread of control is immediately passed to precisely one of the outgoing branches based on the outcome of a logical expression associated with the branch
WCP-5	Simple Merge	The convergence of two or more branches into a single subsequent branch. Each enablement of an incoming branch results in the thread of control being passed to the subsequent branch
WCP-6	Multi-Choice	The divergence of a branch into two or more branches. When the incoming branch is enabled, the thread of control is passed to one or more of the outgoing branches based on the outcome of distinct logical expressions associated with each of the branches
WCP-7	Structured Synchronizing Merge	The convergence of two or more branches (which diverged earlier in the process at a uniquely identifiable point) into a single subsequent branch. The thread of control is passed to the subsequent branch when each active incoming branch has been enabled
WCP-8	Multi-Merge	The convergence of two or more branches into a single subsequent branch. Each enablement of an incoming branch results in the thread of control being passed to the subsequent branch
WCP-9	Structured Discriminator	The convergence of two or more branches into a single subsequent branch following a corresponding divergence earlier in the process model. The thread of control is passed to the subsequent branch when the first incoming branch has been enabled. Subsequent enablements of incoming branches do not result in the thread of control being passed on. The discriminator construct resets when all incoming branches have been enabled
WCP-10	Arbitrary Cycles	The ability to represent cycles in a process model that have more than one entry or exit point. The pattern provides a means of supporting repetition in a process model in an unstructured way without the need for specific looping operators or restrictions on the overall format of the process model
WCP-11	Implicit Termination	A given process (or sub-process) instance should terminate when there are no remaining work items that are able to be done either now or at any time in the future

Number	Pattern	Description
WCP-12	Multiple Instances without Synchronization	Within a given process instance, multiple instances of an activity can be created. These instances are independent of each other and run concurrently. There is no requirement to synchronize them upon completion. It is particularly suited to situations where the number of individual activities required is known before the spawning action commences, the activities can execute independently of each other and no subsequent synchronization is required
WCP-13	Multiple Instances with *à priori* Design-Time Knowledge	Within a given process instance, multiple instances of an activity can be created. The required number of instances is known at design time. These instances are independent of each other and run concurrently. It is necessary to synchronize the activity instances at completion before any subsequent activities can be triggered
WCP-14	Multiple Instances with *a priori* Run-Time Knowledge	Within a given process instance, multiple instances of an activity can be created. The required number of instances may depend on a number of run-time factors, including state data, resource availability and inter-process communications, but is known before the activity instances must be created. Once initiated, these instances are independent of each other and run concurrently. It is necessary to synchronize the instances at completion before any subsequent activities can be triggered
WCP-15	Multiple instances without *à priori* run-time knowledge	Within a given process instance, multiple instances of an activity can be created. The required number of instances may depend on a number of run-time factors, including state data, resource availability and inter-process communications and is not known until the final instance has completed. Once initiated, these instances are independent of each other and run concurrently. At any time, whilst instances are running, it is possible for additional instances to be initiated. It is necessary to synchronize the instances at completion before any subsequent activities can be triggered
WCP-16	Deferred Choice	A point in a workflow process where one of several branches is chosen based on interaction with the operating environment. Prior to the decision, all branches present possible future courses of execution. The decision is made by initiating the first activity in one of the branches; that is, there is no explicit choice but rather a race between different branches. After the decision is made, execution alternatives in branches other than the one selected become nullified
WCP-17	Interleaved Parallel Routing	A set of activities has a partial ordering defining the requirements with respect to the order in which they must be executed. Each activity in the set must be executed once and they can be completed in any order that accords with the partial order. However, as an additional requirement, no two activities can be executed at the same time (i.e. no two activities can be active for the same process instance at the same time)
WCP-18	Milestone	An activity is only enabled when the process instance is in a specific state; typically a parallel branch. The state is assumed to be a specific execution point (milestone) in the process model. When this execution point is reached the nominated activity can be enabled. If the process instance has progressed beyond this state, then the activity cannot be enabled now or at any future time due to its deadline expiration. Note that the execution does not influence the state itself, unlike normal control-flow dependencies, it is a test rather than a trigger

Number	Pattern	Description
WCP-19	Cancel Activity	An enabled activity is withdrawn prior to it commencing execution. If the activity has started, it is disabled and, where possible, the currently running instance is halted and removed. The motivation for this pattern provides the ability to withdraw an activity which has been enabled. This ensures that it will not commence execution such as queue element modification
WCP-20	Cancel Case	A complete process instance is removed. This includes currently executing activities, those which may execute at some future time and all sub-processes. The process instance is recorded as having completed unsuccessfully. This pattern provides a means of halting a specified process instance and withdrawing any activities associated with it
WCP-21	Structured Loop	The ability to execute an activity or sub-process repeatedly. The loop has either a pre- or post-test condition associated with it that is evaluated to determine loop continuation. The looping structure has a single entry and exit point
WCP-22	Recursion	The ability of an activity to invoke itself during its execution or an ancestor in a call chain structure with which it is associated
WCP-23	Transient Trigger	The ability for an activity to be triggered by a signal from another part of the process or from the external environment. These triggers are transient in nature and are lost if not acted on immediately by the receiving activity
WCP-24	Persistent Trigger	The ability for an activity to be triggered by a signal from another part of the process or from the external environment. These triggers are persistent in form and are retained by the workflow until they can be acted on by the receiving activity
WCP-25	Cancel Region	The ability to disable a set of activities in a process instance. If any of the activities are already executing, then they are withdrawn. The activities need not be a connected subset of the overall process model
WCP-26	Cancel Multiple Instance Activity	Within a given process instance, multiple instances of an activity can be created. The required number of instances is known at design time. These instances are independent of each other and run concurrently. At any time, the multiple instance activity can be canceled and any instances which have not completed are withdrawn. This does not affect activity instances that have already completed
WCP-27	Complete Multiple Instance Activity	In a process instance, multiple activity instances can be created. The required number of instances is known at design time. These instances are independent of each other and run concurrently. It is necessary to synchronize the instances at completion before any subsequent activities can be triggered. During the course of execution, it is possible that an activity needs to be forcibly completed so that any remaining instances are withdrawn and the thread of control is passed to subsequent activities
WCP-28	Blocking Discriminator	The convergence of two or more branches into a single subsequent branch following one or more corresponding divergences earlier in the process model. The thread of control is passed to the subsequent branch when the first active incoming branch has been enabled. The discriminator construct resets when all active incoming branches have been enabled once for the same process instance. Subsequent enablements of incoming branches are blocked until the discriminator has reset

Number	Pattern	Description
WCP-29	Canceling Discriminator	The convergence of two or more branches into a single subsequent branch following one or more corresponding divergences earlier in the process model. The thread of control is passed to the subsequent branch when the first active incoming branch has been enabled. Triggering the discriminator also cancels the execution of all of the other incoming branches and resets the construct
WCP-30	Structured Partial Join	The convergence of M branches into a single subsequent branch following a corresponding divergence earlier in the process model. The thread of control is passed to the subsequent branch when N of the incoming branches have been enabled. Subsequent enablements of incoming branches do not result in the thread of control being passed on. The join construct resets when all active incoming branches have been enabled
WCP-31	Blocking Partial Join	The convergence of two or more branches into a single subsequent branch following one or more corresponding divergences earlier in the process model. The thread of control is passed to the subsequent branch when N of the incoming branches have been enabled. The join construct resets when all active incoming branches have been enabled once for the same process instance. Subsequent enablements of incoming branches are blocked until the join has reset
WCP-32	Canceling Partial Join	The convergence of two or more branches into a single subsequent branch following one or more corresponding divergences earlier in the process model. The thread of control is passed to the subsequent branch when N of the incoming branches have been enabled. Triggering the join also cancels the execution of all of the other incoming branches and resets the construct
WCP-33	Generalized AND-Join	The convergence of two or more branches into a single subsequent branch such that the thread of control is passed to the subsequent branch when all input branches have been enabled. Additional triggers received on one or more branches between firings of the join persist and are retained for future firings
WCP-34	Static Partial Join for Multiple Instances	Within a given process instance, multiple concurrent instances of an activity can be created. The required number of instances is known when the first activity instance commences. Once N of the activity instances have completed, the next activity in the process is triggered. Subsequent completions of the remaining $M - N$ instances are inconsequential
WCP-35	Canceling Partial Join for Multiple Instances	Within a given process instance, multiple concurrent instances of an activity can be created. The required number of instances is known when the first activity instance commences. Once N of the activity instances have completed, the next activity in the process is triggered and the remaining $M - N$ instances are canceled

Number	**Pattern**	**Description**
WCP-36	Dynamic Partial Join for Multiple Instances	Within a given process instance, multiple concurrent instances of an activity can be created. The required number of instances may depend on a number of run-time factors: including state data, resource availability, and inter-process communications and is not known until the final instance has completed. At any time, while instances are running, it is possible for additional instances to be initiated providing the ability to do so has not been disabled. A completion condition is specified which is evaluated each time an instance of the activity completes. Once the completion condition evaluates to $TRUE$, the next activity in the process is triggered. Subsequent completions of the remaining activity instances are inconsequential and no new instances can be created
WCP-37	Acyclic Synchronizing Merge	Two or more branches, which diverged earlier in the process, converge into a single subsequent branch. The thread of control is passed to the subsequent branch when each active incoming branch has been enabled. Determination of how many branches require synchronization is based on the information locally available to the merge construct. This may be communicated directly to the merge by the preceding diverging construct or alternatively it can be determined on the basis of local data such as the threads of control arriving at the merge
WCP-38	General Synchronizing Merge	The convergence of two or more branches which diverged earlier in the process into a single subsequent branch. The thread of control is passed to the subsequent branch when each active incoming branch has been enabled or it is not possible that the branch will be enabled at any future time
WCP-39	Critical Section	Two or more connected subgraphs of a process model are identified as *critical sections*. At run-time for a given process instance, only activities in one of these *critical sections* can be active at any given time. Once execution of the activities in one *critical section* commences, it must complete before another *critical section* can commence
WCP-40	Interleaved Routing	Each member of a set of activities must be executed once. They can be executed in any order but no two activities can be executed at the same time (i.e. no two activities can be active for the same process instance at the same time). Once all of the activities have completed, the next activity in the process can be initiated
WCP-41	Thread Merge	At a given point in a process, a nominated number of execution threads in a single branch of the same process instance are merged together into a single thread of execution
WCP-42	Thread Split	At a given point in a process, a nominated number of execution threads can be initiated in a single branch of the same process instance
WCP-43	Explicit Termination	A given process (or sub-process) instance should terminate when it reaches a nominated state. Typically this is denoted by a specific end node. When this end node is reached, any remaining work in the process instance is canceled and the overall process instance is recorded as having completed successfully

Table 3.3: Workflow Control Patterns

Decision-Point	**WCP Patterns**
DP-1 Sequential	WCP-1
DP-2 Branching	WCP-2, WCP-4, WCP-6, WCP-18, WCP-42
DP-3 Merge & Converge	WCP-3, WCP-5, WCP-7, WCP-8, WCP-9, WCP-28, WCP-29, WCP-41
DP-4 Termination & Cancellation	WCP-11, WCP-19, WCP-20, WCP-25, WCP-26, WCP-43
DP-5 *à priori* Run-Time Knowledge	WCP-12, WCP-13, WCP-14, WCP-15
DP-6 Arbitrary or External actions	WCP-10, WCP-16
DP-7 Interleaved & Transient	WCP-17, WCP-23, WCP-24
DP-8 Loop & Recursion	WCP-21, WCP-22
DP-9 Multiple Instance	WCP-27
DP-10 Joins	WCP-30, WCP-31, WCP-32, WCP-33, WCP-34, WCP-35, WCP-36
DP-11 Synchronizing Merge	WCP-37, WCP-38
DP-12 Critical Section	WCP-39
DP-13 One-time Execution	WCP-40

TABLE 3.4: WCP to Decision-Point Matching

3.3 Cloud Computing Ontology

HAVING established the ability to identify work flow partitioning for algorithms in §3.2, this section focuses on the ontological representation. Cloud computing, as a paradigm, is fairly recent and as such the ontology for it is usually represented as a taxonomy with a discussion of product offerings from vendor companies; Hilley [Hil09] does this and Höfer et.al. treats the taxonomy in hierarchical tree form [HK11] and compares services provided by top tier companies offering cloud environments. This taxonomic method is even codified by the United States National Institute of Standards and Technology [LTM+11]. The literature provides a taxonomic method describing services in this new paradigm and presents it as a Service Oriented Architecture (SOA) so that application designs using these services mirror actual business activities in the enterprise business processes.[4] These taxonomies will not be the focus of this thesis. From these taxonomies, work has progressed on cloud computing ontologies and these ontologies will be used in this work to focus on allocation of algorithms to computing elements.

Use of computing patterns for algorithms as described in §3.2.1 and §3.2.2 need an ontology to map the available computing elements and services from the cloud environment. The ontology of cloud computing work by Roman, et.al. in 2005[RKL+05] focused on web service modeling ontologies which referenced the initial work in 2004 that resulted in OWL[5] [DS04], now OWL 2. Cloud computing ontology has been attempted since 2008 by Youseff, et.al.[YBDS08] where the authors had a five layer ontology of applications, software environments, software infrastructure, software kernel, and hardware

[4]Open Group SOA definition http://www.opengroup.org/soa/source-book/togaf/soadef.htm.
[5]Web Ontology Language. OWL is not an acronym, but a convention of the World Wide Web Consortium (W3C).

with each layer providing one or more services. An interesting work, expanding on the "*X-as-a-Service*" naming convention, is the paper by Flahive, et.al.[FTR13] where the authors introduce Ontology as a Service (OaaS) for each cloud provider with demonstration of an example extracting and merging the sub-ontology for multiple cloud providers to allow linking and traversing these vendor cloud environments. At an abstract level this does provide a method to unify disparate ontologies, but for this work it does not yield the needed algorithm decomposition for computing element mapping.

In order to be effective, any mapping method must be able to discover and select candidate cloud systems for mapping. Once discovered, a catalog of available computing elements and services must be maintained in real-time so that assignments are not made on aging catalog data. In an often repeated topic, in multiple papers published by Kang and Sim on cloud service matching and their tool "Cloudle," their definitive work [KS11] proposed a cloud ontology to semantically define the relationship among different cloud services. While this is an interesting method of applying numeric values to intersections of objects, it misses the point that requirements can't be met by evaluation of point values to dissimilar elements. That is to say, a comparison between Windows 7 and Linux v3.17 object properties in an ontology is acceptable but, in actual use, they are not comparable as interchangeable application platforms as each requires specific and unique coding and infrastructure to execute applications under these operating systems and thus the equivalence is inherently false.

Intersections between data object properties can be computed. This is extended to the idea that *concepts* can be similarity computed among objects referred to differently but having the same meaning. Thus the *Similarity* of concept (Sim_{con}) can be computed by $Sim_{\text{con}}(a,b) = \frac{|Super(A) \cap Super(B)|}{|Super(A)|}$; where A and B are the most specific concepts of individuals a and b with $Super(A)$ and $Super(B)$ being the sets of all reachable super concepts from the concepts A and B. The processing steps are: 1. Consider the two individuals for which the concept similarity is to be calculated, a and b. 2. Count all the reachable super concepts from each of the specific concepts, A and B, of the individuals a and b respectively. 3. Determine the commonly reachable super concepts of A and B, the intersection. 4. Divide the result of the intersection by the result of super concepts for A (for an individual a). Repeat for individual b.

For comparable items like storage, CPU speed, memory, and network type this is a valid comparison of similarities as these are components that are agnostic to an operating system where operating systems have to conform to the device interface (a.k.a drivers). In §3.3.4 this work will contribute a novel and better method to ontologically compare cloud systems with a weighted matching method to determine mapping of algorithms to computing elements in a cloud environment. To achieve this an agent or some form of a

Weighted Match

data store must be available. An agent providing information as a "web crawler" service using the user-agent field of an HTTP request in accordance with RFC 723x[6] provides this in a usable manner. Sim does exactly that in his *Agent-Based Cloud Computing* paper [Sim12]. In the paper by Ma, et.al. [MJL11] the authors put forward a VM based ontology controlled job allocation algorithm for a resource management system in cloud system using a cloud ontology based on resource information and Service Level Agreements (SLA). An analog for agents done by Sim, and the allocation method used in Ma's work, will be used in this thesis for resource management.

SLA

3.3.1 Current Cloud System Allocations

The current cloud environment vendors provide calculators to select configurations of their systems. Amazon[7], Google[8], Microsoft[9], and others provide this to facilitate selection of services based on leasing cost calculations. I have used these calculators to estimate computing requirements for algorithm allocation manually and it was obvious that if a particular computing application did not match the proffered configuration, it would not be easy to use generalized computing requirements to find a good fit to the different service offerings. These calculators are designed to select service configurations for each vendor and thus are not able to select common services making generalized comparison very difficult without extensive customer knowledge of the actual hardware, network, and VM configurations – with all control options of VM time and resource control – offered by the vendor. Although there have been taxonomies to classify Cloud services across IaaS, PaaS, and SaaS layers as described at the beginning of §3.3, these have failed to capture low-level configuration information needed for computing elements that make up clusters of systems that offer services, and their dependencies, across cloud service layers.

3.3.2 Representation of Allocations in a Semantic Web

Circa 2001 [BLHL01] the term *Semantic Web* was being used to describe the relationship of objects for Internet connected systems; *à la* Cloud semantics today. The result of reading through many of the descriptive logics mentioned by Baader [BHS09] and Kontoudis [KF14] and their references, has led to a short list of schema languages that have sufficient descriptive ability to allow algorithm allocation to computing elements based on declared computing element properties and relationships giving a framework for

[6] See HTTP protocol documents at http://www.w3.org/Protocols/.
[7] https://aws.amazon.com/blogs/aws/estimate-your-c/
[8] https://cloud.google.com/products/calculator/
[9] https://azure.microsoft.com/en-us/pricing/calculator/

representing cloud resource information. The Network Markup Language (NML) from the Open Grid Foundation [vdHeDŁZ13], the Virtual eXecution Description Language (VXDL) [KPCa09], and the Infrastructure and Network Description Language (INDL) [GvdHG+13] are description logics that share a notational similarity to description logic for networking and computing element identification. NML is used to express device and device linkage details. An example of use is from the NML OGF documentation:

ExampleNMLCode.xml

```
1  <nml:Topology xmlns:nml="http://schemas.ogf.org/nml/2013/05/base#"
2   id="urn:ogf:network:example.net:2013:org"
3   version="20130529T121112Z">
4
5    <nml:Node id="urn:ogf:network:example.net:2013:nodeA">
6      <nml:name>Node_A</nml:name>
7
8      <!-- Outbound Ports -->
9      <nml:Location id="urn:ogf:network:example.net:2013:redcity"/>
10     <nml:Relation type="http://schemas.ogf.org/nml/2013/05/base#hasOutboundPort">
11       <nml:Port id="urn:ogf:network:example.net:2013:port_X:out"/>
12       <nml:Port id="urn:ogf:network:example.net:2013:port_Y:out"/>
13     </nml:Relation>
14
15     <!-- Inbound Ports -->
16     <nml:Relation type="http://schemas.ogf.org/nml/2013/05/base#hasInboundPort">
17       <nml:Port id="urn:ogf:network:example.net:2013:port_X:in"/>
18       <nml:Port id="urn:ogf:network:example.net:2013:port_Y:in"/>
19     </nml:Relation>
20
21     <nml:Relation type="http://schemas.ogf.org/nml/2013/05/base#hasService">
22       <nml:SwitchingService id="urn:ogf:network:example.net:2013:nodeA:switchingService"/>
23     </nml:Relation>
24
25   </nml:Node>
26
27   <nml:Port id="urn:ogf:network:example.net:2013:port_X.1501:in">
28   <nml:Label labeltype="http://schemas.ogf.org/nml/2013/05/ethernet#vlan">1501</nml:Label>
29   </nml:Port>
30
31   <nml:SwitchingService id="urn:ogf:network:example.net:2013:nodeA:switchingService">
32     <nml:Relation type="http://schemas.ogf.org/nml/2013/05/base#hasInboundPort">
33       <nml:Port id="urn:ogf:network:example.net:2013:nodeA:port_X:in" />
34       <nml:Port id="urn:ogf:network:example.net:2013:nodeA:port_Y:in" />
35     </nml:Relation>
36     <nml:Relation type="http://schemas.ogf.org/nml/2013/05/base#hasOutboundPort">
37       <nml:Port id="urn:ogf:network:example.net:2013:nodeA:port_X:out"/>
38       <nml:Port id="urn:ogf:network:example.net:2013:nodeA:port_Y:out"/>
39     </nml:Relation>
40     <nml:Relation type="http://schemas.ogf.org/nml/2013/05/base#providesLink">
41       <nml:Link id="urn:ogf:network:example.net:2013:LinkA:XY"/>
42     </nml:Relation>
43   </nml:SwitchingService>
44
45 </nml:Topology>
```

ExampleNMLCode.xml

Unfortunately, the operational data for VXDL is not fully available as the technology has been patented by LYaTiss[10] and the *Institut National de Recherche en Informatique et en Automatique*[11] [Kos12] so all tools and implementation data has been removed from public sources. This work will not recreate the VXDL ecosystem, but will use it to illustrate the necessary computing element descriptions — and in concert with NML; the network descriptions — needed to perform mapping. The EBNF[12] of the VXDL notation is used in this work as listed in Appendix E.

In an interesting separate work developed at Google by Chris Bunch a domain specific language named *Neptune* [Bun13] was developed as part of Bunch's Ph.D. program. Bunch was the student lead on the AppScale project which used *Neptune* as its description language for AppScale, an open source Google App Engine compatible hosting solution. Bunch was supervised at Google by Professor Chandra Krintz, Bunch's Ph.D. adviser. *Neptune* maps homogeneous cloud services across AppScale instances. The AppScale PaaS automatically deploys and scales unmodified Google App Engine applications over popular public and private cloud systems and on-premises clusters providing an optimizer and resource allocation mechanism but lacks heterogeneous architecture mapping. This distinction will be revisited in §3.3.3.1.

NML and VXDL provide a method to connect compute elements by allowing description of the requisite infrastructure showing communication linkage for the relationships as shown in Figure 3.2 which shows the ontological relationship for computing elements. In Figure 3.3 the decision point ontological relationship for the items in Table 3.4 represented as classes for the workflow control Patterns (WCP) shown in Table 3.3. In Figure 3.3 note that four classes are not shown: *a*) DP-4: Algorithm exit. *b*) DP-6: External system input or control and interrupt *c*) DP-10 and DP-11: These are WCPs for stack or queue unwinding which do not have an ontological class associated with them

In each of these four classes, the ontology does not represent external actions, or flow of control actions. The OWL syntax ontology that will be used for a *ComputingElement* is shown in Appendix D. In Figure 3.4 a reduced member cloud ontology is shown. The members not shown are network and storage systems as these can be considered simple data value properties; that is, a network is a polynomial resulting in a numeric value for capacity and throughput, *ipso facto* the "speed" of the network; and so is storage as it is a polynomial with a resultant numeric value for capacity, access time, and throughput for the same semantic value of "speed." These are considered qualifiers for ubiquitous elements of a computing configuration.

[10] `http://www.lyatiss.com` now `http://www.cloudweaver.com/`
[11] IN-RIA, `http://www.inria.fr/`
[12] Extended Backus–Naur Form

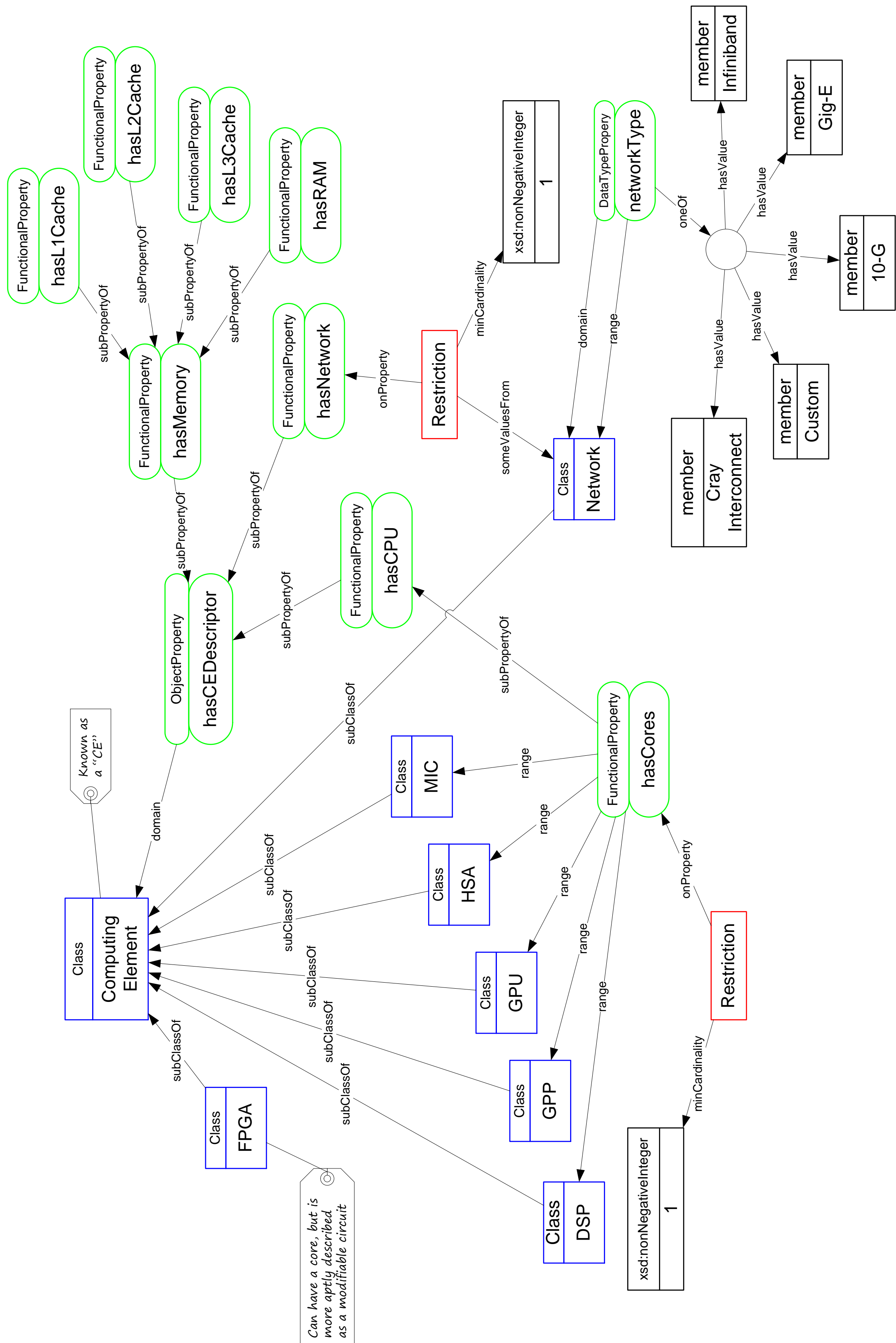


Figure 3.2: Computing Element Ontology

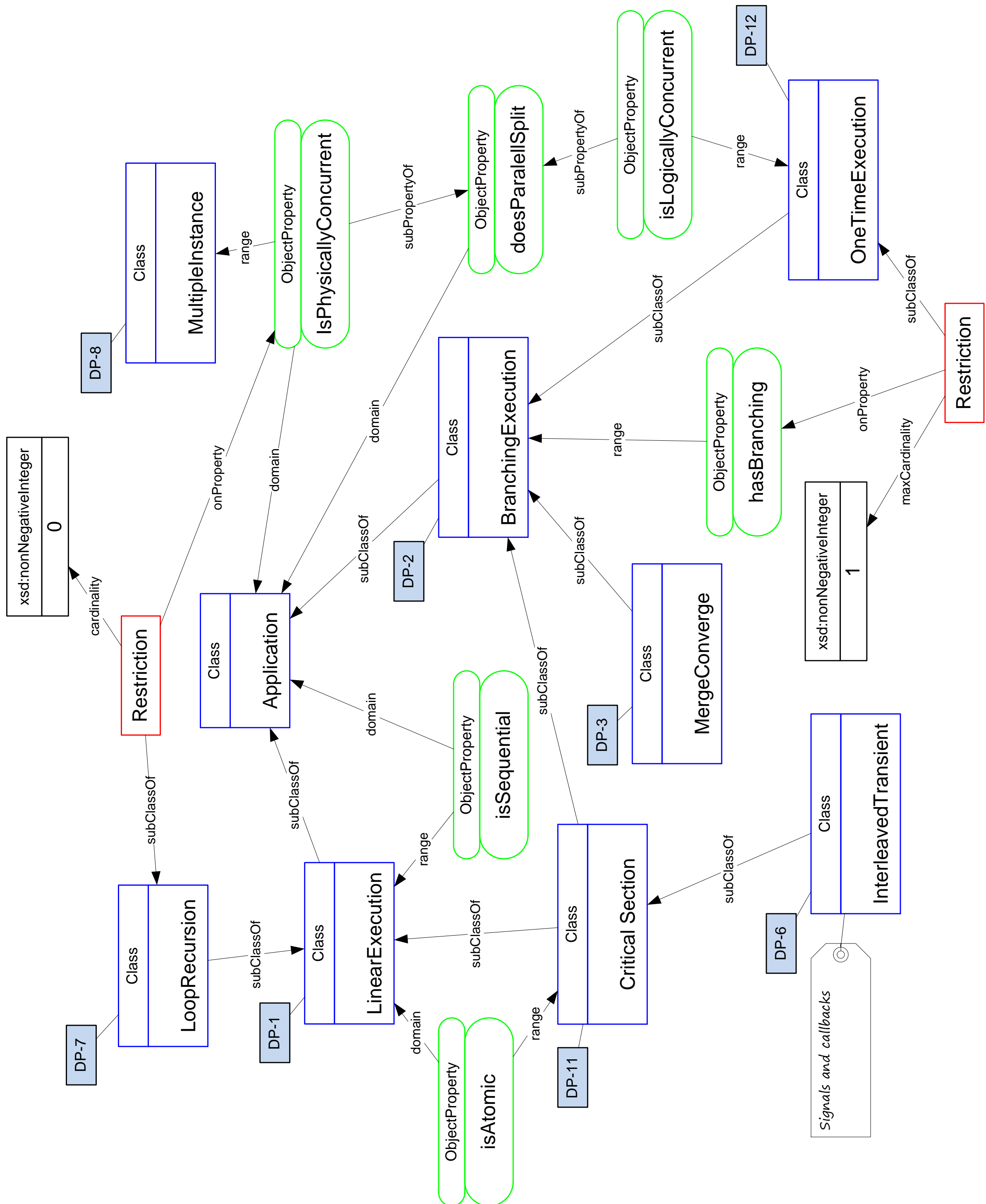


FIGURE 3.3: Decision Point Ontology

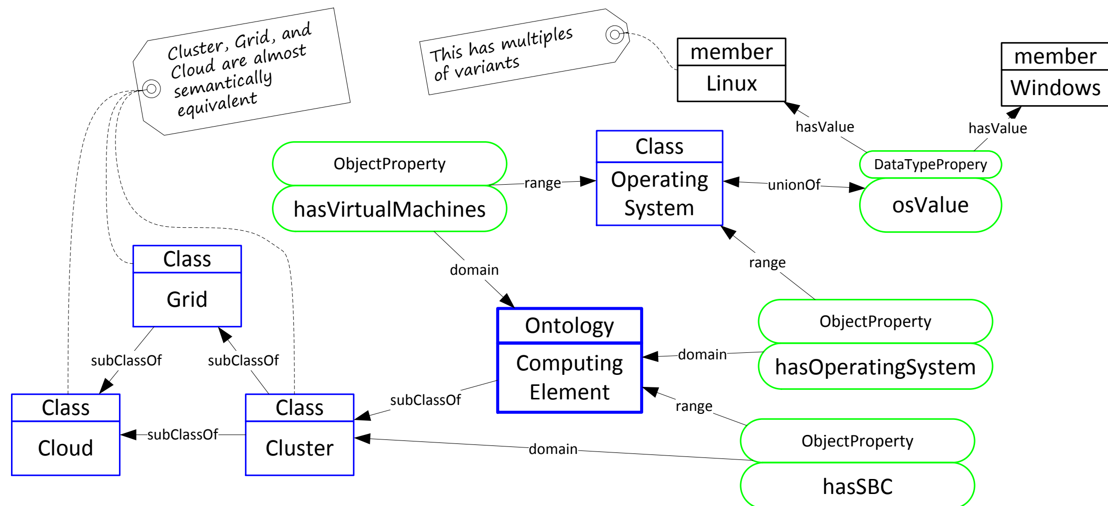


FIGURE 3.4: Cloud Ontology

3.3.3 Agent Supplied Cloud System Data

Using cloud resources requires that agents be associated with resources and those agents advertise the resource. Consumer agents must be able to query for resources (via a resource agent) so that algorithm to computing element mapping can occur. Placing the agent correctly depends on which service layer is used. Cloud systems can be separated into four major layers. Figure 3.5 from Sosinsky [Sos11] delineates three of these layers.

- Infrastructure as a Service (IaaS) is utility computing with physical or virtual machines with storage, hypervisors, and the ability to scale services. Additional components include firewalls, load balancers, pools of IP addresses, virtual local area networks (VLANs), and support software. Network as a Service (NaaS) cloud services are network and transport connectivity services, inter-cloud network connectivity services, or both provided to application service providers and web service organizations for data communications. This group of services is typically bundled with IaaS.
- Top level user applications are the basis for Software as a Service (SaaS).
- Software in the form of operating systems, files systems, and IPC communication are the basis for Platform as a Service (PaaS); that is, the software ecosystem, usually single system specific, in which applications operate. Services, Agents, and RITA are part of the PaaS layer.

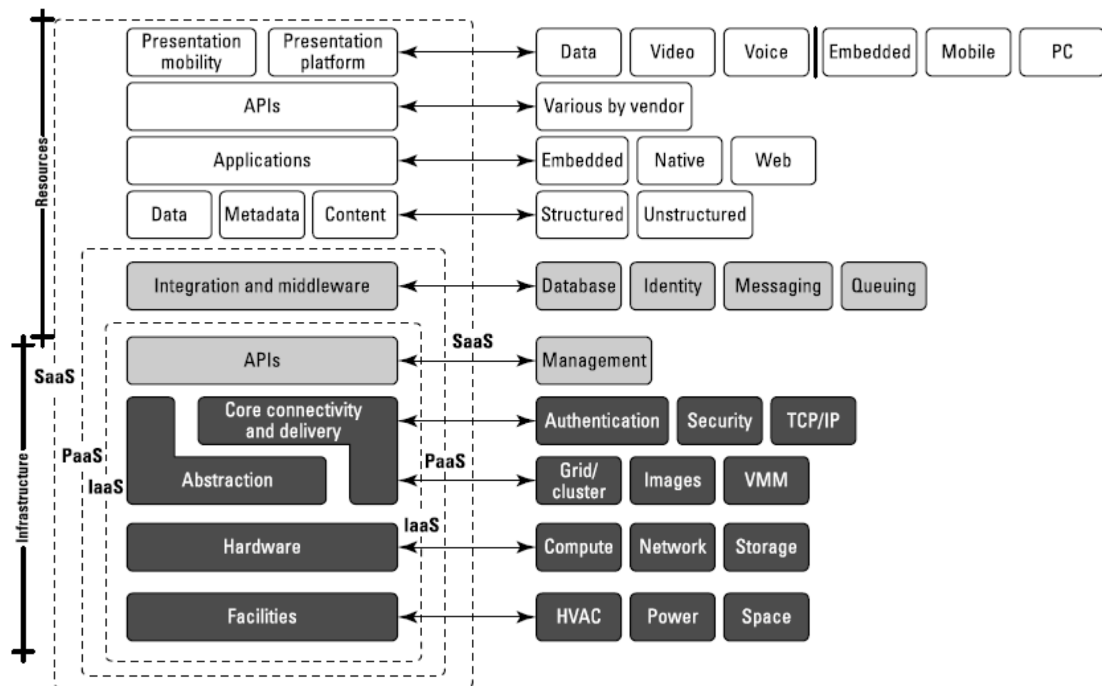


FIGURE 3.5: Cloud Reference Model.

Agents in the PaaS layer of each system instance, virtual or physical, have four functions:

1. Web services (WS)
2. Service Ontology (SO)
3. Resource agents (RA)
4. Consumer agents (CA)

Web services are the mechanism for other systems to query for data that the resource agent populates data in a local store that is retrieved by query from remote clients.

The **service ontology** provides the service specifications that describe the resources available as computing element types. Web services provide a structured message, using XML, identifying the computing elements and their location in the networking topology and geographic location. The locations of web services are expressed as uniform resource identifier[13] (URI) addresses. The following is an example of a web service definition using VXDL in its XML form.

[13] http://www.w3.org/TR/uri-clarification/ and https://www.ietf.org/rfc/rfc2396.txt

ExampleVXDLCode.xml

```xml
1 <?xml version="1.0" encoding="UTF-8" ?>
2 <description xmlns=" http://www.ens-lyon.fr/LIP/RESO/Software/vxdl"
3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4   xsi:schemaLocation="http://www.ens-lyon.fr/LIP/RESO/Software/vxdl VXDL.xsd">
5   <virtualInfrastructure id="TwoServer" owner="RITA project">
6     <user>Wallace</user>
7     <startDate>2014-11-8T08:00:00</startDate>
8     <totalTime>PT11H</totalTime>
9     <vGroupid="Storagecluster" multiplicity="20">
10      <vNode id="nodes storage cluster">
11        <storage>
12          <interval> <min>200</min> </interval>
13          <unit>GB</unit>
14        </storage>
15      </vNode>
16    </vGroup>
17  <vGroup id="Filtering cluster" multiplicity="40">
18    <vNode id="nodes filtering cluster">
19      <memory>
20        <interval> <min>4</min> </interval>
21        <unit>GB</unit>
22      </memory>
23      <cpu>
24        <cores>1</cores>
25        <frequency>
26          <interval> <min>2.0</min> </interval>
27      <unit>GHz</unit>
28      </frequency>
29      </cpu>
30    </vNode>
31  </vGroup>
32  <vGroup id="Mapping cluster" multiplicity="30">
33    <vNode id="nodes mapping cluster">
34      <memory>
35        <interval> <min>4</min> </interval>
36      <unit>GB</unit>
37      </memory>
38    <cpu>
39      <cores>1</cores>
40        <frequency>
41          <interval> <min>2.0</min> </interval>
42        <unit>GHz</unit>
43        </frequency>
44    </cpu>
45    <storage>
46      <interval> <min>80</min> </interval>
47      <unit>GB</unit>
48    </storage>
49    </vNode>
50  </vGroup>
```

ExampleVXDLCode.xml

Resource agents manage both physical and virtual machine activation for advertised computing elements by web services. Resource agents control access to physical and virtual machines. Resource agents receive requests, resolve requirements from service providers, and then handle the requests via their associated web service, returning the output to consumer agents. Resource agents manage cloud provider resources. The resource agent performs several accounting functions for the cloud providers, such as:

i. Allocating and releasing cloud resources when tasking has been assigned or completed
ii. Tracking available resources
iii. Synchronizing the execution of concurrent and parallel RAs
iv. Establishing service tasking with consumer agents.

Consumer agents compose and provide a single virtualized service to cloud resources by:

i. Selecting and contacting a set of possibly heterogeneous service providers
ii. Handling consumer update requests
iii. Receiving and mapping consumer requirements to available cloud resource types
iv. Submitting service composition requests to RAs.
v. Belonging to the application as a library entry point (Window systems: .DLL or Linux/UNIX: .so) performing web searches for advertised services (web crawler discovery of services) or the consumer agent may be constrained to only query for known systems such as EC2.

3.3.3.1 Open Source IaaS and PaaS

To support agent actions in the cloud a compatible pairing of IaaS and PaaS must exist as part of the cloud ecosystem. From hundreds of IaaS and PaaS systems reviewed [Fin13, Goo13] there are several open source systems consistently described as being some of the best ecosystems to use.
The IaaS systems listed are:

- OpenStack (`http://www.openstack.org`)
- OpenNebula (`http://www.opennebula.org`)
- Eucalyptus (`http://www.eucalyptus.com`)
- Google Compute Engine (`http://cloud.google.com/products/compute-engine`)

The PaaS systems listed are:

- AppScale (`http://www.appscale.com`)
- Cloud Foundry & BOSH (`http://www.cloudfoundry.com`)
- OpenShift (`http://www.openshift.org/`)

The open source products leading PaaS and IaaS development, and having the highest impact on the industry, are OpenShift and OpenStack respectively [Met12]. These products have integrated auto-scaling, fluid fault tolerance, and support developer extensions. OpenShift winning the "Best Platform-as-a-Service" category for The Cloud Awards 2013 program [GBRW13] and the success of OpenStack [Bad13] led to selecting these technologies for this work. OpenStack and OpenShift are more agnostic in their requirements for cloud computing environments and have built-in configuration controls for heterogeneous computing[14] than AppScale, and given the dynamic environment of open-source software, the Google Compute Engine and AppScale products would be a good second selection. Figure 3.6 shows the the overall flow of control for analyzing, mapping, compiling, and deploying an application. Prior to using the analyzer, an RA would have to be deployed to systems that are advertising their resources. In Figure 3.6 systems $\mathcal{A}, \mathcal{B}, \mathcal{C} \ldots \mathcal{N}$ have an RA installed. The RA uses WSDL[15] to describe resources that are discovered by a CA web crawl. WSDL is,

> . . . an XML format for describing network services as a set of endpoints operating on messages containing either document-oriented or procedure-oriented information. The operations and messages are described abstractly, and then bound to a concrete network protocol and message format to define an endpoint. Related concrete endpoints are combined into abstract endpoints (services).

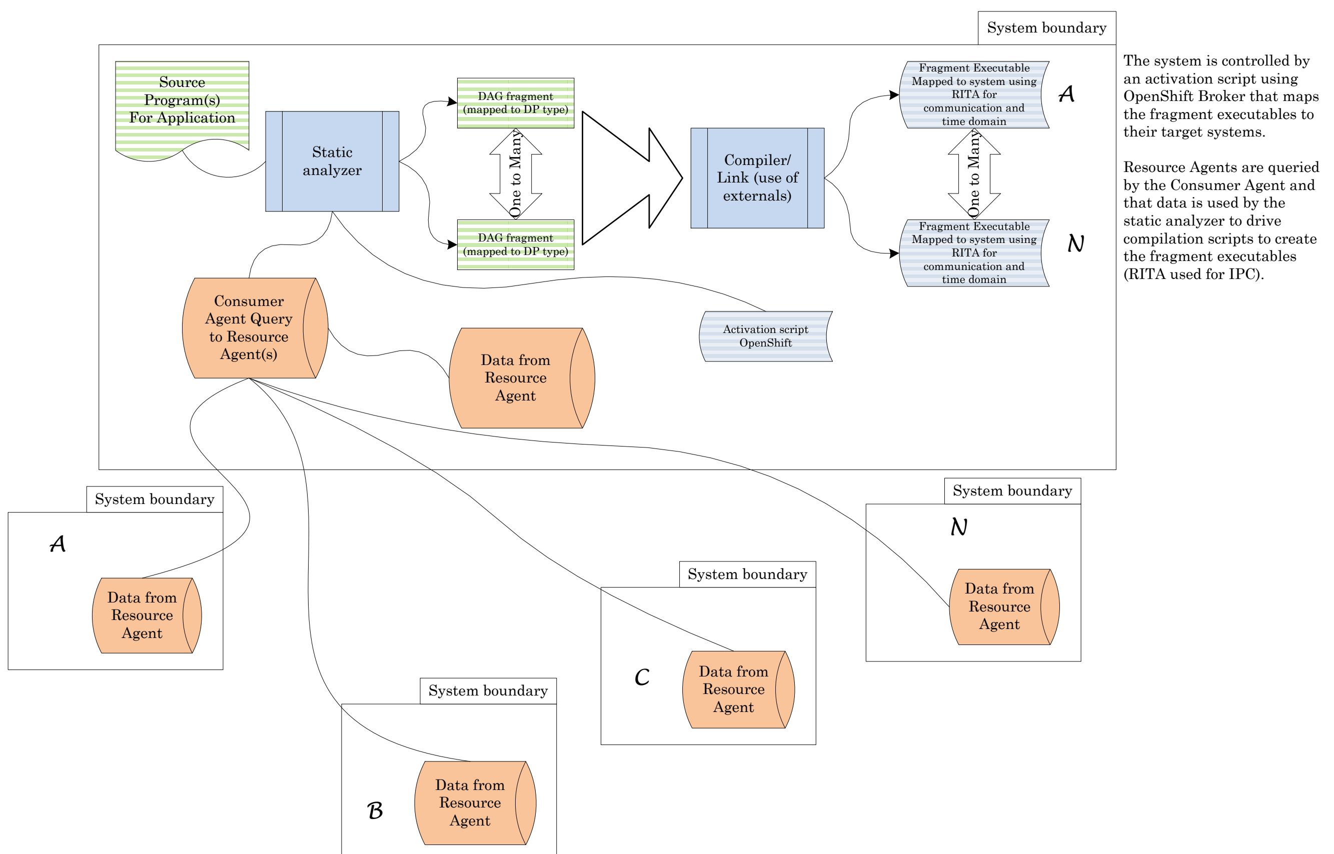


FIGURE 3.6: Distributed Flow of Control with CA and RA.

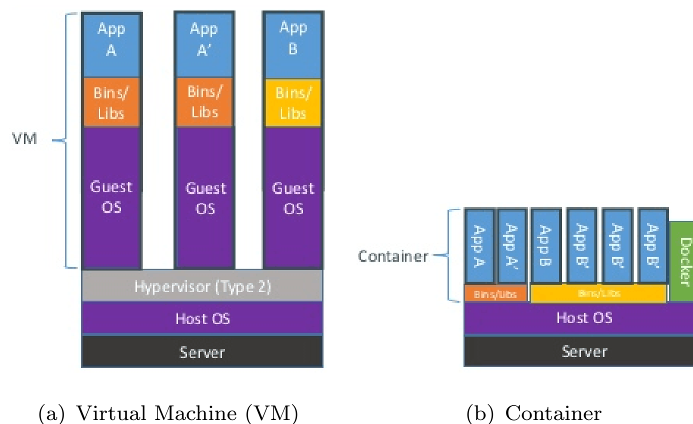
[14] https://wiki.openstack.org/wiki/HeterogeneousInstanceTypes
[15] Web Services Description Language, http://www.w3.org/TR/2001/NOTE-wsdl-20010315

Since August 2014[16], the OpenShift community has released its third generation OpenShift platform which integrates four significant capabilities facilitating algorithm mapping which are listed in Table 3.5.

Subsystem	Description
Docker	Technology for the *libcontainer* project. A container is a self contained execution environment that shares the kernel of the host system and which is optionally isolated from other containers in the system by specifying configuration options for the container
Kubernetes	A standard for application containers in the OpenShift ecosystem so they can be managed at very large scales. Applications typically span multiple containers deployed across multiple hosts
geard	`geard` (a gear *daemon*) is a command-line client and agent for integrating and linking Docker containers into `systemd` across multiple hosts and is a core utility in OpenShift Origin helping administrators install and manage application components
Project Atomic	A lean operating system designed to run Docker containers

TABLE 3.5: OpenShift Subsystems

An OpenShift container can be compared to a VM. This is a valid comparison until the resources required by each instance are analyzed. In a virtual machine there is a full operating system, device drivers, memory management, and associated utilities for the system. Containers use and share the O/S and device drivers of the host; thus containers have a smaller memory allocation footprint than VMs and will start up much faster and have better performance at the expense of less system isolation and greater compatibility requirements due to sharing a host kernel. This comparison is like-and-kind for a `fork()/exec()` pair versus a light weight process (i.e. a thread). Graphically the comparison between the two is shown in Figure 3.7. In Figure 3.7(a) the "Guest OS" consumes a great deal of space (on the order of tens of MB) and in Figure 3.7(b) the Docker interfaces shares the host O/S but provides a common API and some thread-oriented execution separation.



(a) Virtual Machine (VM) (b) Container

FIGURE 3.7: Comparison of VM and Container Memory Footprints

[16] https://blog.openshift.com/openshift-v3-platform-combines-docker-kubernetes-atomic-and-more/

Using these third generation capabilities that support RA and CA, RITA — as a PaaS — provides new functionality:

- The priority-based model of execution has tasks that can only be preempted by another task of higher priority. However, scenarios can arise where a lower priority task may indirectly preempt a higher priority task, in a sense inverting the priorities of the associated tasks, and violating the priority-based ordering of execution. This is called "priority inversion" and usually occurs when resource sharing is involved. RITA prevents application unbounded priority inversion by reducing multiprocessing interrupt thrash conditions leading to this problem
- Having event propagation designed for partitioned, communicating processes in a "share all," "share partial," or a "share nothing" parallel environment requires heavy use of synchronization primitives, publication/subscription systems, or message passing interfaces. RITA does not depend on tight integration and a shared time domain so it natively supports a "share nothing" loosely coupled message environment giving it semantics that are the least restrictive when aggregating processes for federated distributed system operation
- High performance worker threads used in "scatter gather" configurations can use RITA bifurcation of communication and computation to improve performance by encouraging a functional programming style. RITA purposely separates communication from application execution allowing function evaluation that avoids state dependent or user-controlled, globally mutable data. Thus a declarative programming environment is supported so that the output value of a function depends only on its input arguments thus eliminating side effects
- Canonical event forms as described in §2.4.5 define the communication forms supported in a distributed application making communication explicit and, with use of the condition-event matrix, reducing communication load-based traffic, race-condition errors

Cloud computing has, at its core, the ability to do many parallel homogeneous things at once or to do many heterogeneous things at once or a mixture of the two; thus there needs to be a network-based communication system in place to allow process control. Code written for cloud computing should create processes that do not depend on related processes being on the same system, or even within the same cloud instance. Processes should, instead, depend on the IPC of the system. As processes execute they are neither autonomous nor independent so they need to communicate with each other and that communication can quickly increase to a level that decreases system throughput.

3.3.3.2 Integration of RITA into OpenShift

RITA capabilities are mapped into OpenShift Origin/OpenStack systems. Red Hat OpenShift Origin/OpenStack uses the terminology in Table 3.6.

Term	Definition
Broker	Host manager, controls nodes
Cartridge	A technology stack or framework (PHP, Perl, JEE, Python, MySQL, et.al.) to build applications
Plugin	System utilities used with a kernel; `authconfig`, DNS, et.al.
Gear	Allocation of memory, compute, and storage resources to run applications. A container is like a "gear" but differs in resources required
Node	A computer; single-board, blade, et.al. Usually has only local storage for the kernel
Application	Instantiation of a cartridge (over-loaded term). Differentiated from a nominal cartridge in that a user has written additional application code that uses cartridges
Scaled Application	Application instantiated in multiple gears

Table 3.6: OpenShift Term Definitions

For OpenShift Origin, a "Gear" is a shell on a node in a shared-nothing instance of the OpenStack IaaS. The usage is to "spin-up" another "Gear" when more instances are created by the OpenShift load balance utility, HA-Proxy. Using the OpenShift subsystems identified in Table 3.5 the computing element mapped "fragmented" algorithms can be easily containerized and pushed to their respective systems — physical, virtual, or both — and then started on the target system.

The RITA processing stack has two forms; the first is a processing "cell" for singleton RITA systems as shown in Figure 3.8. In Figure 3.9, the singleton form is expanded by moving the embedded event engine out to form an independent, centralized event engine per physical computing system. This allows multiple VMs supported on a single system to have minimal communication latency and also share a unified time domain.

The required IPC system in OpenShift Origin is the Apache Foundation ActiveMQ[17] which is accepted by cloud providers as a highly efficient JMS compliant system that can be tuned for the IaaS being used. The Go language is supported for ActiveMQ. Deploying RITA to OpenShift Origin is done according to the OpenShift Origin Cartridge Developer's Guide [Hat13].

Using the OpenShift Origin terms, the mapping of the RITA infrastructure to OpenShift is shown in Table 3.7. While there is no VM support in RITA, it can be deployed as a Cartridge in an OpenShift Origin PaaS.

[17] http://activemq.apache.org/

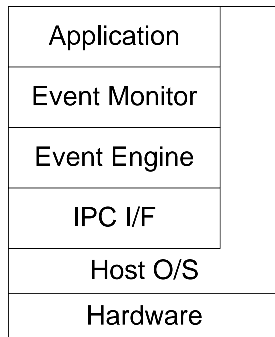


FIGURE 3.8: RITA Processing Cell

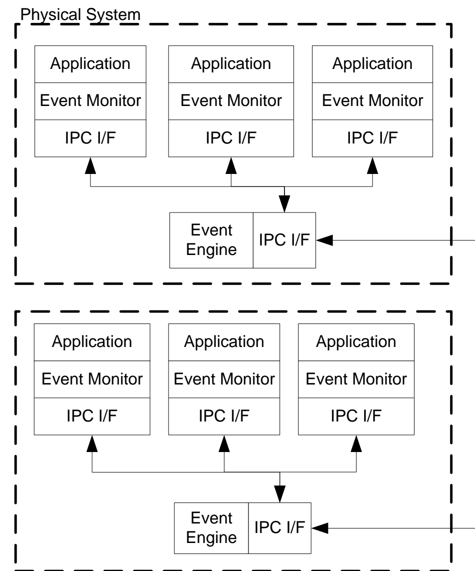


FIGURE 3.9: Multiple RITA Systems

OpenShift	**RITA**
Cartridge	Event monitor & engine
Plugin	IPC I/F
Gear	Cell
Node	System
Application	Application
Broker	No VM control in RITA

TABLE 3.7: OpenShift, RITA Comparison

In addition to the third generation OpenShift PaaS subsystems described at the start of this section, the additional enabling technologies for application distribution through the OpenStack IaaS are: *a*) The Distributed Management Task Force (DMTF) Open Virtualization Format (OVF) [For13] *b*) The Distributed Resource Management Application API (DRMAA) [Fou12] sponsored by the Open Grid Forum, and *c*) A derivative of DRMAA, Simple Linux Utility for Resource Management (SLURM). With the development of *Docker* and the *libcontainer* project this method of application mapping would only be necessary if the PaaS services were not available in the processing stack as shown in Figure 3.7(b).

3.3.4 Ontological Weighted Match for Algorithms to CE

Using OWL as our description logic, it is possible to machine process the CE ontology (see Appendix D); and categorize source code to DP/WCP patterns providing a CE

mapping for that code by using the Static Analyzer. To identify a computing element, the objective is explicitly determining the similarity between an algorithm's "weight" and "shape" to match it to one or more CEs represented by the ontology in Figure 3.2. To understand the hierarchy for rules a "similarity stack" for matching is shown in Figure 3.10. Ontologies are based on vocabularies that are well defined, well understood, and have a generally accepted meaning. The left-hand part of Figure 3.10 shows this arranged along a Semantic Complexity axis derived from the "layer cake" of Berners-Lee [BLHL01]. The right-hand part shows that domain-specific verbs, i.e. knowledge, can span any level of ontological semantic complexity. In Figure 3.11, the relationship of the CEs is shown using complex programming and specialized circuitry axes. This relationship was derived from the work by Hennessy and Patterson [HP11]. Programming any of the CEs can have an overlap in DP categories as shown in Figure 3.3. In Figure 3.12, the ratio of instructions that map to WCP type and summed in a DP category gives the numeric value used in Algorithm 6 at line 15 on page 106.

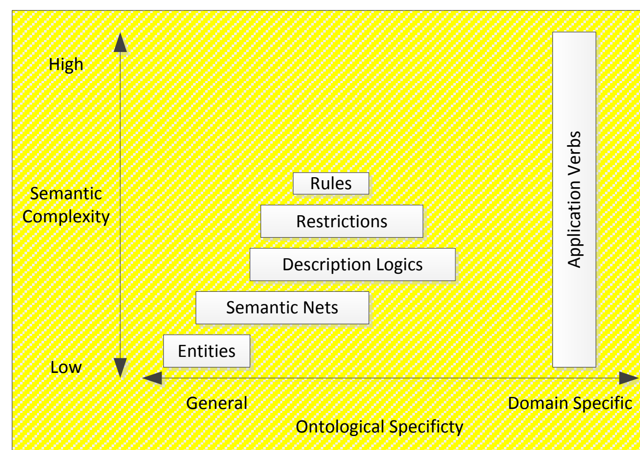


FIGURE 3.10: Ontological Similarity Stack

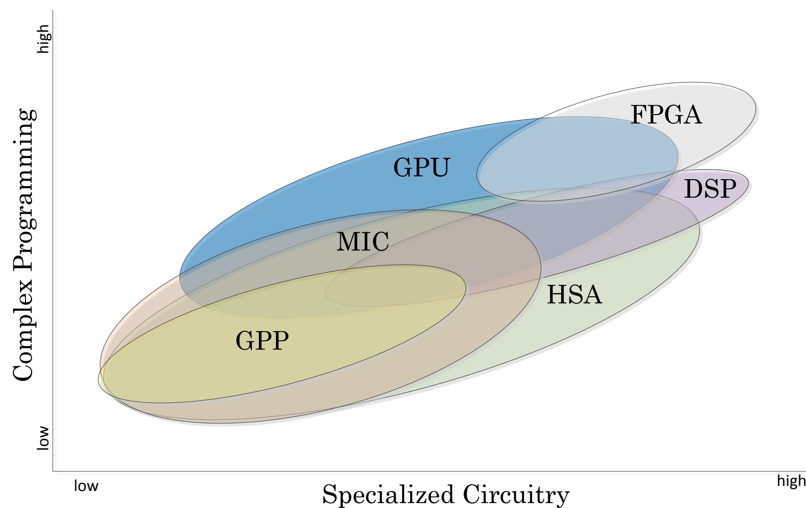


FIGURE 3.11: CE Architecture Overlap

An algorithm code base can have multiple DP categories. Scoring each of the DP categories involves scoring each of the WCP patterns that belong to that DP category for any SLOC[18] that have been allocated to the WCP. This is done by calculating the affinity and weight, standardizing the values and then normalizing the values so the overall score can be calculated for the analyzed code. The electronics industry has no uniform merit values for affinity and weight, thus these values based on empirical experience with specific architectures. This method is a framework for comparison of CE architectures. CEs vary significantly within architecture families as shown by the high degree of overlap shown in Figure 3.11. Values used for calculation of weight would be input for the specific models of architecture being compared. In Figure 3.12 an example manual calculation is provided for elaboration of the computations. After each maximal value is found (as shown by the "X" in the red cell), the architecture source line of code adjusted values are summed by column. In this example, all GPP architecture scores sum to a normalized value of 3.78 and the HSA architecture scores sum to a normalized value of 3.72. SLOC scores are log_{10} normalized values to provide a linear relationship and not give the simple (SLOC×Normalized) value an undue bias in the scoring value. The Standardize value, Z, gives us how many standard deviations a datum is above or below the mean, μ, of the population.

In this example, the program is small and so is the difference of 0.06 points between the two architectures. Thus there is a requirement that a significant discriminant is needed to do the differentiation to perform directed compilation. Indeed, any of the non-GPU or FPGA architectures will have very low differences in scores as their architectural elements have a high degree of overlap. In §3.4.4 DDE calculations are used to provide this additional discriminant after first discussing allocation of resources in §3.4.

$$
\begin{array}{ll}
\texttt{Normalize:} & x' = \frac{x-\min(x)}{\max(x)-\min(x)} \\
\texttt{Mean:} & \bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i \\
\texttt{Standardize:} & Z = \frac{X-\mu}{\sigma} \\
\texttt{Population St.Dev.:} & \sigma_p = \sqrt{\frac{\Sigma(x-\bar{x})^2}{n}} \\
\texttt{Max weight SLOC:} & max(\log_{10}(SLOC * x'_i))
\end{array}
\tag{3.1}
$$

[18] Source Line of Code

<Architecture>(<WCP Number>) = Well known affinity for that WCP

DP-1 Sequential			GPP(1)	MIC	HSA	DSP	GPU	FPGA	
	WCP-1	Affinity (1=low, 10=high)	10	9	9	9	9	9	
		Weight (sums to 1)	0.3	0.2	0.2	0.1	0.1	0.1	1.55 mean()
			3	1.8	1.8	0.9	0.9	0.9	0.763217 stdev.p()
			1.89985	0.32756	0.32756	-0.85166	-0.85166	-0.85166	Standardize
			1.00000	0.42857	0.42857	0.00000	0.00000	0.00000	Normalized 0 to 1
	SLOC	500	2.70	2.33	2.33	0.00	0.00	0.00	
	Max	2.70	X						
DP-2 Branching			GPP	MIC	HSA	DSP(42)	GPU(42)	FPGA	
	WCP-2	Affinity (1=low, 10=high)	5	8	9	8	8		
		Weight (sums to 1)	0.05	0.25	0.25	0.25	0.2		1.62 mean()
			0.25	2	2.25	2	1.6		0.715961 stdev.p()
			-1.91351	0.53076	0.87994	0.53076	-0.02793		Standardize
			0.00000	0.87500	1.00000	0.87500	0.67500		Normalized 0 to 1
	SLOC	59	0.00	1.71	1.77	1.71	1.60		
	Max	1.77			X				
	WCP-4	Affinity (1=low, 10=high)	9	2	2	2	2		
		Weight (sums to 1)	0.4	0.15	0.15	0.2	0.1		0.96 mean()
			3.6	0.3	0.3	0.4	0.2		1.321514 stdev()
			1.99771	-0.49943	-0.49943	-0.42376	-0.57510		Standardize
			1.00000	0.02941	0.02941	0.05882	0.00000		Normalized 0 to 1
	SLOC	12	1.08	-0.45	-0.45	-0.15	0.00		
	Max	1.08	X						
	WCP-6	Affinity (1=low, 10=high)	3	6	8	8	3		
		Weight (sums to 1)	0.05	0.25	0.35	0.3	0.05		1.4 mean()
			0.15	1.5	2.8	2.4	0.15		1.104083 stdev.p()
			-1.13216	0.09057	1.26802	0.90573	-1.13216		Standardize
			0.00000	0.50943	1.00000	0.84906	0.00000		Normalized 0 to 1
	SLOC	62	0.00	1.50	1.79	1.72	0.00		
	Max	1.79			X				
	WCP-18	Affinity (1=low, 10=high)							mean()
		Weight (sums to 1)							stdev.p()
									Standardize
									Normalized 0 to 1
	SLOC	0	0.00	0.00	0.00	0.00	0.00		
	Max	0.00							
	WCP-42	Affinity (1=low, 10=high)							mean()
		Weight (sums to 1)							stdev.p()
									Standardize
									Normalized 0 to 1
	SLOC	0	0.00	0.00	0.00	0.00	0.00		
	Max	0.00							

FIGURE 3.12: DP Scoring

3.4 Processing Allocation to Cloud Components

ALLOCATING cloud system resources currently depends on manual allocation of application executables to one or more system images to provide balanced and efficient data processing. This allocation can be improved by novel use of algorithm identification routines and a controlled method for communicating sequential processes. In this work the static code analysis uses two primary axioms:

Axioms for code analysis

1. All work can be arbitrarily divided into sub-tasks indicated by pragmas. This creates the **divisible load**
2. Delay local throughput optimization until end of allocation. This creates the assumption of **steady state throughput**

By relaxing the initial allocation and execution optimization from the beginning of code analysis, the allocation algorithm can be simplified and optimization will be simpler to perform as overly complex allocation decisions can be delayed until the full elaboration is understood. This requires multiple passes through source code analysis with

recalculation of the DDE in Equation 2.22 leading to better mapping of algorithms to computing elements.

All compilation from a high level representation to a machine level instructions use an intermediate representation (IR). Intermediate representation during code analysis and compilation can use well known structures of abstract syntax trees, control flow graphs (CFG), static single assignment, and a linear representation. Most imperative languages map neatly to CFGs while functional languages map to Value State Dependence Graph (VSDG) IRs [JM03] which is an extension of the Value Dependence Graph (VDG)[WCES94]. Generating code from a program represented as a CFG is straightforward; however more flexible IRs require additional analysis and transformation to give an ordering of instructions before being passed to the code generator. Target architectures can be stack or register machines. This work uses Go language and its intermediate form represented in Go macro assembler as...

> ...it is not a direct representation of the underlying machine. Some of the details map precisely to the machine, but some do not. This is because the compiler suite ...needs no assembler pass in the usual pipeline. Instead, the compiler emits a kind of incompletely defined instruction set, in binary form, which the linker then completes. In particular, the linker does instruction selection, so when you see an instruction like `MOV` what the linker actually generates for that operation might not be a move instruction at all, perhaps a clear or load. Or it might correspond exactly to the machine instruction with that name. In general, machine-specific operations tend to appear as themselves, while more general concepts like memory move and subroutine call and return are more abstract. The details vary with architecture, and we apologize for the imprecision; the situation is not well-defined. The assembler program is a way to generate that intermediate, incompletely defined instruction sequence as input for the linker.
>
> — `https://golang.org/doc/asm`

This not a new concept. When I worked for Digital Equipment Corporation in the 1980-1990s, the GEM back-end[19] [GBGN93] was in use with a number of front-end compiler products for a variety of languages and hardware and software platforms. GEM[20] produced portable, modular software components with carefully specified interfaces that simplified the engineering of diverse compilers. Thus GEM was a single optimizer, independent of the language and the target platform that could transform the intermediate language generated by the front end into a semantically equivalent form that executes faster on the target machine. Mainstream compilers are including newer intermediate representations from open-source projects which have research projects developing compiler internals such as LLVM[21].

[19]DTJ V4 N4, 1992 `http://www.hpl.hp.com/hpjournal/dtj/vol4num4/vol4num4art8.pdf`
[20]GEM is not an acronym, it was *en vogue* for projects to be named at Digital, e.g. OPAL and PRISM
[21]`http://llvm.org/`

In this work, the necessary intermediate representation will be done through use of the native Go language `Parser` package. This package provides the AST for Go source code as input. The AST is annotated by attributes identifying which WCP type it is, grouped as DP type, from Tables 3.3 and 3.4 respectively. These attributes are used by the static analyzer to generate the DAG fragments.

3.4.1 Algorithm Recognition and Mapping to CEs

The algorithms the static analyzer uses (Figure 3.6) that process source code into DAG fragments are packaged into task-parallel components producing compilation units for both data-parallel and data-serial processing. The compilation units are then compiled into executables that are executed on systems that have the computational circuitry needed to directly support the algorithmic method. These executables are moved to the correct systems based on the NML and VXDL descriptions. The movement of the executables are done by Docker. Of note, in a heterogeneous environment, there will be target system compilation and linker dependencies. As the target systems are known from the VXDL descriptions, target compilation and execution can be controlled by Docker as part of the distribution process.

The process starts by using the application `Makefile` to examine the source files that make-up the application. To illustrate the creation of a DAG fragment the following simple computation is shown in Figure 3.13 with its DAG in Figure 3.14.

```
x = (a + b) * (a - b);
y = a * x - b * x;
```

FIGURE 3.13: Data Parallel Calculations

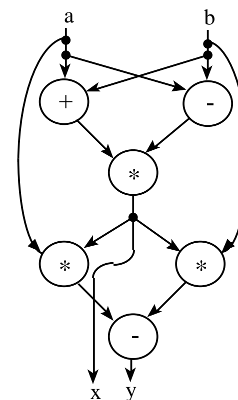


FIGURE 3.14: Data Parallel DAG

The DAG has the potential for parallelism addition of a and b can be done at the same time as the subtraction. A lateral traverse across the DAG indicates a potential for parallelism. If we have several add and multiply actions (e.g. matrix multiplication) the DAG could be evaluated by replication parallelism. This example also shows the

potential for pipeline parallelism if the computation is used in a larger repetitive calculation computing x and y is done many times with different values for a and b. Assuming equal stages for a multiply and an add (a blatantly naïve assumption), the DAG can be seen as a two-stage pipeline for x and a four-stage pipeline for y. While efficiency is a focus, instruction order or processor pre-fetch pipelining is not the primary focus of this work. An optimizing compiler for a computing element would be used for that level of parallelism. In a similar way, algorithm code segments can be analyzed and the specific decision point and workflow control patterns can be determined.

This work is focused on discrete algorithmic execution for distributed, federated computing elements *en masse*. To do that at a source code level, enough parallelism in the program function graph has to be exposed and then assigned to processors to minimize parallel execution time with consideration for communication overhead. This top level processing is shown in Algorithms 4, 5, and 6.

Algorithm 4 performs the initial mapping of guards and construction of the condition-event matrix based on the VXDL and NML output from the Go language parser. As this is an iterative process based on each of the source file dependencies, this is algorithm cycles through each executable created from the Go tool-chain `Makefiles` used to create the system.

Algorithm 5 adds control flow attributes to the DAGs from the initial mapping from Algorithm 4. This algorithm has simple statements for the evaluation and examination of the DAGs in intermediate form generated from the Go Parser; thus almost every line from line 4 through line 15 is an iterative statement as each module's intermediate form records are evaluated for variables, controls, and system calls. At the end of the algorithm, at lines 16 and 17, the elements are grouped to create functional programming units based on the WCP and DP attributes; see Tables 3.3 and 3.4 respectively.

Algorithm 6 uses the pre-processed data from Algorithms 4 and 5 and ontology to create the compilation units and OpenShift containers with the associated Docker scripts for the application. Of note, the line, "**if** $\Sigma(DAG)\ldots$," highlighted in **yellow**, is the calculation of the RITA DDE from Equation 2.22 which uses the evaluation of the NML and VXDL attributes in the DAG fragment that contribute to latency. The service level agreement (SLA) is stated *à priori* and is used as the boundary condition for mapping algorithms to computing elements. This processing occurs in lines 14—17 and is highlighted in **green**.

Algorithm 4: Top Level Mapping Algorithm

Input: makespan `Makefile` for system application
Input: VXDL & NML Agent data
Input: SLA for the application (in time cost, monetary cost, or both)
Output: $1 \ldots n$ sub-Makefiles mapped 1:1 HW type and a Map of Channels for algorithm communication

```
1  begin
2      Initialization of control structures and temporary files
3      foreach E ∈ M do                                  ▷ Executable target in Makefile
4
5          foreach D ∈ E do                              ▷ Dependency of Executable
6
7              foreach S ∈ D do                          ▷ Source file of dependency
8
9                  Create DAG from source file           ▷ Use of Go package Parser
10
11                 Call Add-CF-To-DAG(DAG, S)            ▷ Add Control-Flow Attr. to DAG
12                 Call Control-Flow-Processing(CF, DAG, SLA)
13                 if Guard needed then
14                     Create Guard G
15                     Add to Condition Event matrix
16         Write-out RITA control code module
17         Create sub-Makefile for E using paired WCP and HW target
```

Algorithm 5: Add Control Flow to DAG

Input: DAG, Source file
Output: DAG with CF attributes

```
1  Add-CF-To-DAG(DAG, S) begin
2      foreach N_i ∈ DAG do                                  ▷ Node in DAG
3
4          foreach DP_i identified, map corresponding WCP attribute do
                   ▷ Each DP_i identified will result in a WCP_i attribute applied based on the
               identified code characteristics.
5              Examine for system calls for thread or process creation
6              for All variables do
7                  Identify loop scope
8                  Local variables – candidate for HSA, GPU
9                  Global variables – candidate for MIC, HSA, GPP
10             Identify I/O modes
11             Establish internal branching scope for each function
12             Identify à priori run-time parameters such as data files, command line
               parameters, hard-coded values
13             Identification of calls for critical sections
14             Identify one time execution (i.e. fork with no join)
15             Identify program termination
             ▷ Fragment (multiple DAG nodes) should be at functional programming scope.
16         if N_i in functional scope then
17             Chain fragments together based on WCP & DP characteristics
18     return DAG fragment chain
```

Algorithm 6: Control Flow Processing

```
   Input: CF, DAG, SLA
   Output: Compilation Unit(s), OpenShift activation script, Docker script
1  Control-Flow-Processing(CF,DAG,SLA) begin
2      foreach CF ∈ DAG do                              ▷ Control Flow attributes of DAG
3
           ▷ Simple sequence has no partitioning regardless of size
4          if ¬ Simple sequence WCP then
5              Evaluate recursive functions and argument(s) for cycle type
6              Evaluate function and arguments for cycle type
7              Evaluate file I/O                          ▷ Block/NonBlock, Local, SAN,...
8              Evaluate network I/O                       ▷ System, Cluster, Network,...
9              Build decision point DAG fragment using CF
10             Initialize VXDL and NML graph traversal, P_{i...n}
11             foreach DAG decision point fragment do
12                 while SLA ¬ met do
13                     forall the DP pragma tagged CF attributes of DAG do
14                         Match to WCP
15                         Match WCP to HW using CE ontology
16                         Using CF, VXDL, and NML assign weighted costs to DAG
                           for fragment execution
17                         Mark VXDL, NML path i, used

18                     if Σ(DAG) fragment execution weighted costs ≤ SLA then
19                         SLA ← met
20                     else
21                         if ∀P_i traversed then
22                             Report FAIL

23             foreach WCP and HW pair do
24                 Map WCP and HW pair with RITA Condition element
25                 Add mapping to Condition Event matrix
26                 Write-out DAG fragment source file

27         else
28             Map WCP and HW pair to RITA Condition Element
29             Add to Condition Event matrix

30     Write-out OpenShift activation script
```

3.4.2 Decomposition Mapping Example Using Tower of Hanoi

Using the Java code in §F.1 to demonstrate this analysis, it shows that the `main` method is a simple sequence (DP-8/WCP-21) `while( true )` code loop. This code has a call to the `SolveTOH()` function that is a recursive (DP-8/WCP-22). The number of steps to solve this problem is given by the difference equation,

$$hanoi(n) = \begin{cases} & \square \\ 1, & \text{if } n = 1 \\ 2 \cdot hanoi(n-1) + 1, & \text{if } n > 1 \end{cases} \tag{3.2}$$

Examination of the code in §F.1 shows that any compiler decomposing this general code (inclusive of JIT[22] compilation methods) into a true parallel implementation for multicore or multiple SBC[23] would have to divine the intent of the author as there is no indication that parallelism has been indicated, or if it is even needed. This requires that the algorithm writer has to provide pragmas or code constructs allowing the compilation process to correctly partition and construct processes allowing such parallel operation.

Working from the two axioms for code analysis on page 101, the arbitrary division for the Tower of Hanoi algorithm is the recursive call and that each division can be equally mapped to a separate computing element for steady-state operation. Putting these axioms into action requires a Java-related parallel processing capability. At this writing, parallel processing is done by a Java MPI[24] or OpenMPI variant, Parallel Java 2 (PJ2) from Dr. Alan Kaminsky, Rochester Institute of Technology[25], or the support for parallel streams in Java 8.[26] In Java 8, an implementation would have to avoid issues with the implementation of the *parallel()* method as all parallel streams use the common fork-join thread pool and if a *submit()* is done for a long-running task, this would effectively block all threads in the pool. There is a "work-around" for this (i.e. the *parallel()* method is flawed) — I put quotes around it as it is a clever trick and not a true solution — by making an explicit call to the *submit()* method for worker threads. This degenerates quickly becoming a pathological series of calls requiring programmers to understand the internals of parallel streams and hard-code the fork-join pool which is not the smooth sequential-to-parallel implementation that the JDK developers intended.

To illustrate the steps of decomposition, the classic "Tower of Hanoi"[27](ToH) algorithm is used as an example. The smallest number of movements is given by the exponential value, $2^n - 1$, where n is the number of discs to be moved. Examination of the Java source in §F.1 shows that the structure of the program is a recursive, depth-first solution. A second, equivalent, program in Go is shown in §F.2. In this work the Go language is used as it has language elements that make it possible for the Static Analyzer to add a few lines of additional code to make the `hanoi(···)` function parallel, but not recursively so. This will be discussed shortly. In Listing 3.1 below the lines highlighted in yellow are the Go language expressions that allow parallel operation and communication where a pragma could control their use. The listing lines in blue are those that would have to be intimated by the code writer as they are specific to the ToH algorithm and can not be deduced by any parallel analysis tool. Given these few lines

[22] Just In Time compilation
[23] Single Board Computer
[24] Message Passing Interface
[25] http://www.cs.rit.edu/~ark/pj2.shtml
[26] First version available in the first half of 2015, http://www.java.com/en/download/faq/java8.xml
[27] A puzzle invented by the French mathematician Édouard Lucas in 1883.

the analyzer can recognize the simple merge (WCP-5) at line 62, multi-instance with *à priori* knowledge (WCP-15) at line 59, recursion (WCP-22) at lines 79 and 85, blocking discriminator (WCP-28) at lines 63-65, and explicit termination (WCP-43) initiated at line 64 to 67.

The goal of the Static Analyzer is assignment of code to computational units. There is a limit to the practicality of doing this and that is why the ToH algorithm was chosen as it shows these limits. In ToH, the recursive algorithm has an inherent order based on stack frames. This occurs at lines 79 and 85. In the full parallel version of the ToH program, all calls to the `hanoi(···)` function are replaced by `go hanoi(···)`, called "Go routines," which implement a light-weight thread of execution; thus lines 59, 79, and 85 all start concurrent execution of the `hanoi(···)` function as `go hanoi(···)`. In Go, the Go routine is a non-blocking call so, as can be seen in the code, the fall-through of execution from lines 79 and 85 would cause inappropriate code to execute and thus the algorithm would fail to execute correctly. The Go routines make ToH a multiply re-entrant, random execution (go routines execute non-deterministically) program. With this, the entire algorithm would have to be rewritten into a series of frame based data sets with code added to have the correct order performed to assure correct moves for the ToH discs. The Static Analyzer cannot be clairvoyant to add such code and thus the rewrite is not possible. The ToH is a very good example of code that can not be partitioned due to the algorithm's dependence recursive call frames to maintain state.

```
1  package main
2
3  import (
4    "flag"
5    "fmt"
6    "math"
7  )
8
9  const (
10   A = iota // start at 0, then 1, 2, ...
11   B
12   C
13 )
14
15 type stack struct {
16   Name  string
17   Discs []int
18 }
19
20 func (s *stack) pop() int {
21   var disc int
22   stack_length := len(s.Discs)
23   disc = s.Discs[stack_length−1]
24   s.Discs = s.Discs[0 : stack_length−1]
25   return disc
26 }
27
28 func (s *stack) push(disc int) {
29   s.Discs = append(s.Discs, disc)
30 }
31
32 var (
33   discs  = flag.Int("discs", 4, "Number of discs to use.")
34   moves  = 0
35   stacks []*stack
36   e     chan int
37 )
38
```

```
39 func init() {
40   flag.Parse()
41 }
42
43 func main() {
44   e = make(chan int)
45   fmt.Printf("Solving Towers of Hanoi for %d discs\n", *discs)
46   // Create the stacks
47   // A Go slice type, initial rows, and capacity (i.e. max length)
48   stacks = make([]*stack, 3, 3)
49   // Initialize the stacks by adding their name, and the "columns" (another slice) and
50   // set the initial length to zero and the capacity to the number of discs.
51   stacks[A] = &stack{"A", make([]int, 0, *discs)}
52   stacks[B] = &stack{"B", make([]int, 0, *discs)}
53   stacks[C] = &stack{"C", make([]int, 0, *discs)}
54   // Create the discs
55   for i := 0; i < *discs; i++ {
56     stacks[A].Discs = append(stacks[A].Discs, i)
57   }
58
59   go hanoi(*discs, A, C)
60   max := int(math.Pow(2, float64(*discs))) - 1
61   for {
62     x := <-e  //channel read
63     if x >= max {
64       break
65     }
66   }
67 }
68
69 func via_stack(src, dst int) int {
70   return (A + B + C - src - dst)
71 }
72
73 // This recursive function launches multiple go function threads.  Note
74 // the hanoi() function has global data.  Passing all as stack or channel
75 // data would be necessary for non-shared memory multicore
76 func hanoi(d, src, dst int) {
77   via := via_stack(src, dst)
78   if d > 1 {
79     hanoi(d-1, src, via)
80   }
81   moves++
82   stacks[dst].push(stacks[src].pop())
83   fmt.Printf("Move #%d: %s -> %s\n", moves, stacks[src].Name, stacks[dst].Name)
84   if d > 1 {
85     hanoi(d-1, via, dst)
86   }
87   e <- moves //channel write
88 }
```

Listing 3.1: Partial Parallel ToH in Go

3.4.3 Worker-Dispatch Mapping Example

In Listing 3.2 the normal form for multiple parallel algorithms is shown. This form is very different from the ToH routine to incorporate parallelism. At lines 24 and 27 (highlighted in yellow) the two functions `worker(···)` and `dispatch(···)` are started as Go routines. In `dispatch(···)`, at line 48, the input channel receives data for processing at line 35 in `worker(···)` (highlighted in blue). When the worker thread is complete the result is sent back to the dispatch receiver channel at line 41. This is then sent to the output channel in `dispatch(···)` at line 49. The `main()` function receives this data at line 28 (highlighted in green).

The use of channels to communicate not only data, but also a communication channel is common in Go programs. In this example the input data is limited to the slice declaration at line 16 with simulated work being done on lines 36-39. Next we will examine the output of the Worker threads in Listing 3.3.

```
1  package main
2
3  import (
4    "fmt"
5    "math/rand"
6    "runtime"
7    "time"
8  )
9
10 type inputCS struct {
11   input string
12   rCh   chan string
13 }
14
15 // dummy data set
16 var input = []string{"one", "two", "three", "four", "five",
17                      "six", "seven", "eight", "nine", "ten"}
18
19 func main() {
20   nWorkers := runtime.NumCPU()
21   fmt.Println("Number of workers: ", nWorkers)
22   inCh := make(chan *inputCS)
23   for i := 0; i < nWorkers; i++ {
24     go worker(inCh, i)
25   }
26   outCh := make(chan chan string, nWorkers*2)
27   go dispatch(inCh, outCh)
28   for rCh := range outCh {
29     fmt.Println(<-rCh)
30   }
31 }
32
33 func worker(inCh chan *inputCS, i int) {
34   rg := rand.New(rand.NewSource(time.Now().UnixNano()))
35   for cs := range inCh {
36     // dummy computation
37     x := time.Duration(1e8 + rg.Int63n(1e8))
38     fmt.Println("Worker thread: ", i)
39     time.Sleep(x)
40     result := "DONE: " + cs.input
41     cs.rCh <- result
42   }
43 }
44
45 func dispatch(inCh chan *inputCS, outCh chan chan string) {
46   for _, s := range input {
47     rCh := make(chan string, 1)
48     inCh <- &inputCS{s, rCh}
49     outCh <- rCh
50   }
51   close(outCh)
52 }
```

Listing 3.2: Worker Threads in Parallel Go

```
1   Number of workers: 2
2   Worker thread: 0
3   Worker thread: 1
4   Worker thread: 0
5   DONE: one
6   DONE: two
7   Worker thread: 1
8   Worker thread: 0
9   DONE: three
10  DONE: four
11  Worker thread: 1
12  Worker thread: 0
13  DONE: five
14  DONE: six
15  Worker thread: 1
16  Worker thread: 0
17  DONE: seven
18  DONE: eight
19  Worker thread: 1
20  DONE: nine
21  DONE: ten
22  Success: process exited with code 0.
```

Listing 3.3: Worker Thread Output

Note in the output there are two worker threads, as computed at line 20 in the program. The program was run on an Intel Core2™ Extreme X9100 which has no hyper-threading, so the maximal processors are two. If this program had been run on quad-core processor that had hyper-threading[28], then the number of workers would have been eight. Note the interleaving between tasks and also note the in-order execution. The interleaving shows that thread 1 is always slightly behind thread 0 (an artifact of the pseudo-work at line 39) and thus it is thread 0 that processes most the output; that is until line 51, where thread 0 has its `outCh` channel closed, and then at line 19 in Listing 3.3 thread 1 processes the last two work items. The in-order execution is guaranteed by the FIFO nature of the channels as per the Go language specification, not the sends to the queue, but the order in the queue. In Listing 3.2, function `main()` uses sequence (WCP-1), synchronization (WCP-3), implicit termination (WCP-11), and blocking discriminator (WCP-28). The `worker(···)` function uses sequence (WCP-1), synchronization (WCP-3), and implicit termination (WCP-11). Concluding with function `dispatch(···)` using sequence (WCP-1), synchronization (WCP-3), explicit termination (WCP-43). In the VXDL description on page 92 the XML describes the 40 CPU "Filtering cluster" that the Static Analyzer can use to map the `main()`, `dispatch(···)`, and `worker(···)` threads across multiple processors. The mapping would be in the most loosely coupled manner as there is scant information in how tightly or loosely connected the 40 CPUs are. Additional information would have to be provided as to the `<resource-parameter>`, `<link-parameters>` and `<timeline>` VXDL description constructs to ascertain the true

[28] An Intel Core™ i7 930

computing element capabilities (see Appendix E, lines 57, 74, and 91 respectively.) The Static Analyzer allocation is based on the detailed information provided by the Resource Agents.

3.4.4 DDE Input Data for Algorithm Mapping

RITA DDE cost function theory has already been described in §2.4.2.3 and §2.4.2.4. In Equation 2.22 on page 58 each of the $\mathcal{L}_\mathcal{N}$, $\mathcal{L}_\mathcal{S}$, and $\mathcal{L}_\mathcal{P}$ value calculations are shown. In this section, predictive data — some based on empirical observations — are described and used as values in latency functions for the $\mathcal{L}'$ integral. Actual values from existing long-haul networks, switches, and NoC system and components are described and shown. Special attention is placed on NoC communication as these paths have a high cumulative impact on latency during system operation, not the largest value, but the highest cumulative impact on algorithm performance.

3.4.4.1 Latency in cloud provider systems: Part 1 of $\mathcal{L}_\mathcal{N}$

With the ontological weighted matching from §3.3.4, the RITA DDE Cost Function from page 58 is used to provide an additional discriminant using the expected steady-state solution for the latency optimized mapping of algorithm executables to CEs. To compute the necessary values, system and device data must be known. This data is nominally provided by the Resource Agent NML and VXDL data. Data also is derived from system data sheets and empirical system execution information especially for intra- and inter-cloud network latency. Examining AWS, Google, and RackSpace cloud providers' network latency and throughput using several of the cloud providers' no-cost configurations has been done. Using the `iperf`, `ping`, and `traceroute` utilities for single stream TCP latency, tests were performed by the GigaOM[29] trade group on the three cloud providers' systems. Using these utilities, throughput measurements were taken. Each test was run three times and averaged for each instance pair. The results are shown in Tables 3.8 to 3.10.

Server <*hostname*>: `iperf -f m -s`
Client: `iperf -f m -c <hostname>`

	t1.micro (1 CPU)	c3.8xlarge (32 CPUs)
us-east-1 zone-1a ↔ us-east-1 zone-1a	135 Mbits/sec	7013 Mbits/sec
us-east-1 zone-1a ↔ us-east-1 zone-1d	101 Mbits/sec	3395 Mbits/sec
us-east-1 zone-1a ↔ us-west-1 zone-1a	19 Mbits/sec	210 Mbits/sec

Table 3.8: Amazon Web Services (AWS) `iperf` data transfer

[29] https://gigaom.com/

	f1-micro (shared CPU)	n1-highmem-8 (8 CPUs)
us-central-1a ↔ us-central-1a	692 Mbits/sec	2976 Mbits/sec
us-central-1b ↔ us-central-1b	905 Mbits/sec	3042 Mbits/sec
us-central-1a ↔ us-central-1b	531 Mbits/sec	2678 Mbits/sec
us-central-1a ↔ europe-west-1a	140 Mbits/sec	154 Mbits/sec
us-central-1b ↔ europe-west-1a	137 Mbits/sec	189 Mbits/sec

Table 3.9: Google Compute Engine `iperf` data transfer

	512MB Std. (1 CPU)	120GB Perf.2 (32 CPUs)
Dallas (DFW) ↔ Dallas (DWF)	595 Mbits/sec	5539 Mbits/s
Dallas (DFW) ↔ N. Virginia (IAD)	30 Mbits/sec	534 Mbits/s
Dallas (DFW) ↔ London (LON)	13 Mbits/sec	88 Mbits/s

Table 3.10: Rackspace `iperf` data transfer

In the case of Amazon, the performance was inconsistent as speeds varied dramatically more than with any other provider across the three test runs for all instance types. Inside zone communication uses internal IP addresses and outside a zone uses the Internet with public IP addresses.

With Google there was inconsistent performance for the f1-micro instance for all zone tests. Within the same us-central-1a zone, the first run resulted in 991 Mbits/sec with the next two runs showing 855 Mbits/sec and 232 Mbits/sec respectively. Across regions between the US and Europe, the results were much more consistent, as were all the tests for the higher spec n1-highmem-8 server. On April 2, 2014 Google announced a new networking infrastructure in us-central-1b and europe-west-1a which would later roll out to other zones. There was about a 1.3 times improvement in throughput using this new networking and users should also see lower latency and CPU overhead. Google uses internal IP addresses globally inside and outside its zones and across regions using internal private transit instead of the Internet making it much easier to deploy across zones and regions. Google is known as having one of the fastest networks in the world[30] thus the relatively low network bandwidth between servers on the same zone appeared inconsistent with the resources available. Using the `-p` switch for `iperf` allows parallel processes on separate processors. Separate tests with this switch turned on gave much better throughput results:

Inter-zone, US network	[SUM] 0.0-10.0 sec	9186 MB	7689 Mb/sec
Intra-zone, us-central-1a↔us-central-1b	[SUM] 0.0-10.0 sec	8082 MB	6778 Mb/sec
Inter-zone, us-central-1a↔europe-west1-b	[SUM] 0.0-10.0 sec	3272 MB	2720 Mb/sec
Inter-zone, us-central-1a↔asia-east1-a	[SUM] 0.0-10.0 sec	2032 MB	1690 Mb/sec

[30] http://tinyurl.com/ojqlmou/

Rackspace does not offer the same kind of zone/region deployments as Amazon or Google so no between-zone tests could be run. Tests were done between next closest data centers using the Para-Virtualization Hardware Virtual Machine (PVHVM) platform, which is specific to the Xen hypervisor resulting in better I/O and networking performance. The Xen hypervisor used by Rackspace, first developed at the University of Cambridge Computer Laboratory in 2003, is now a free and open-source software project[31]. Rackspace has account quotas and requires a separate account for the London region.

3.4.4.2 Long-haul latency between data centers: Part 2 of $\mathcal{L}_\mathcal{N}$

As this work is focused on data-center to data-center latency rates, it is important to refer to the global Internet performance work done by PingER (Ping End-to-end Reporting)[32], the common name of the the Internet End-to-end Performance Measurement (IEPM) project monitoring end-to-end performance of Internet links. It is led by SLAC and development includes NUST/SEECS (formerly NIIT), FNAL, and ICTP/Trieste, together with UM, UNIMAS and UTM in Malaysia. The project's original mission starting in 1995 was for the High Energy Physics community, however, it has been more focused on measuring Internet Performance. The project now involves measurements to over 700 sites in over 160 countries. The project will be used to give jitter and loss statistics in §3.4.6.

For dedicated networks, provided by telecommunication companies, data available from Verizon data networks for 2014 has SLAs of Monthly latency:

- 45ms or less for regional round trips within North America
- 30ms or less for regional round trips within Europe
- 90ms or less for transatlantic round trips between London and New York

Packet delivery of:

- 99.5% or greater for regional round trips within Europe and North America
- 99.5% or greater for transatlantic round trips between London and New York

Data provided by Verizon Enterprise Solutions, shown in Table 3.11, shows latency statistics in milliseconds for several domestic, European, trans-Atlantic, and trans-Pacific routes the named links and goal millisecond latency in parenthesis after the link name.

Using the wire line latency, L_{WL}, from §2.4.2.3.3, the nominal distance from New York, USA to London, UK should be 5,560,680.80 meters with a resulting latency of 27.66507859 milliseconds. Given the Trans-Atlantic average of 75.788583333 milliseconds there is a delta of 48.12350441 milliseconds that would be the remainder of the

[31] http://www.xenproject.org/
[32] http://www-iepm.slac.stanford.edu/pinger/

TABLE 3.11: Verizon Long-haul Network Latency

Verizon Enterprise Solutions Latency Statistics (ms)													
2014	Dec	Nov	Oct	Sep	Aug	Jul	Jun	May	Apr	Mar	Feb	Jan	Average (milliseconds)
Trans Atlantic (90.000)	72.538	72.486	75.019	72.845	78.914	78.246	78.663	74.045	72.275	76.771	79.019	78.642	75.788583333
Europe (30.000)	11.781	11.74	11.849	11.677	11.627	11.702	11.647	11.696	11.766	13.602	13.911	13.87	12.239000000
North America (45.000)	36.447	35.834	35.388	35.741	35.665	35.924	35.972	35.859	36.92	37.131	38.264	38.14	36.440416667
Intra-Japan (30.000)	11.454	8.629	8.388	8.818	8.338	8.689	8.289	8.298	8.182	8.385	9.82	10.805	9.007916667
Trans Pacific (160.000)	109.795	109.797	109.762	109.7	109.784	109.654	109.665	109.671	109.669	109.665	110.973	111.757	109.991000000
Asia Pacific (125.000)	114.276	96.065	95.538	97.376	95.899	95.669	95.116	97.16	95.312	96.641	97.384	95.94	97.698000000
Latin America (140.000)	136.883	137.234	142.068	144.063	147.838	150.417	146.642	140.536	140.69	143.051	137.299	137.644	142.030416667
EMEA to Asia Pacific (250.000)	134.449	161.106	142.132	123.219	146.992	158.393	139.635	136.264	161.888	158.436	143.249	142.032	145.649583333

queue, switch, and store and forward latency elements in this network segment. From experimentation with `traceroute` to known locations in London, major network routing latency shows approximately 121 millisecond averages from Cincinnati, USA to London, UK with between 20 to 25 "hops" in the route. There are multiple providers of long-haul cable systems. Using Verizon as an example, a review of the Trans-Atlantic cable routes, Figure 3.15 and Table 3.12, shows that cable paths do not follow a "Great Circle" route for point to point traversals from New York to London.

FIGURE 3.15: Partial Verizon Trans-Atlantic Cable Map

Given this information any data center to data center calculation of L_{WL} would require actual network route empirical data and would require a table look-up of data based on measurements. With data being done with a table look-up, an evaluation of network traversal was done from West Chester, Ohio (my location) to the University of London in the UK. A `traceroute` augmented with geographical data in parenthesis from the DB-IP Database web site (https://db-ip.com/) is shown in Table 3.12. This data shows is that given segment information from a vendor, the actual number of providers, and the actual number of "hops" have to be known to determine data center to data center network latency. In the `traceroute` four providers and an approximate network distance of 11,183,894.73 meters was calculated. Of interest, and not available with

tracing, is the internal Level 3 Communications path to a demarcation point on the Atlantic seaboard side to of the USA as this would add additional paths and distance.

3.4.4.3 NoC Latency internal to CEs: Calculating $\mathcal{L}_{\mathcal{S}}$

Looking at NoC values, the comparison of magnitude of latency must be kept in mind. In Table 3.13[33] the orders of magnitude of difference between actions that produce latency is shown. Not only should the action times be considered, but the frequency of each action must be properly accounted for to construct a minimal latency calculation.

```
>tracert www.london.ac.uk
Tracing route to www.london.ac.uk [128.86.130.195]
over a maximum of 30 hops:
 1     4 ms    <1 ms    <1 ms  192.168.1.1
 (Private network)
 2    27 ms     8 ms    13 ms  cpe-98-28-224-1.woh.res.rr.com [98.28.224.1]
 (Internet Assigned Numbers Authority, New York, NY)
 3    12 ms    12 ms    12 ms  24.29.4.17
 (Time Warner Cable, Columbus, OH)
 4    13 ms    11 ms    15 ms  tge1-8-0-10.blasohdp01r.midwest.rr.com [65.29.37.92]
 (Time Warner Cable, Columbus, OH)
 5    16 ms    15 ms    19 ms  be28.clmkohpe01r.midwest.rr.com [65.29.1.44]
 (Time Warner Cable, Cincinnati, OH)
 6    28 ms    27 ms    27 ms  bu-ether35.chctilwc00w-bcr00.tbone.rr.com [107.14.19.60]
 (Time Warner Cable, Kansas City, KS)
 7    25 ms    25 ms    25 ms  0.ae0.pr0.chi30.tbone.rr.com [66.109.1.76]
 (Time Warner Cable, Herndon, VA)
 8     *        *        *     Request timed out.
 9   115 ms   114 ms   122 ms  ae-4-4.car1.Manchesteruk1.Level3.net [4.69.133.101]
 (Level 3 Communications, Paris, TX)
10   113 ms   124 ms   114 ms  ae-4-4.car1.Manchesteruk1.Level3.net [4.69.133.101]
(Level 3 Communications, Paris, TX)
11   112 ms   111 ms   111 ms  195.50.119.98
(Level 3 (was Businessnet), Camden Town, London, UK)
12   116 ms   114 ms   114 ms  ae29.erdiss-sbr1.ja.net [146.97.33.41]
(Janet, Camden Town, Greater London, UK)
13   116 ms   114 ms   120 ms  ae31.londpg-sbr1.ja.net [146.97.33.21]
(Janet, Camden Town, Greater London, UK)
14   118 ms   117 ms   115 ms  be24.londic-rbr1.ja.net [146.97.37.198]
(Janet, Camden Town, Greater London, UK)
15   114 ms   117 ms   114 ms  be2.londsh-rbr1.ja.net [146.97.66.33]
(Janet, Camden Town, Greater London, UK)
16   120 ms   125 ms   121 ms  ulcc-1.ja.net [146.97.137.54]
(Janet, Camden Town, Greater London, UK)
17   124 ms   134 ms   124 ms  fw.ulcc.net [128.86.200.178]
(Janet, Camden Town, Greater London, UK)
18     *        *        *     Request timed out.
19     *        *        *     Request timed out.
20     *        *        *     Request timed out.
21     *        *        *     Request timed out.
22     *        *        *     Request timed out.
23     *        *        *     Request timed out.
24     *        *        *     Request timed out.
25     *        *        *     Request timed out.
26     *        *        *     Request timed out.
27     *        *        *     Request timed out.
28     *        *        *     Request timed out.
29     *        *        *     Request timed out.
30     *        *        *     Request timed out.
Trace complete.
```

TABLE 3.12: Traceroute from West Chester, Ohio to University of London, England

[33] Jeff Dean: http://research.google.com/people/jeff/, original by Peter Norvig: http://norvig.com/21-days.html#answers

TABLE 3.13: Comparison of Latency Time Magnitudes

Access Type	Time (ns)	Time (ms)	Comparison
L1 cache reference	0.5		
Branch mispredict	5.0		
L2 cache reference	7.0		14x L1 cache
Mutex lock/unlock	25.0		
Main memory reference	100.0		20x L2, 200x L1 cache
Compress 1K bytes to Zip file	3,000.0		
Send 1KB over 1 Gbps network	10,000.0	0.01	
Read 4KB randomly from SSD*	150,000.0	0.15	
Read 1 MB sequentially from memory	250,000.0	0.25	
Round trip within same data center	500,000.0	0.50	
Read 1 MB sequentially from SSD*	1,000,000.0	1.00	4X memory
Disk seek	10,000,000.0	10.00	20x datacenter roundtrip
Read 1 MB sequentially from disk	20,000,000.0	20.00	80x memory, 20X SSD
Send packet CA → Netherlands → CA	150,000,000.0	150.00	

Notes: 1ns = 10^{-9} seconds, 1 ms = 10^{-3} seconds, * Assuming 1GB/sec SSD

Calculation of NoC latency requires the actual network used. Specific modeling thus requires detailed information from each vendor. As listed below there are many topologies. For this work, the canonical forms and extensions of NoC are shown in Figures 3.16 to 3.21 are used.

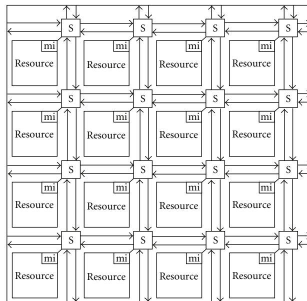


FIGURE 3.16: Mesh NoC Topology

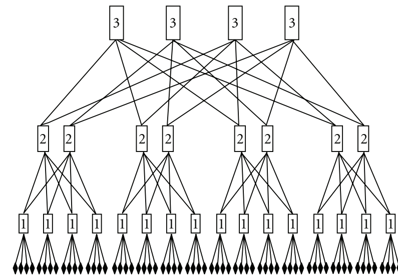


FIGURE 3.18: Butterfly Fat NoC Topology

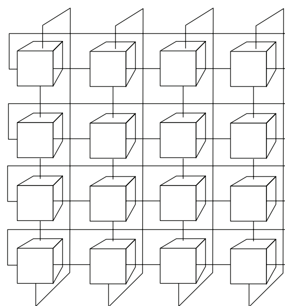


FIGURE 3.17: Torus NoC Topology

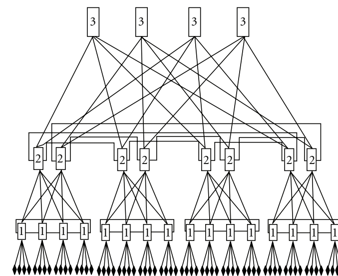


FIGURE 3.19: Butterfly Fat Extension NoC Topology

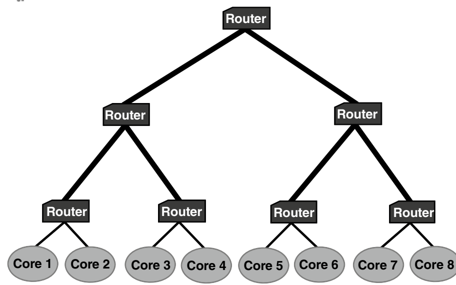


FIGURE 3.20: Fat-Tree NoC Topology

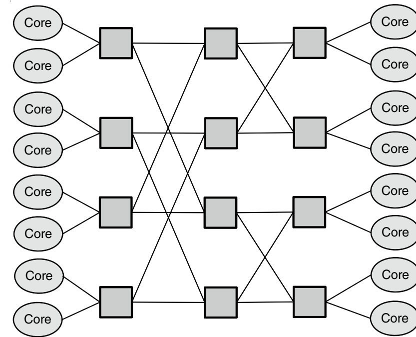


FIGURE 3.21: 3-Stage Butterfly NoC Topology

There are many other specialized topologies as well. In Cota, et.al. [CLdMA12] several topologies are described that augment these topologies.

- In [GL85], Greenberg and Leiserson introduced a fine structure which Greenberg calls a butterfly fat-tree in his thesis [Gre89]. Currently "fat-tree" almost always refers to a butterfly fat-tree. In a butterfly fat-tree, every link connects a port on a switch at a parent node to a port on a switch at a child node (i.e., there are no links between switches at the same tree node) with at most one link between any two switches and each switch at a parent node is connected to exactly one switch at each of the children. An interesting note is that a butterfly fat-tree is not actually a "tree" as it contains cycles, but its coarse structure is a tree.
- A mesh topology is either a full mesh or partial mesh. A full mesh is where a node is connected to every other node in the network and is a high cost method to connect buses. A partial mesh is where a node does not have to be directly connected to all other nodes, thus it has a lower cost and also less redundancy. A 2D-array is a type of mesh where each node is connected to the four adjacent neighbor routers. The routers at the edges have only two or three connections since they do not have more adjacent routers. The number of nodes will then become C×R, where C is columns and R is rows.
- A torus topology is similar to the 2D-array where all routers have four connections since with a wrap-around from North-South and East-West edges.
- A star topology uses a central hub to which all resources are connected. All communication between resources is then passed through the central hub.
- A ring topology when the resources are connected to each other in a ring. An octagon network is a cross between a start and ring. Every resource is then connected to its two neighbors communication with other resources then has to pass through the neighbors.
- A bus topology has several resources using the same communication channel.

- The binary tree topology has a root node that is connected to one or more nodes of a lower hierarchy. In a symmetrical hierarchy, each node in the network has a specific fixed number of nodes connected to those at a lower level.
- Scalable, Programmable, Integrated Network (SPIN) topology is one of the first proposed NoC architectures developed at LIP6. It presents a fat-tree topology with wormhole switching, deterministic and adaptive (deflective) routing, input buffering and two shared output buffering. It uses 36-bit links (32 data bits plus 4 control bits). It also implements the virtual component interface (VCI) socket in the network interface.
- An ÆTHEREAL topology is a best-effort and a guaranteed throughput NoC implemented in a synchronous indirect topology with wormhole switching, and contention-free source routing algorithm based on time-division multiplexing . It implements a number of connection types including narrowcast, multicast, and simple connection and the network adapter can be synthesized for four standard socket interfaces (master or slave, OCP, DTL, or AXI based).
- STNoC – Proposed by ST Microelectronics is a guaranteed service NoC "spidergon"/ring topology with minimal path, 32-bit links, and input buffering resulting in an efficient performance NoC.
- Nostrum – In this guaranteed bandwidth NoC where the protocol includes the data encoding to reduce power consumption. It is implemented in a 2D mesh topology with hot potato routing. Links are composed of 128 bits of data plus 10 bits of control. Virtual channels and a TDM mechanism are used to ensure bandwidth.
- XPIPES – This NoC presents an arbitrary topology, tailored to the application to improve performance and reduce costs. XPipes consists of soft macros of switches and links that can be instantiated during synthesis. The standard open-core protocol is used in the network interface.
- SoCin – SoCin NoC has been proposed by Universidade Federal do Rio Grande do Sul (UFRGS) as a simple, parameterized best-effort NoC implemented in 2D mesh or torus topology, with narrowcasting routing, input buffering, and parameterized channel width.
- QNoC – Developed at Technion in Israel, this direct NoC is implemented in an irregular mesh topology with wormhole switching and XY minimal routing scheme. Four different classes of traffic are defined to improve QoS, although hard guarantees are not given.
- HERMES – This NoC was proposed by Pontifícia Universidade Católica do Rio Grande do Sul and implements a direct 2-D mesh topology with wormhole switching and minimal XY routing. Hermes is a best-effort NoC with parameterized input queuing.

- MANGO – Developed at Technical University of Denmark; this NoC implements a message-passing asynchronous (clock-less) protocol with guaranteed services over open-core protocol interfaces. Mango also provides best-effort services using credit-based and source routing.

Despite the variability in the design decisions, one can map some common design trends among available NoCs: most implementations use packet switching for communications for its efficiency; most NoCs use 2D mesh topologies because of the good trade-off between cost and performance; XY routing is very common for mesh topologies, although not standard, due to its property of being deadlock-free; most NoCs use input buffering only, again because of the trade-off between cost and performance gain. Revisiting Equation 2.16 with consideration to the topologies discussed on page 117, the packet creation and ejection times, ts_1 and ts_2 respectively, are measured in flits/node/cycle. This work reviews and uses the data from experiments done by Hanjoon Kim, Gwangsun Kim, et.al.[KKM+14], Rachata Ausavarungnirun, Chris Fallin, et.al. [AFY+14], Benjamin Johnstone [Joh14], and Jain, Parikh, and Bertacco [JPB14] for emperical data values.

From Kim and Kim [KKM+14], Figure 3.22 shows the average hop count for the different topologies they considered using uniform random traffic with minimal routing. The hierarchical ring (HRING) reduces the hop count compared with the ring topology (shown as RING in figure) but it is still higher than alternative topologies. For example, for 64 nodes, hierarchical ring reduces the hop count by 52.5% compared with ring but it is still 43% higher than that of the 2D mesh topology and 5.1 times higher than 2D flattened butterfly (FBFLY). The times are: T_h, the header latency, and T_s, the serialization latency.

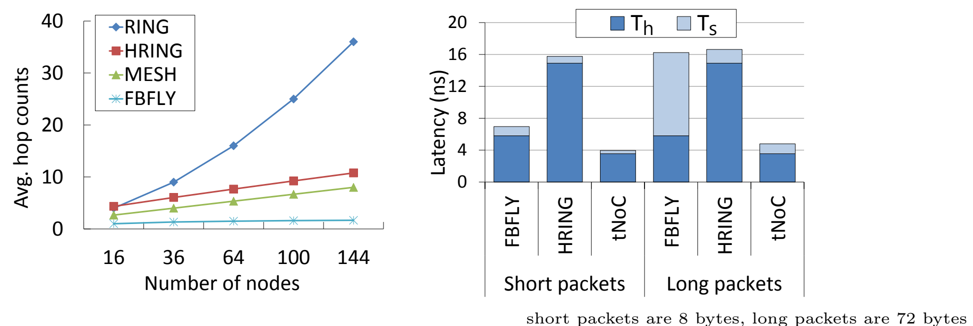


FIGURE 3.22: Kim and Kim latency comparison

Latency breakdown of the different topologies is shown in Figure 3.23, where latency is divided into T_h, T_s, as well as contention latency T_c (or queuing latency in the network) components. The `CREDIT & INTM.` value was developed by Kim and Kim for their tNOC

method as the latency of acquiring a credit and the queuing latency in the intermediate buffer of the hub router. This credit-based data is not used in this work.

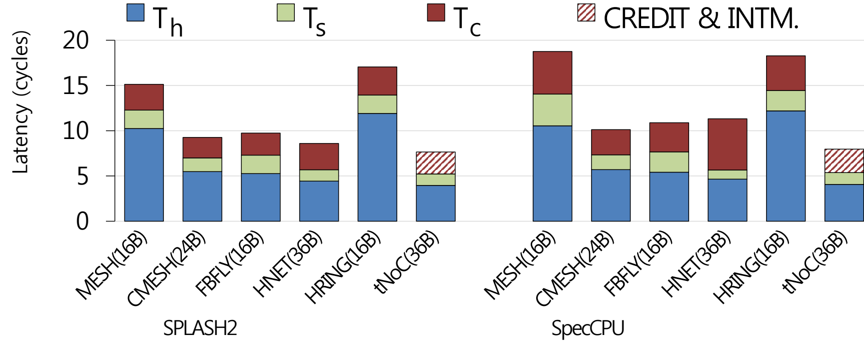


FIGURE 3.23: Kim and Kim SPLASH2 and SpecCPU Benchmark Topology Latency

Given the zero-load latency for a 64-node network shown in Figure 3.23, based on the per-hop router latency from Table 3.14, figure 3.24 shows performance based on the load level in the network for 2, 4, and 8 virtual channels. Thus, different micro-benchmarks have to cover different load levels of background traffic by controlling the traffic emission rate to the network — a central tenet of the RITA condition-event matrix. The following cases should be covered given as a fraction of the ideal throughput, Θ-ideal of the network, i.e 10%, 30%, 50%, 70%, or 90% efficiency. The ideal throughput Θ-ideal is the maximum throughput that a network could carry with perfect flow control and routing as described in Dally and Towles [DT03] and depending on network topology and traffic pattern. Shown are two methods of test sampling for latency: Launch On Capture (LOC) and Uniform Random (UR). LOC based testing is when the first n-bit vector is scanned into the circuit with n scan flip-flops at a slow speed, followed by another clock that creates the transition. An at-speed functional clock is applied that captures the response; thus only one vector has to be stored per test and the second vector is directly derived from the initial vector by pulsing the clock. In synthetic traffic, the source and destination node patterns are typically driven by a stochastic UR injection process. The spatial characteristics of the source and destination node patterns and the temporal characteristics of the UR injection process are intended to model the characteristics of realistic workloads.

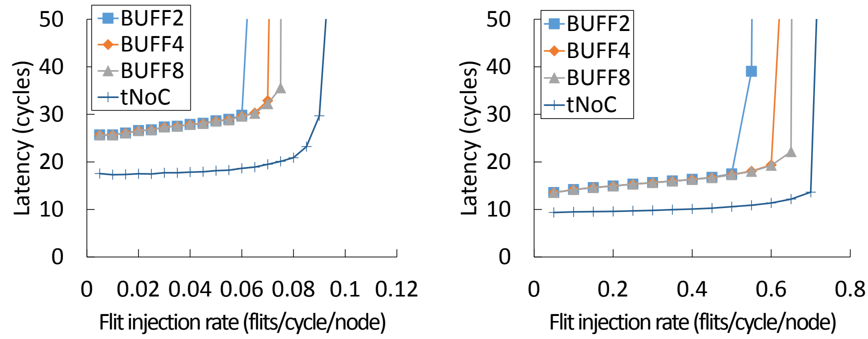


FIGURE 3.24: Kim and Kim, UR (left), LOC at 90% traffic (right)

TABLE 3.14: NoC Latency Paramenters (multiple units)

	MESH	CMESH	FBFLY	HNET	HRING
Ports	5	8	10	12	3
Message Class[a]	3	3	3	3	3
VCs/Class	4	4	4	4	4
Buffer Depth	8	8	12	8	8
Critical Paths (ns)	0.99	1.10	1.16	1.22	0.86
Router Delay[b]	2	2	2	2	2
Channel Delay[b]	1	1	0.5/tile	1	1

[a]Deadlock prevention: request, snoop, response. VC=virtual circuit. Buffer depth in flits. [b] cycles.

Ausavarungnirun, Fallin, et.al. define a network — Hierarchical Rings with Deflection (HiRD) — built on five operation principles:

1. Every node (e.g., CPU, cache slice, or memory controller) resides on one local ring, and connects to one node router on that ring.
2. Node routers operate exactly like routers (ring stops) in a single-ring interconnect: locally-destined flits are removed from the ring, other flits are passed through, and new flits can inject whenever there is a free slot (no flit present in a given cycle). There is no buffering or flow control within any local ring; flits are buffered only in ring pipeline registers. Node routers have a single-cycle latency.
3. Local rings are connected to one or more levels of global rings to form a tree hierarchy.
4. Rings are joined via bridge routers. A bridge router has a node-router-like interface on each of the two rings which it connects, and has a set of transfer FIFOs (one in each direction) between the rings.
5. Bridge routers consume flits that require a transfer whenever the respective transfer FIFO has available space. The head flit in a transfer FIFO can inject into its new ring whenever there is a free slot (exactly as with new flit injections). When a flit requires a transfer but the respective transfer FIFO is full, the flit remains in its current ring. It will circle the ring and try again next time it encounters the correct bridge router (this is a deflection).

By using deflections rather than buffering and blocking flow control to manage ring transfers, HiRD retains node router simplicity, unlike past hierarchical ring network designs. This change comes at the cost of potential livelock (if flits are forced to deflect forever).

With the ability to have a flit per cycle, without the need for buffering, it would seem to obviate the need to account for NoC latency. In the work by Johnstone, based on works done by Aamodt, Wilson, Fung, et.al. [AFB12][AF15] and based on the the work by Ausavarungnirun, et.al. [AFY+14], it would appear that flit size would have a continual linear improvement in performance. Using Johnstone's work, where the flit size was increased to 1024 bytes to simulate an "infinite" bandwidth, that is, the bandwidth between cores and DRAM was not a performance bottleneck. Figure 3.25 shows the speedup of benchmarks from Johnstone's work, Section §3.2.

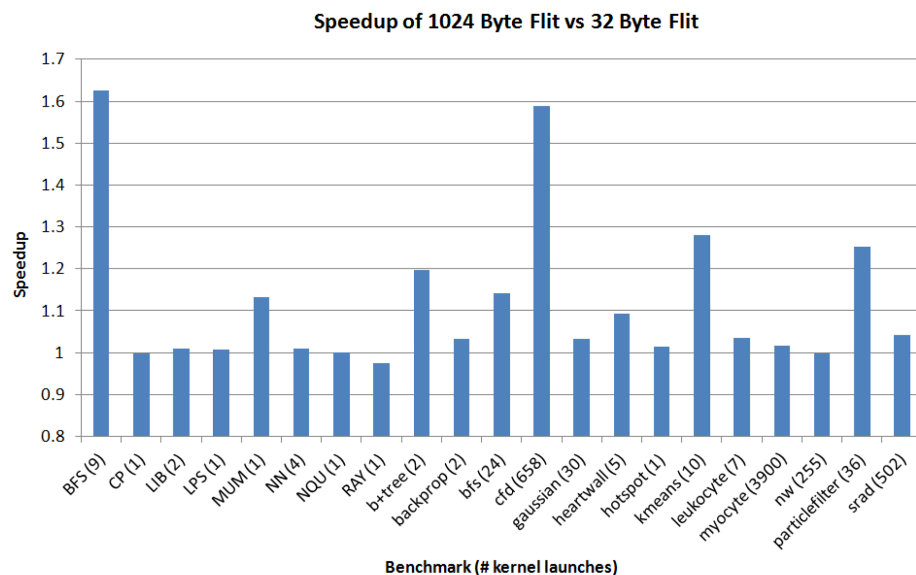


FIGURE 3.25: Benchmarks used for flit performance

In Figure 3.26 Johnstone shows that there is a diminishing return on flit size across all benchmarks beyond 128 bytes per flit. Thus, over-provisioning flit bandwidth will be a waste of resources and power consumption would increase without significant improvement in performance.

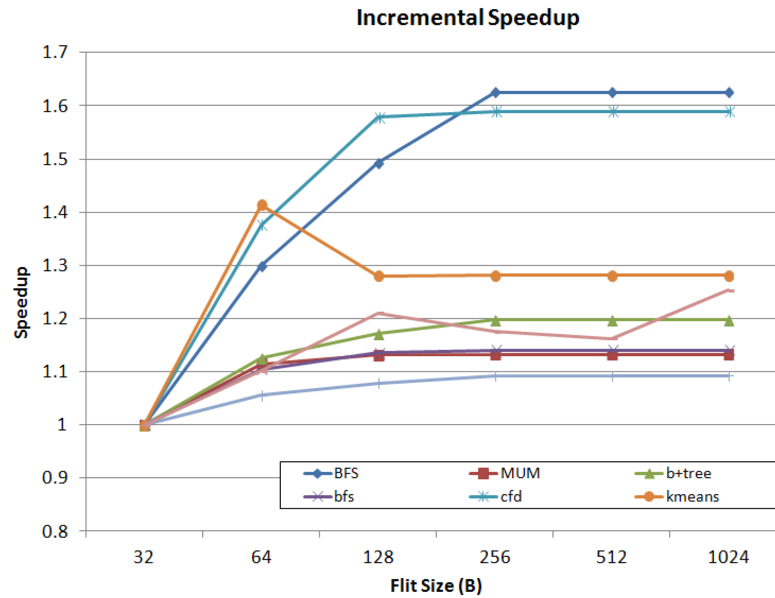


FIGURE 3.26: Speedup Versus Flit Size for Selected Benchmarks

Next, the introduction of nearest neighbor limitation to NoC performance is provided from the work by Jain, Parikh, and Bertacco. In Figure 3.27 they graph the high usage of source-destination pairs. The graph shows that references are highly localized (>60%) by incident. The curve in Figure 3.27 rapidly reduces slope and plateaus. The goal in their work was identification of pairs transporting the majority of traffic to minimize the pair hop count in their design. The measure is referred to as a Frequently Communicating Pair (FCP). In Figure 3.28 the plot compares the average network latency for three topologies under directed traffic with increasing injection rate for the FCPs.

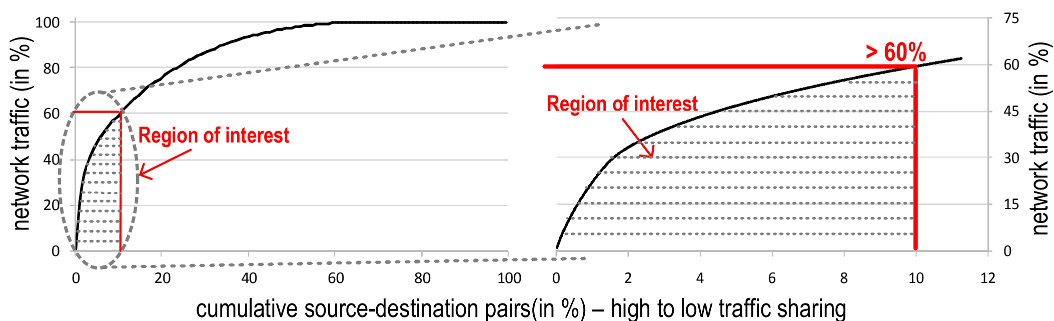


FIGURE 3.27: Network activity by most exercised source-destination pairs

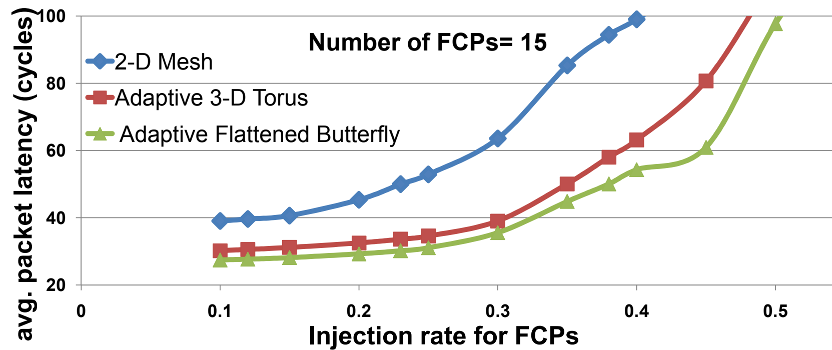


FIGURE 3.28: Average network latency with varying number of FCPs

In Figure 3.29 the efficiency decreases beyond 25 FCPs and the optimal number of FCP entries varies depending on the network load and congestion.

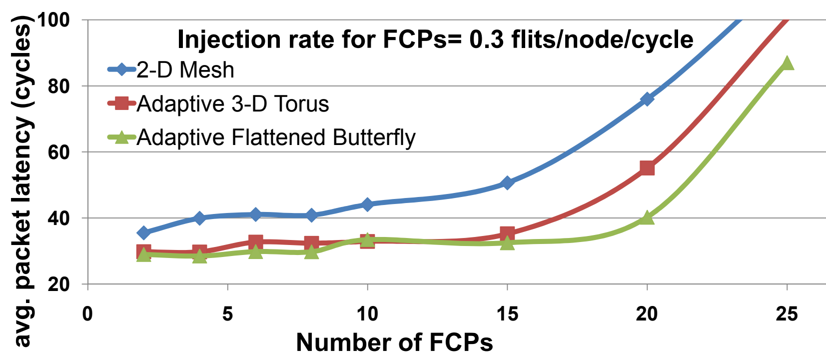


FIGURE 3.29: Average network latency under directed traffic

3.4.4.4 Heterogeneous Computing Element Compilers; Computing $\mathcal{L}_{\mathcal{P}}$

If the OpenCL heterogeneous compilation environment is used, the auto-parallelization feature automatically translates serial portions of the input program into equivalent multi-threaded code. Automatic parallelization determines the loops that are good work-sharing candidates, performs the data-flow analysis to verify correct parallel execution, and partitions the data for threaded code generation as needed in programming with OpenMP* directives[34]. The OpenMP* and auto-parallelization features provide the performance gains from shared memory on multiprocessor and dual core systems. See Appendix G for sample output for $\mathcal{L}_{\mathcal{P}}$ data from the OpenMP* compiler.

The IBM Liquid Metal effort has produced the Lime language [ABCR10], based on a super-set of Java. The programmer uses Lime language structures to create parallel

[34]OpenMP* is a high level, pragma-based approach to parallel application programming. Cluster OpenMP is a simple means of extending OpenMP parallelism to 64-bit Intel architecture-based clusters. It allows OpenMP code to run on clusters of Intel Itanium™ or Intel 64™ processors, with only slight modifications.

constructs in the Lime language. A central design decision in developing Lime by Auerbach, et.al. was making it Java compatible. Most legal Java programs are legal Lime programs. All legal Java programs can be imported with only syntactic transformation. Lime reserves twelve additional keywords. If these are present in a Java program they can be annotated to escape from Java processing by use of a " ` " (backtick) character. Generic types and methods have expanded semantics in Lime, with the original Java semantics obtained by use of the " ~ " (tilde) character. Lime programs are binary compatible with Java and can invoke Java methods. If Lime-specific types are avoided in public signatures, Java can call Lime methods. The auto-parallelizer analyzes the dataflow of the loops in the application source code and generates multi-threaded code for those loops which can safely and efficiently be executed in parallel. This behavior enables the potential exploitation of the parallel architecture found in symmetric multiprocessor systems(SMP) systems. Thus the specific code elements can be counted for computing the serial and parallel constructs in the source program.

In [RYC+13], Rossbach, et.al. from Microsoft, describe Dandelion; a compiler and run-time environment for heterogeneous computing. Dandelion is a system designed for data-parallel applications. Dandelion provides a unified programming model for heterogeneous systems spanning diverse execution contexts including CPUs, GPUs, FPGAs, and the cloud. It adopts the .NET LINQ (Language INtegrated Query) approach, integrating data-parallel operators into general purpose programming languages. It therefore provides an expressive data model and native language integration for user-defined functions, enabling programmers to write applications using standard high-level languages and development tools.

Dandelion addresses writing data-parallel applications for heterogeneous systems by adopting a "single machine" abstraction for target systems of compute clusters of multicore CPUs and GPUs. Dandelion assumes a cluster consists of a small number of moderately powerful computers. Such a cluster can easily have aggregated compute resources of more than 100,000 cores and 10TB of memory for such large-scale systems such as machine learning and computational biology. The programmer writes sequential code in a high-level programming language with the compilation system executing it using all the parallel compute resources available in the execution environment.

To support data-parallel computation, Dandelion embeds a set of data-parallel operators using the LINQ language integration framework. LINQ is a .NET framework for language integration. It introduces a set of declarative operators to manipulate collections of .NET objects. The operators are integrated into high-level .NET programming languages, giving developers direct access to all .NET libraries and user-defined application code. Collections manipulated by LINQ operators can contain objects of any

.NET type, allowing computations with complex data, e.g. vectors, matrices, and images. LINQ operators perform transformations on .NET data collections, and LINQ queries are computations formed by composing these operators. Most LINQ operators are familiar relational operators including projection (*Select*), filters (*Where*), grouping (*GroupBy*), aggregation (*Aggregate*), and join (*Join*). LINQ also supports set operations such as union (*Union*) and intersection (*Intersect*). These constructs also have a high correlation to the WCP patterns in Table 3.3 helping identify which WCP patterns in source software are candidates for parallel processing.

To enable execution across distributed heterogeneous systems without any modification the LINQ and .NET programming model uses a set of Dandelion specific extensions integrated into the LINQ programming model as user-defined operators and language attributes. Dandelion extends LINQ with three new operators. The first operator is *source.AsDandelion(*`<target>`*Type)* which turns the LINQ collection source into a Dandelion collection enabling any LINQ query using it as input to be executed by Dandelion. The second operator added in Dandelion is *source.DoWhile(body, cond)*, a do while loop construct for iterative computations. The arguments body and cond are both Dandelion query functions, and DoWhile repeatedly executes body until cond is false. The third operator is *source.Apply(f)*, which is semantically equivalent to function *f(source)* with its execution being deferred. At the cluster level, the input data is partitioned across the cluster machines, and the function *f()* is applied to each of the partitions independently in parallel. At the machine level, the function *f()* runs on either a CPU or other computing element, depending on its implementation. The primary use of *source.Apply(f)* is to integrate existing CPU and GPU libraries such as the nVidia CUDA Toolkit 4.0 CUBLAS Library and Intel Math Kernel Library (MKL) into Dandelion making the primitives defined in those libraries accessible at the programming API level. The Dandelion compiler relies on a library of generic primitives to construct the execution plans and a cross-compiler to translate user-defined types and lambda functions from .NET to GPU code. This compilation step takes .NET byte-code as input and produces CUDA source code as output. As in Lime, the specific code elements can be counted for computing the serial and parallel constructs in the source program.

A non-heterogeneous compiler that has potential to become one is the Intel Parallel Studio announced in November 2014, with a stable shipping release of February 2015. The Intel Parallel Studio XE 2015 provides OpenMP, Cilk, and MKL support for Intel processors.[35] This compiler is mentioned here to identify it as a potential, future tool for $\mathcal{L}_{\mathcal{P}}$ calculations.

[35] `https://software.intel.com/en-us/intel-parallel-studio-xe`

3.4.5 Consideration of Software Defined Networks for $\mathcal{L}_{\mathcal{N}}$

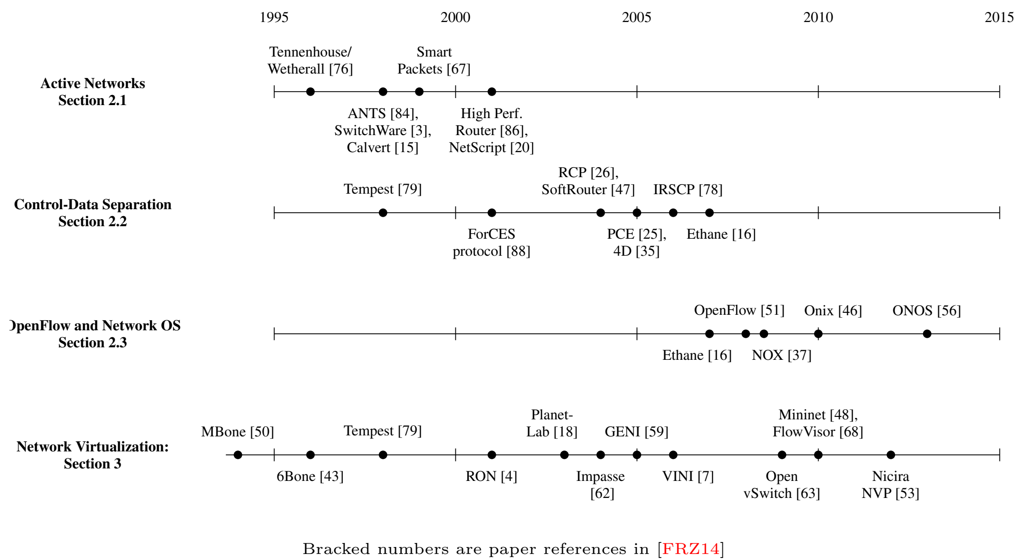
In this work cloud environments are assumed to have network components that are static entities. That is, a network control device is some form of a single function device (i.e. bridge, router, switch, or variant) that is either simple or complex as the need demands. While this static device may be true of corporate, government, university systems, and small PaaS vendors these systems are seen as "fragile" as the installed hardware networks are ridiculously difficult to modify. And when making a change to a network to support a different topology there is no choice but to physically rearrange the hardware. For a cloud computing service this can not be tolerated. Since the 1990's there has been progress toward a programmed virtualized network to overcome this fragile structure. In Feamster, Rexford, and Zenura [FRZ14], there is a good history of the — now common term — Software Defined Network (SDN). SDN is currently a "buzz word" that has a fluid definition based on which vendor and product is being considered for a programmed virtualized network. In Figure 3.30 the time-line by Feamster, et.al. shows the recent efforts in developing programmed virtualized network technology. The history is divided into three stages with each stage having its own contributions. Active networks (mid-1990s to early 2000s) introduced programmable functions in the network; Control and data plane separation (circa 2001 to 2007) developed open interfaces between control and data planes; Resulting in the OpenFlow API and network operating systems (2007 to circa 2010) representing the first instance of widespread adoption of an open interface and developed ways to make control-data plane separation scalable and practical.

This technology is now hotly pursued by cloud system providers as faster and more flexible networks allow rapid reconfiguration for resource allocation to customer needs. In short, the fastest networks, with the quickest reconfiguration wins the customer business. Google uses its own variant of Quagga open source software along with OpenFlow for a SDN called "B4," Amazon has a customized version of Xen, and VMWare has bought Nicira (August 2012) with its customized versions of OpenFlow, Open vSwitch (OVS), and OpenStack. Microsoft cloud has a VMWare Hyper-V integrated solution for its Azure cloud system.

An additional note on Google, Google is known to have extensive investments in separate high speed networks and the WAN at Google has two backbones. The first is called "I-Scale" for Internet user traffic that is usually smooth and diurnal, requiring high availability and loss sensitivity. The second backbone is called "G-Scale" for data center internal traffic that has high volume and can tolerate higher loss and has less stringent high-availability requirements. G-Scale handles most of the server-to-server or VM-to-VM (known as "east-west" in data center jargon) traffic that is growing at a much higher speed than the traffic going from the server to the data center core, to the

campus core, and on to the campus-wide network or Internet (known as "north-south") user traffic handled by I-Scale. Google implemented B4 to have a logically centralized traffic-engineering controller, which allows applications to manipulate bandwidth across data centers through WAN networks.

The RITA DDE equation has no dependency on the logical control of the network as all variables are set by the subject network, so a SDN or a physical network device is only differentiated by routing tables and not physical devices. SDNs still use vendor supplied APIs for the same physical equipment.



Bracked numbers are paper references in [FRZ14]

FIGURE 3.30: Programmable networking chronological relationship to advances in network virtualization.

3.4.6 Example RITA DDE Operational System

An example operational scenario is illustrative of the RITA DDE and shows the calculations of $\mathcal{L}_{\mathcal{N}}, \mathcal{L}_{\mathcal{S}}$, and $\mathcal{L}_{\mathcal{P}}$ from the empirical data provided in this section. This data is normally provided by the Resource Agents during operations and the output of the Compiler and Linker during application executable creation. In this scenario the system assumptions are:

1. Four globally distributed data centers, see Figure 3.31.
   (a) Each data center has its neural network processing done on a multicore system using a continuous-time recurrent neural network.
   (b) The neural network code executes in three VMs on one physical machine. The neural network subsystem communicate with each other between the VMs. The neural network subsystem communicates with data acquisition nodes on other VMs on other physical machines within the region.

(c) There is one neural network processing instance per data center communicating with peer neural networks at other data centers so that each data center has knowledge of peer data centers allowing eventually consistent [Vog09] results of predictive data.

2. Each data center has a minimum time requirement to communicate with each peer data center for round-trip times. This can also be stated as a maximal communication latency between peer data centers.
3. Communication internal to a data center and between data center peers is on a continuous, as needed basis. There is no batch processing of data. Communication is "bursty" on an aperiodic basis thus in a short-term epoch it is effectively a random distribution that has Gamma distribution.

For this example, the localized processing of spot prices is done as described in Wallace, Turchenko, Sheikhalishahi, et.al. [WTS+13] but in this example the neural netwok is modified to have a continuous-time recurrent neural network (CTNN). The original feed-forward neural-network is replaced by a continuous one from Beer [Bee95]:

$$\dot{y}_i = \frac{1}{\tau_i}\left(-y_i + \sum_{j=1}^{N} w_{ji}\sigma\left(y_j + \Theta_j\right) + I_i\right) \quad i = 1, 2, \ldots, N \tag{3.3}$$

τ_i	Time constant of post-synaptic node
y_i	Activation of post-synaptic node
y_j	Activation of pre-synaptic node
$\dot{y}_i$	Rate of change of activation of post-synaptic node
w_{ji}	Weight of connection from pre- to post-synaptice node
$\sigma(x)$	Sigmoid of x, e.g. $\sigma(x) = \frac{1}{(1+e^{-x})}$
Θ_j	Basis of pre-synaptic node
I_i	Constant external input to node

This gives a continually predictive system using streaming data at each data center. The presumption on the overall system is that it has no aberrant training and thus the variance of the predictions against empirical data is acceptable over time. This is assumed as the DDE calculation is the topic of investigation. In addition, each site has an elastic ability to create service instances as described in §3.3.3.2 for each type of virtual machine instance that is determined by the heterogeneous system targeting compilers. The intent here is to give an analog of the top-tier cloud providers, focusing on the calculable latency prior to controlling the communication using the RITA language described at the end of this section.

Using the data from the AWS, Google Compute Engine, and Rackspace from Tables 3.8, 3.9, and 3.10 respectively, these data are intra- and inter-zone throughput for vendor data center sites. The example system presumes a high data rate long-haul network for

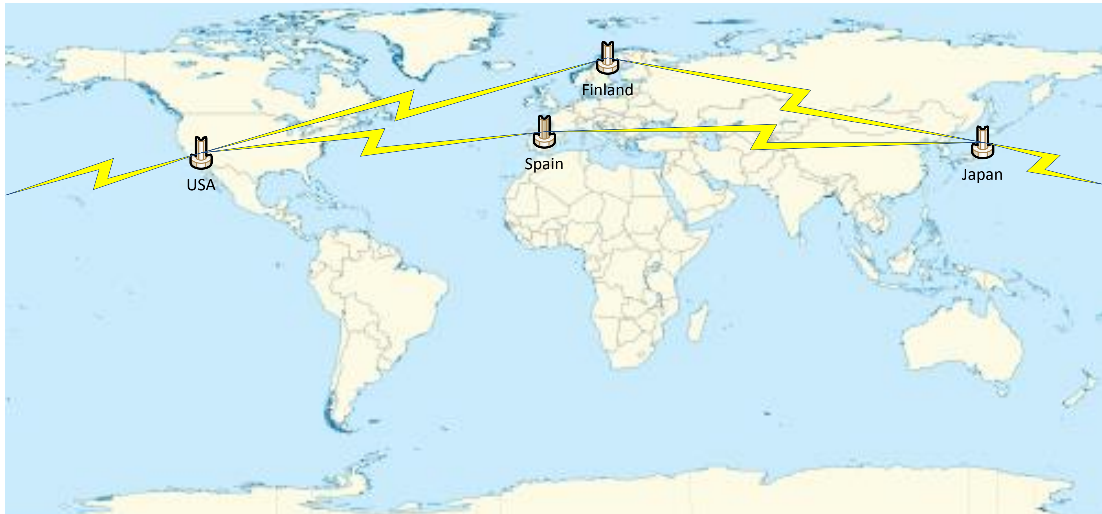


FIGURE 3.31: Overview of Long-haul Example DDE System

each vendor. This involves using a large TCP window over "long fat networks," (LFN)[36]. These are network paths with a large bandwidth-delay product with a large TCP receive buffer allowing these hosts to receive larger windows of data by advertising a large TCP receive window; Nominally referred to as "the window," since there is no equivalent "send window."

The TCP window itself is a 16 bit value at bytes 15–16 in the TCP header. In TCP, as originally specified, the window is limited to a value of 65,535 (nominally 64K bytes). The receive window sets an upper limit on the sustained throughput achievable over a TCP connection since it represents the maximum amount of unacknowledged data (in bytes) there can be on the TCP path. Mathematically, achievable throughput can never be more than $\frac{WindowSize}{RoundTripTime}$. For a trans-Atlantic link, with an Round Trip Time (RTT) of 150ms, the throughput is limited to a maximum of 3.4Mbps. With the emergence of LFN the limit of 64K bytes, with some systems at just 32K bytes, was clearly insufficient and so RFC 7323 detailed a method of scaling the advertised window so that the 16-bit window value can represent numbers larger than 64K bytes.

RFC 7323, *TCP Extensions for High Performance*, dated September 2014 is a proposed standard and supersedes RFC 1323 which defines several mechanisms to enable high-speed transfers over LFNs: Window Scaling, TCP Time-stamps, and Protection Against Wrapped Sequence numbers (PAWS).

The TCP window scaling option increases the maximum window size from 64KB to 1GB through the initial *handshake* portion of the protocol. The TCP transmitter announces a scaling factor — a power of two between 2^0 (no scaling) and 2^{14} (full scaling) — allowing an effective window of up to 2^{30} (one Gigabyte). Window scaling

[36]Common name for networks with high Bandwith-Delay Product (BDP) values: $BDP = \frac{RTT \times IBR}{MTU}$

only comes into effect when both the transmitter and receiver in the connection advertise the option, if only just the scaling factor of 2^0. The window scale option is used only during the TCP 3-way handshake (both sides must set the window scale option in their SYN segments if scaling is to be used in either direction).

It is important to use the TCP *timestamps* option with large TCP windows. With this option, each segment contains a time-stamp. The receiver returns that time-stamp in each ACK and this allows the sender to estimate the RTT. On the other hand with the TCP *timestamps* option, the problem of wrapped sequence numbers can be solved (i.e. PAWS – Protection Against Wrapped Sequences) which could occur with large windows.

One of the issues with this type of network is that it can be challenging to achieve high throughput for individual data transfers with small memory VM systems (Linux and variants or MS Window-based). LFNs are thus a main focus of research on high-speed improvements for TCP (See Appendix H).

The system has possible $\mathcal{L}_{\mathcal{N}}$, part 1, cloud provider internal latencies on the order of less than 1ms based on latency rates from Table 3.13. In the case of Table 3.8 for the *us-east-1 zone-1*↔*us-east-1 zone-1* AWS Instance/Availability Zone, the TCP buffer is calculated with the formula for TCP throughput:

$$\frac{TCPWindowSizeInBits}{LatencyInSeconds} = BitsPerSecondThroughput \tag{3.4}$$

Solving for *LatencyInSeconds* results in a TCP window scale of 2^{14} and a multiple window count of 16 windows to achieve less than 1ms values. This, of course, is ludicrous use of system resources. The actual mechanism is a shared memory zone between the two VM systems[37] supported by the VM controller host system. This is an optimization the VM controller software provides, especially internal to an Instance/Availability Zone. Continuing to the *us-east-1 zone-1a*↔*us-west-1 zone-1a* transfer, which is more realistic as the transfer is an internal network site-to-site transfer, the resultant TCP window has a scaling factor of 2^9 by using values from Table 3.13 the expected latency using Equation 3.4 and solving for *LatencyInSeconds* is $^{1.9000\text{E}+07}/_{2.6843\text{E}+08} = 7.0782\text{E-}02$ seconds.

All of the major CPU vendors have incorporated shared memory zones by using a dedicated virtual address translation service (ATS) into their chipset I/O memory management units (IOMMU). The feature is called “VT-d” (**V**irtualization **T**echnology for **D**irected I/O) for Intel processors since 2008 and “AMD-V” (AMD Virtualization) for AMD processors since 2006. Simplistically, VT-d allows hosts to provide PCIe pass-through to VMs by acting as the “glue-code” between an I/O device on the PCIe bus and the VM’s memory address space.

[37]Common with VMWare since 2005

For intra-zone communication in AWS the C3, C4, R3, and I2 instances can be enabled with enhanced networking capabilities. Amazon EC2 supports enhanced networking capabilities using single root I/O virtualization (SR-IOV) resulting in higher packets per second, lower latency, and lower jitter[38]. SR-IOV allows a 10Gb NIC or InfiniBand host channel adaptor (HCA) to present itself as multiple separate I/O devices, which are virtual functions, and these functions interact with VT-d independently. This, in turn, allows all VMs to bypass the hypervisor entirely when performing DMA operations.

As the industry optimizes virtualization through software or hardware, the $\mathcal{L}_{\mathcal{N}}$, part 1 latency times trend toward native, non-virtualized latencies but only if applications are carefully crafted. In a carefully crafted MPI benchmark run by Lockwood et.al. [LTW14], they compared SR-IOV for a four node `c3.8xlarge` Amazon EC2 image. The benchmarks[39] were run for SR-IOV enabled and disabled, Figure 3.32, with jitter compared, Figure 3.33, and with latency comparison for SR-IOV enabled, disabled, and native modes, Figure 3.34.

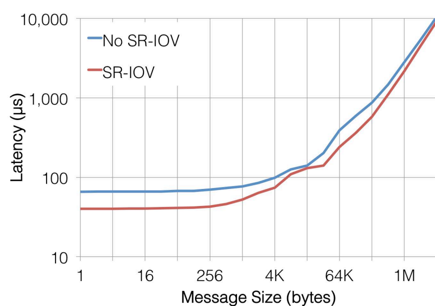


FIGURE 3.32: SR-IOV enabled and disabled

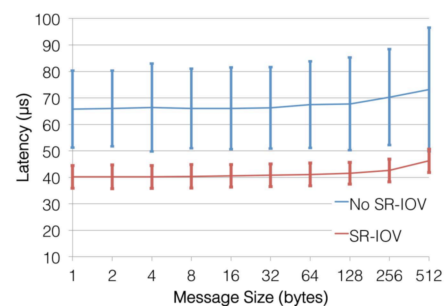


FIGURE 3.33: SR-IOV enabled and disabled with jitter error bars

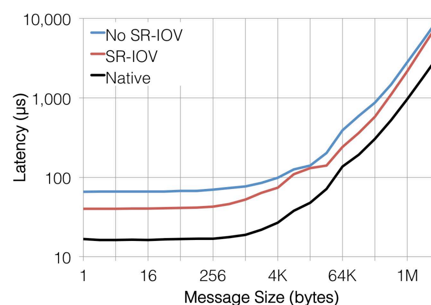


FIGURE 3.34: SR-IOV enabled, disabled, and Native modes

As can be seen from the use of SR-IOV, latencies were reduced and jitter was improved. In the end, SR-IOV is not better that native-mode execution latency which was expected. This is expected as virtualization will introduce latency due to context

[38] http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/enhanced-networking.html

[39] 1)OSU Micro-Benchmarks - unidirectional and bidirectional, 2)CONUS-12km benchmark, and 3)Distributed European Infrastructure for Supercomputing Applications AUSURF112 benchmark.

switching, buffering, decision logic, and similar constructs that take more time than direct circuitry. What the benchmark does show is that current cloud computing provider virtualization latency is improving. Combining these values with throughput measurements from the beginning of this section for Amazon, Google, and Rackspace in Tables 3.8, 3.9, and 3.10 respectively, gives a good parameterization for internal cloud provider values. The system has possible $\mathcal{L}_{\mathcal{N}}$, part 2, long-haul distances and millisecond latencies as shown in Table 3.15.

Cities		Distance	Milliseconds				
City A	City B	Km	Internal-US	Oceanic	Japan	Europe	Total (ms)
San Francisco, US	Helsinki, FI	8717.2617	36.4404	75.7885		12.2390	124.4680
San Francisco, US	Madrid, SP	9313.6083	36.4404	75.7885		12.2390	124.4680
Helsinki, FI	Tokyo, JP	7816.5867		145.6495	9.0079		154.6575
Madrid, SP	Tokyo, JP	10760.3158		145.6495	9.0079		154.6575
Tokyo, JP	San Francisco, US	8262.7892		109.991			109.9910

TABLE 3.15: Example System $\mathcal{L}_{\mathcal{N}}$ Values

With $\mathcal{L}_{\mathcal{N}}$ parts 1 and 2 quantified with empirical data, $\mathcal{L}_{\mathcal{S}}$ is now quantified. In this section flit injection and cycle latency has been described. As flit traffic is measured in flits per node per cycle (unit-of-time), the empirical data from Kim & Kim the $\mathcal{L}_{\mathcal{S}}$ values are used to quantify flit latency. The latency is from the 0.06 flit injections/cycle/node ±0.02 flits with a nominal latency of 30 cycles ±10% with a buffer depth of 8 flits; see Figure 3.24. As part of NoC processing, large network packets are broken into small pieces (flits) where the first flit, the header flit, contains information about this packet's destination address and routing control which sets up the routing behavior for all subsequent flits associated with the packet. The head flit is followed by zero or more body flits that contain the transmitted data. The last flit, the tail flit, performs book keeping to close the connection between the two nodes. In Kim & Kim "small" packets are 8 bytes and "large" packets are 72 bytes.

The zero-load latency (T_o) [DT03] of a packet can be summarized as follows: $T_o = T_h + T_s = H \cdot t_r + {}^{L}/_{b}$. Where T_h is the header latency, T_s is the serialization latency, H is the hop count, t_r is the per-hop router latency, L is the packet size, and b is the channel bandwidth. With data from Kim & Kim the $T_h + T_S$ ranges from 6–17ns. Or using the alternate equation 2–10 hops (H), 2 cycles for t_r and L ranges from 16 (8 bytes) to 576 bits (72 bytes). The bandwidth is defined as: $b = w \cdot f_c$, where w is the width of the channel in bits (64–1024) and f_c (1ns) is the inverse of the propagation delay of a bit

along the longest communication channel. Using these alternate values we get between 5–20ns. With this, ranges for $\mathcal{L}_{\mathcal{S}}$ are known for the NoC parameters and topologies from the empirical data sources at the bottom of page 120.

The $\mathcal{L}_{\mathcal{P}}$ values for the example system are derived from the source code for the neural networking source code from [WTS+13] using Algorithms 4, 5, and 6 for used in Equation 2.21. In Table 3.16 the initial percentages of parallel and serial code are determined from the source lines of code. In Table 3.17 the instruction types and counts that occur in the parallel and serial portions of the code are identified. The number of machine cycles needed for each instruction type are determined and the cycle count and percentage of all instructions are determined for a specific architecture (see end of table notes). Finally, a gross indication of parallel and serial allocation is done. In Table 3.18 the joint delay, J_D (page 58), is determined by summing the parallel and serial times by percentage of instructions used. Note N is the number of cores used in the parallel operation. Finally Table 3.19 sums the parallel and serial times for a single pass of execution. Additional information is given for the number of iterations (training epochs) and volume of data that is processed. The total execution time calculated at the end of the table is in proportion to the original neural network spot price calculation using the original Intel Core 2 Duo processor 2.4GHz computing equipment.

Count	Type	Percentage
112	Serial SLOC	40.73
163	Parallel SLOC	59.27
275	Total SLOC	100.00

Table 3.16: Serial and Parallel portions of Example Application

Count	Inst. Type	Cycles/Inst.	Count*Cycles	% of Inst.	Parallel/Serial
114	Multiplication	7	798	13.81	P
116	Addition	3	348	6.02	P
79	For (Branch)	2	158	2.73	P
38	If (Jump)	2	76	1.31	S
245	Store	6	1470	25.43	P
35	Division	70	2450	42.39	P
51	I/O	8	408	7.06	S
6	File I/O	12	72	1.25	S
684	Sum of types				

Intel Core i7 Extreme Edition 980X (Hex Core), 147,600 MIPS, 3.33GHz
Intel 64 and IA-32 Architectures Optimization Reference Manual, P/N 248966-030, Sept. 2014

Table 3.17: Instruction Types, Instruction Density, and Parallel/Serial Allocation

F_p	59.27			F_s	40.73		
N	3						
G_{p_i}	P_{d_i}	ns	$G_{p_i} \cdot ns$	G_{s_i}	S_{d_i}	ns	$G_{s_i} \cdot ns$
13.81	7	2.1021E-09	2.903E-08	1.31	2	6.00601E-10	7.86787E-10
6.02	3	9.00901E-10	5.42342E-09	7.06	8	2.4024E-09	1.6961E-08
2.73	2	6.00601E-10	1.63964E-09	1.25	12	3.6036E-09	4.5045E-09
25.43	6	1.8018E-09	4.58198E-08			Sum	2.22523E-08
42.39	70	2.1021E-08	8.91081E-07				
		Sum	9.72994E-07				

TABLE 3.18: Calculations for J_D for $\mathcal{L}_\mathcal{P}$

Parallel	Serial	Total
1.92231E-05	9.06334E-07	2.01295E-05
	20.12945224	seconds for 10^6 training epochs
	3842	spot prices
	20	spot prices per window
	192.1	spot prices per window size
	3866.867776	seconds to process spot prices per window size
		1 hour 4 minutes 27 seconds to process spot prices

TABLE 3.19: Calculation Check for Total Processing Time

Table 3.20 gives a summary of empirical values for the example system. For $\mathcal{L}_\mathcal{P}$ High values a maximal upper variation of 60% time increase is used based on Giles Reger's work [Reg10]. Reger performed a number of micro-benchmarks and selected benchmarks from the DaCapo benchmark suite. These multicore benchmarks provided an execution of controlled, dense, processes on a multicore architecture. The execution gave a worse case, run-time variance of 60% for the benchmarks. This controlled set of executions was preferred over the"rule-of-thumb" of 50% variance expected for non-dedicated, shared VM execution in a cloud system. This is described in the work by Jörg Schad, et.al. on run-time measurements for cloud computing [SDQR10].

				seconds		
	Low	High	Units	Low	High	Method
$\mathcal{L}_\mathcal{N}$ part 1 (Latency in Cloud Provider Systems)	35	50	ms	3.5000000E-05	5.0000000E-05	
$\mathcal{L}_\mathcal{N}$ part 2 (Latency Between Data Centers)	109.9910	124.4680	ms	1.0999100E-01	1.2446800E-01	
$\mathcal{L}_\mathcal{S}$ (Network on Chip)	6	17	ns	6.0000000E-09	1.7000000E-08	Kim & Kim Empirical
	5	20	ns	5.0000000E-09	2.0000000E-08	Zero-load Latency
$\mathcal{L}_\mathcal{P}$ (Parallel and Serial portions of S/W)	20.1219500	32.1951200	ns	2.0121950E-08	3.2195120E-08	Single Thread
Totals for Single thread				1.10026026121950E-01	1.24518049195120E-01	Kim & Kim Empirical
				— Or —		
				1.10026025121950E-01	1.24518052195120E-01	Zero-load Latency
Full application run time added in:						
$\mathcal{L}_\mathcal{P}$ (Parallel and Serial portions of S/W)	3866.867776	6186.988442	sec	3866.867776	6186.988442	Full application
Totals for Full application				3.8669778020060E+03	6.18711295961700E+03	Kim & Kim Empirical
				— Or —		
				3.86697780200500E+03	6.18711295962000E+03	Zero-load Latency

TABLE 3.20: Summary $\mathcal{L}_\mathcal{X}$ Empirical Values

3.4.7 DDE Calculated Optimal Message Rate

Using MathWorks MATLAB v8.5.0.197613 (R2015a), a series of DDEs were calculated using the `dde23()` function for Lotka-Volterra equations [Lot10, Vol26]; extended by Holling [Hol59a, Hol59b]. Using the values in Table 3.21, the graphs for equilibria and population over time shown in Figures 3.35 and 3.36 are calculated as initial, "ground" conditions.

Variable	Value
repPrey	0.25
PredCoef	0.01
PredMort	1.00
repPredPrey	0.0195
m	200
y1 (prey)	100
y2 (predator)	10

TABLE 3.21: Stablized Lotka-Volterra DDE

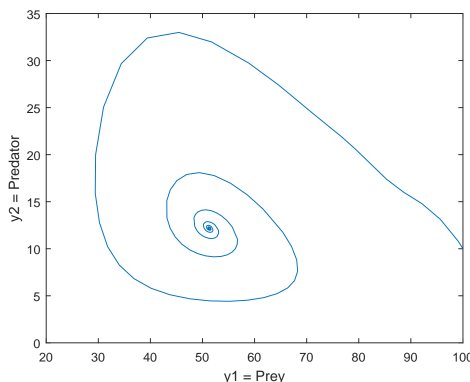


FIGURE 3.35: Initial Condition Stable Equilibriua

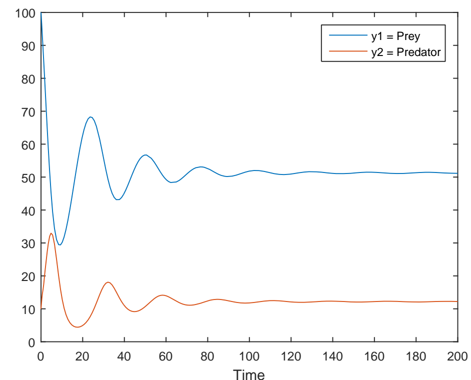


FIGURE 3.36: Initial Condition Time-phased Populations

Next, a change in the definition of variables is given in Table 3.22 making the Lotka-Volterra equations applicable for use in this work. The MATLAB code is given in Appendix I. The novel approach is making latency a multiplicative value of the Predator Coefficient (PredCoef). Using the computed total latencies for Zero-load Latency[40] from Table 3.20, a random coefficient in the range of the high and low values is multiplied with the predation coefficient allowing latency to effect consumption of resources. This value is not added to the `lags` array as that would only produce a $(t - \tau)$ term further in the past and would not account for latency accretion in the system. To illustrate how this coefficient effects the relationship, Figures 3.37 and 3.38 show a ten-times increase in the value from 0.1245sec to 1.245sec. The message capacity mean is μ = 51.6598 messages with this latency as compared to the mean of μ = 561.9005 as shown in Figure 3.44.

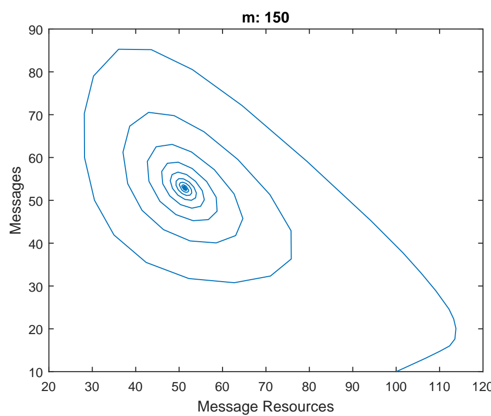


FIGURE 3.37: Equilibriua for m = 150, High Zero-load latency (1.245sec)

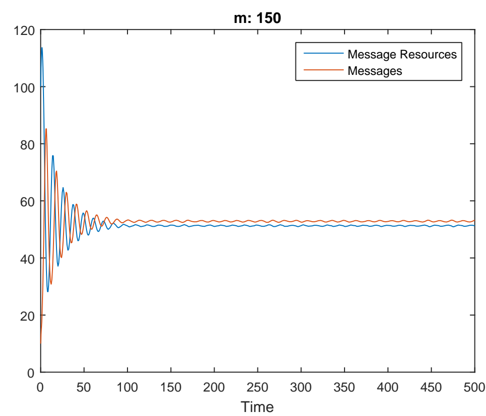


FIGURE 3.38: Time-phased Populations for m = 150, High Zero-load latency (1.245sec)

In Table 3.22 this work converts “prey” to system resources and “predator” to usage of system resources. As can be seen in the table there is 100% resource recovery for system and messaging resources. It is the message resource use coefficient that has the latency delay applied to it as it increases the predation of resources making them unavailable (i.e. a time lag in the system). The remaining two values are message processing creation rate and carrying capacity. Message processing creation rate has a slight growth value as this model does not need to have multiples of the same process as it would not increase the ability of the continuous neural network, Equation 3.3, to predict any better. We have done the necessary parallel activity to speed-up the computation as shown by calculation of $\mathcal{L}_{\mathcal{P}}$. This leaves the carrying capacity of message resources as the free variable.

Carrying capacity, m, is varied in Figures 3.39 to 3.52 with Figures 3.43 and 3.44 representing a stable ratio with a high performing use of resources. From these tables the

[40]Zero-load is used as Kim & Kim empirical values, while good, are statistically narrow.

Variable	Value	Description
repPrey	1.00	System resource recovery rate for processed messages.
PredCoef	0.01	Message resource use coefficient.
PredMort	1.00	Message resource recovery rate after receipt.
repPredPrey	0.0195	Message processing creation rate per application running on a single processor.
m	150	Carrying Capacity
y1	100	Initial condition; number of message resources
y2	10	Initial condition; messages in system

TABLE 3.22: MATLAB variable settings for Lotka-Volterra DDE

increase in message resource carrying capacity has a sustaining time-phased population until the pressure on message resources starts to induce rapid, large amplitude oscillation in messages and message resources. Eventually the system "crashes" as shown in Figures 3.47 and 3.49. Normal Lotka-Volterra equations do not have a 100% "prey" population increase with a 100% "predator" mortality rate. This is a novel use of the Lotka-Volterra model in this work identifying over-subscription (i.e. carrying capacity) of resources leads to "thrashing" or live-lock as shown in the time-phased graphs as m increases.

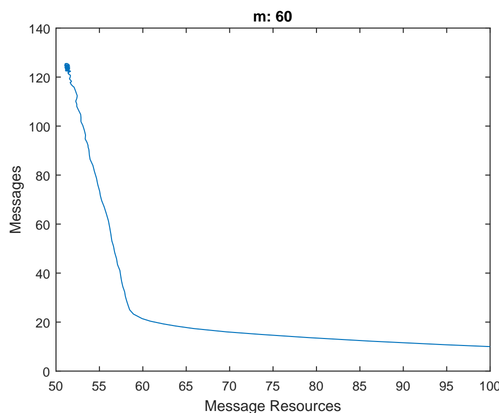


FIGURE 3.39: Equilibriua for m = 60

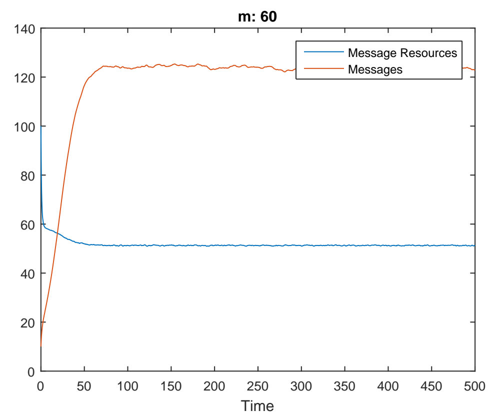


FIGURE 3.40: Time-phased Populations for m = 60

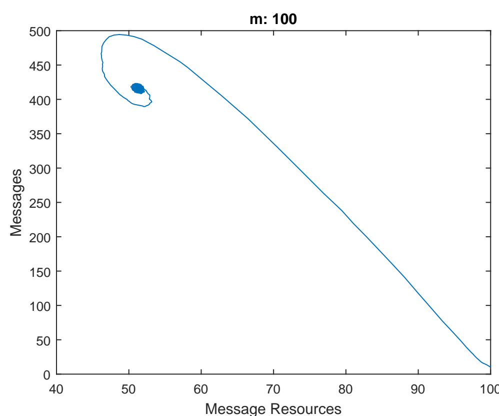


FIGURE 3.41: Equilibriua for m = 100

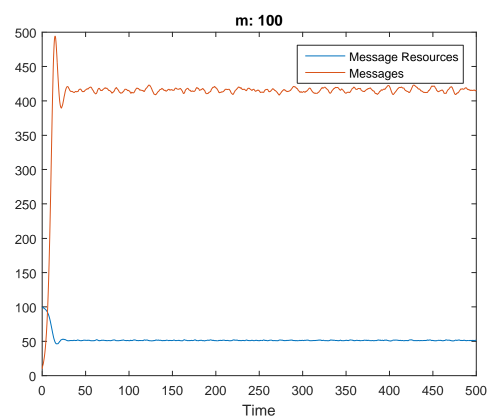


FIGURE 3.42: Time-phased Populations for m = 100

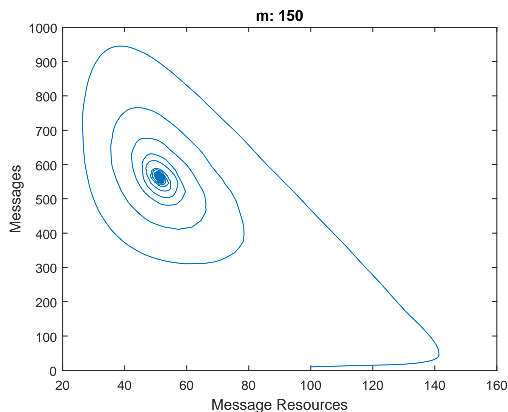


FIGURE 3.43: Equilibriua for m = 150

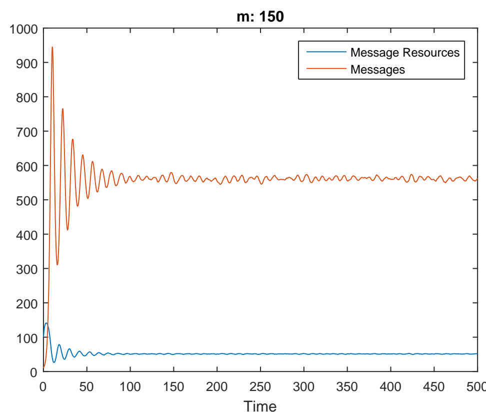


FIGURE 3.44: Time-phased Populations for m = 150

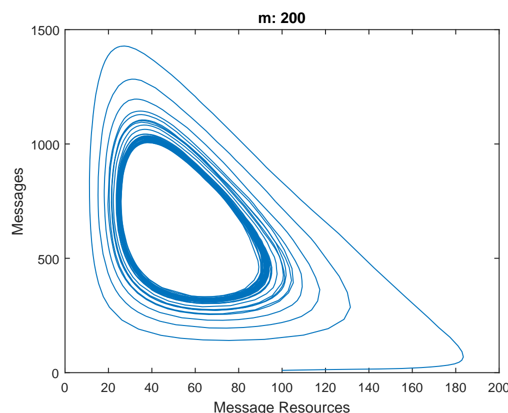


FIGURE 3.45: Equilibriua for m = 200

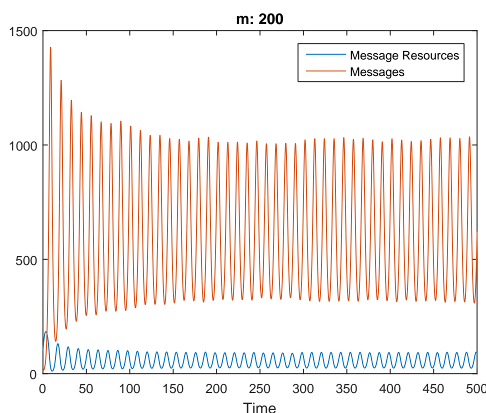


FIGURE 3.46: Time-phased Populations for m = 200

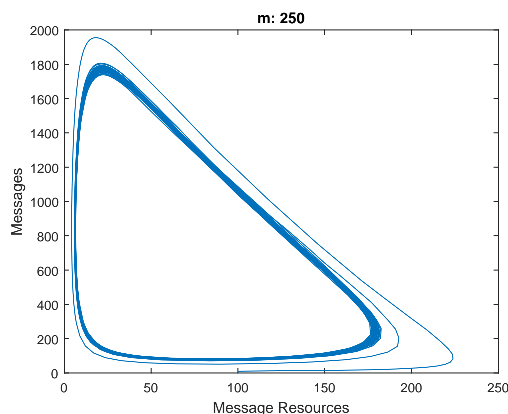


FIGURE 3.47: Equilibriua for m = 250

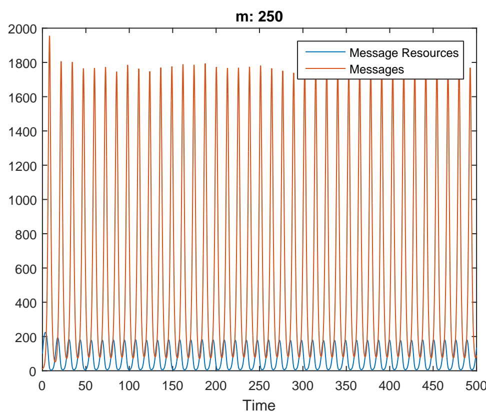


FIGURE 3.48: Time-phased Populations for m = 250

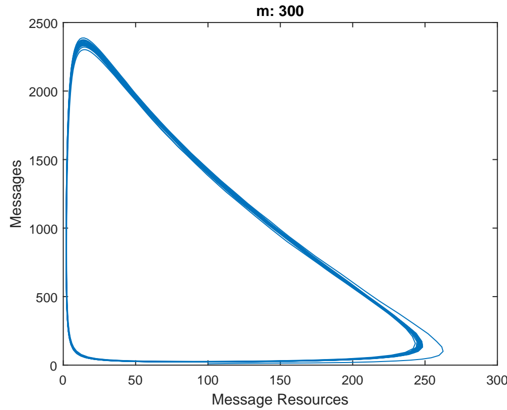


FIGURE 3.49: Equilibriua for m = 300

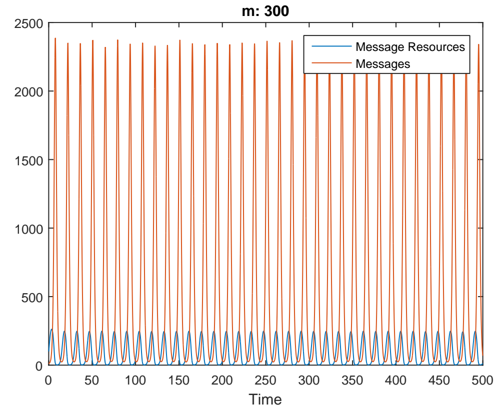


FIGURE 3.50: Time-phased Populations for m = 300

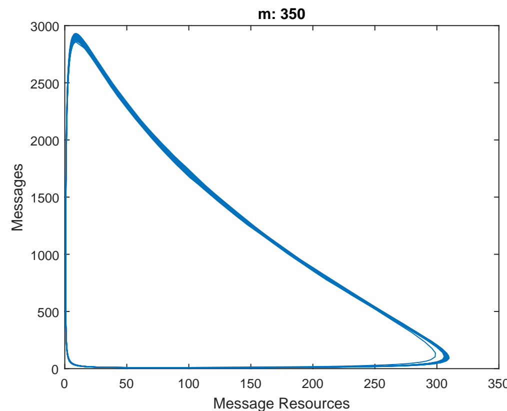


FIGURE 3.51: Equilibriua for m = 350

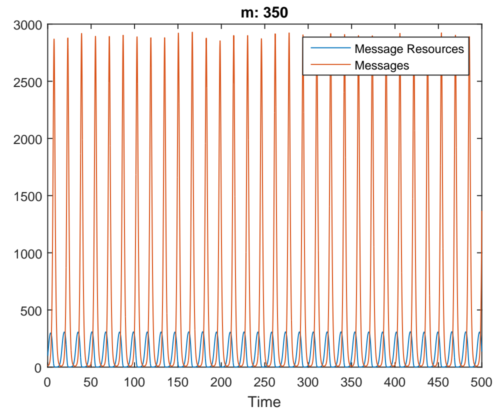


FIGURE 3.52: Time-phased Populations for m = 350

3.4.8 RITA Notation for Example CTNN Application

This section has shown the quantification of latency, message capacity, and messaging requirements needed. At the end of this section the RITA notation for Continuous-Time Neural Network, as shown in Listing `ContinuousNN.rit`, is created with the event controls necessary for this example application. Of note, each data center has the same prediction code, and thus the same RITA control structure. The goal is to update the other nodes with the broadcasting node's data. While the price record data rates are very low (about 17 per hour), there are four data centers with bi-directional communication so there are ten communication paths; but this is not quite the true number as Finland and Spain must pass through the USA or Japan routing to communicate. While a poor design, it is more realistic as world-wide networks have some poorly designed segments. The graph of nodes in Figure 3.31 was purposely drawn to not give the $\frac{n(n-1)}{2}$ form of "all nodes connected to all nodes". In Chapter 4 this will be taken into consideration for RITA message flow control simulations.

ContinuousNN.rit

```
1  with "../include/contNN.hpp";
2  use  "CTRNN",     // Continuous-time recurrent neural network calculation.
3       "trainNN",   // Train neural network with window of data.
4       "sendVal",   // Send values to peer data centers.
5       "getLower",  // Get lower bound delta value for significance comparison.
6       "getUpper",  // Get upper bound delta value for significance comparison.
7       "getTime",   // Get the current time for the application.
8       "getDelta",  // Get the comparison delta for evaluate_time().
9       "newValue",  // Reports that a new spot price has been generated.
10      "getWindow"; // Get the size of the moving window on data.
11
12 /////////////////////////////// -=[ Control ]=- //////////////////////////////
13 // ++
14 // The control structure is repeated at each data center as the purpose is to
15 // have the training and prediction
16 // --
17 control {
18   event( setat, bool   ) : systemOn   ( false );
19   event( spike, float8 ) : valSend    ( 0.0 );
20   event( trans, string ) : epoch      ( false );
21   event( spike, bool   ) : newRecord  ( false );
22   event( spike, bool   ) : doTraining ( false );
23
24   float4 : lowerVal( getLower() );
25   float4 : upperVal( getUpper() );
26   int4   : windowLimit( getWindow() );
27   bool   : newValue( false );
28   time   : epochStart(""); // A null string is a "NaN" for time calculations and
29                            // a value must be assigend prior to any comparison
30                            // or use of the variable in a calculation.
31   begin
32     TRAINING   <- systemOn( true ), PREDICTION;
33     PREDICTION <- TRAINING, SYSTEM_IO;
34     SYSTEM_IO  <- PREDICTION;
35   end
36 }
37
38 ////////////////////////////// -=[ TRAINING ]=- /////////////////////////////
39 // ++
40 // Training controls the CTNN training for price records as they come into
41 // the neural network by starting a new epoch.  Data comes from this data center
42 // and peer data centers.
43 // --
44 system TRAINING {
45   int2 : windowCount( 0 );
46
47   guard systemOn {
48     if( systemOn == true ) {
49       return systemOn_TRUE;
50     }
51     break StartTraining;
52   }
53
54   condition NewTraining {
55     if( windowCount + 1 > windowLimit ) {
56       windowCount = 0;
57       epochStart = getTime();
58       trainNN();
59       return NewTraining_TRUE;
60     }
```

```
61      else {
62        if( newValue == true ) {
63          windowCount = windowCount + 1;
64          newValue = false;
65        }
66      }
67      return NewTraining_FALSE;
68    }
69
70    vector StartTraining {
71      guard           : systemOn;
72      condition( and ) : NewTraining;
73      result          : epoch("new"), newRecord ( true );
74    }
75
76    guard doTraining {
77      return doTraining_TRUE;
78    }
79
80    vector ProcessData {
81      guard           : doTraining
82      condition( and ) : NewTraining;
83      result          : newRecord( true );
84    }
85
86  } // TRAINING
87
88  ///////////////////////////// -=[ SYSTEM_IO ]=- /////////////////////////////
89  // ++
90  // SYSTEM_IO is an output only system.  As peer systems send their output,
91  // the application code receives it via network communication, and using
92  // those values for the spot price table used to train the next prediction
93  // cycle to submit bids for cloud system execution of application code.
94  //
95  // Output only occurs when a significant plus or minus bound from the current value
96  // is calculated from the CTNN() function.
97  // --
98  system SYSTEM_IO
99  {
100   int4   : status( 0 );
101   float4 : delta ( 0.0 );
102   float4 : spotPrice( 0.0 );
103
104   guard valSend {
105     if( ( delta < lowerVal ) or  // plus or minus from control band:
106         ( delta > upperVal ) {   // action <[ no action ]> action
107       return valSend_TRUE;
108     }
109     break SendData
110   }
111
112   condition UpdateNet {
113     sendVal( delta );      // Non-blocking call. SendVal() sends data to other
114     return UpdateNet_TRUE; // data centers with new data.
115   }
116
117   vector SendData {
118     guard           : valSend;
119     condition( and ) : UpdateNet;
120     result          : epoch("reset");
121
122   guard newRecord {
```

```
123      newRecord = false;
124      status = signalData(); // A blocking I/O
125      switch status {
126        case 0 :
127          return newRecord_FALSE;
128        case 1 :
129          return newRecord_TRUE;
130      }
131      break UpdateSystem;
132    }
133
134    vector UpdateSystem {
135      guard  : newRecord;
136      result : doTraining;
137    }
138
139  } // SYSTEM_IO
140
141  //////////////////////////// -=[ PREDICTION ]=- /////////////////////////////
142  // ++
143  // Has sufficient training been done for the prediction algorithm? As per RITA
144  // dependencies, the create_time() is done in the application code with
145  // epochStart being set by getTime().  If it is a new epoch predict using
146  // getTime() value being returned.
147  // --
148  system PREDICTION
149  {
150    float4 : delta( 0.0 );
151    time   : predictDelta( getDelta() ); // Application getDelta() call.
152
153    guard epoch {
154      // ++
155      // Is the data available? That is, have delta seconds since the last
156      // prediction run occured?
157      // --
158      if( evaluate_time( epochStart, AFTER, predictDelta ) {
159          return epoch_TRUE;
160        }
161      }
162      break EpochCheck;
163    }
164
165    condition advanceNN {
166      switch epoch {
167        case "new" :
168          delta = CTRNN(); // Beer's function in the application.
169          return advanceNN_TRUE;
170        case "reset" :
171          epochStart = getTime();
172          return advanceNN_FALSE;
173      }
174      break EpochCheck;
175    }
176
177    vector EpochCheck {
178      guard           : epoch;
179      condition( and ) : advanceNN;
180      result          : valSend( delta );
181    }
182  } // PREDICTION
```

ContinuousNN.rit

Chapter 4

Experiment, Simulation, and Analysis

EXPERIMENT creation and hosting on cloud providers such as Amazon's EC2 and Google's App Engine using OpenStack and OpenShift would be ideal for study of application performance. The development period for such a study can take between 6 to 18 man-months depending on tools available, hosting requirements, and difficulty in acquiring products. Additionally, proficiency with these products (which are not taught, but are learnt via practicum) can add 24 man-months. Thus the only efficient method to demonstrate the novelty in this work for the condition-event matrix, mapping algorithms, and RITA description language for allocation of applications across federated computational systems is to not do unconstrained execution "in the wild" but provide high fidelity simulations to constrain the experiment through well controlled parameters analyzing the results for controlled, repeatable, and finite execution results demonstrating improved optimization in cloud computing systems.

4.1 Choice of Simulation Experimental Environment

AN evaluation of simulation tools was done in surveys by Alman [APR05] and Codl [CKK03]. An interesting simulator was created from a Google trace using data from 2011 and was the basis of the work by Di and Cappello for "GloudSim" [DC14] which used this singular analysis of one month of production at Google for analysis. The density of the Google data, and parsing it using GloudSim, generated several papers in the literature. While GloudSim was interesting, it is fixed in time and space which

calls into question its veracity as time progresses. None of the simulators discussed by Alman, Codl, or Di and Cappello met the need for this work.

Initially OMNeT++/INET based iCanCloud [NnVPC+12] was considered as the simulator for this work but iCanCloud is currently not supported and is non-operable, does not match its documentation, and fails to execute its own self tests. CloudNetSim was considered next. In a comparison paper by Cucinotta [CS13] CloudNetSim developed by Malik, et.al.[MBA+14] seemed to have features needed for evaluating heterogeneous processor simulations. Mostly the simulator was designed just for power comparisons. This is a good metric as cloud providers are only profitable if they can control this resource consumption cost but had no utility for this work. CloudAnalyst [WCB10], built on CloudSim, was also evaluated. CloudAnalyst is based on CloudSim[1] which is based on simjava[2]. Both CloudAnalyst and CloudSim are focused on modeling network queries to a cloud system and modeling the internal communication between CEs is not simple or straight-forward and were focused on only long-haul network evaluation for web servers.

In the end, the result was reverting to a native OMNeT++ simulation. OMNeT++ gives a cycle-accurate simulation fidelity needed for modeling CE communication and throughput. A framework of native OMNET++ simulation constructs from Technion, Israel Institute of Technology in Haifa, Israel called the Heterogeneous NoC Simulator (HNoCS)[3] built on OMNeT++ is used to model core and router elements. This framework allows varied NoC and PE bus topologies such as GPU, GPP, MIC, and HSA architectures to use RITA in different ways based on the condition-event matrix structure to take advantage of the various PE architectures. Evaluation of the condition-event matrix needs a cycle-accurate simulator and building a larger simulation with such a low-level set of primitives can be long and involved. To alleviate this, a discrete-event simulation environment is created with the ability to accelerate events using a simulation clock that can "leap" forward in simulation time to subsume cycle-accurate events thus giving the simulation the ability to focus on the event stimuli needed to show RITA performance improvements for heterogeneous systems.

Again, reviewing more simulators for cloud computing[4] [5] there has been little heterogeneous variance in the PEs, either hard-coded or parametrically set. The simulators consider PEs as homogeneous and are focused on cluster to cluster networking and network transport issues or energy required by the cloud system to operate. As the need to account for heterogeneity is repeatedly stated in this work, this leads to a need to

[1] http://www.cloudbus.org/cloudsim/
[2] http://www.icsa.inf.ed.ac.uk/research/groups/hase/simjava/
[3] http://webee.technion.ac.il/matrics/software.html
[4] https://networkonchip.wordpress.com/2011/02/22/simulators/
[5] http://research.cs.wisc.edu/arch/www/tools.html

simulate various architectures listed in Tables 2.1 and 2.2. Without a cycle-accurate hardware simulator, like SystemC[6], it is not possible to represent the register-transfer logic (RTL) at a sufficient level showing the accumulating latency described in Equations 2.15, 2.16, and 2.21 that result in Equation 2.22. While RTL simulation is good, it is too low a level for efficient simulation as shown in calculations in Table 3.20 where these small time values were fairly insignificant to allocation prediction. The next level of implementation in hardware where simulation shows the utility of RITA and algorithm allocation to PEs is at the NoC layer. This is where the heterogeneous compilers discussed in §3.4.4.4 can effectively modify communication through architecture specific machine instruction and micro-code. Above the NoC layer, we have the machine network stack and the discussion becomes one of RITA controlling communication at the IP or IB level (discussed in §2.4.2.3.2 and referenced in §3.4.6).

In §1.4 of Dally & Towles book [DT03], they give a history of interconnection networks over the decades. Networks have developed along three general paths: telephone switching networks, inter-processor communication, and processor-memory interconnects. Early telephone networks were built from electro-mechanical crossbars or electro-mechanical step-by-step switches. As late as the 1980s, most local telephone switches were still built from electro-mechanical relays. This is contrasted with long-distance switches being digital by that time due to newer equipment that separated the local exchange carrier and the long-distance providers. The key developments in telephone switching include the non-blocking, multistage Clos network from 1953 [Clo53] and the Bend network from 1962 [Ben62]. Many large telephone switches today are still built from Clos or Clos-like networks. Over the decades these switching networks became 2-D and 3-D mesh torus NoCs that led to the development of hyper-cube networks and then to butterfly NoC networks. Since the early 1990's there has been little difference in the design of processor-memory and inter-processor interconnection networks. Currently a variant of the Clos and Benes networks has emerged in multiprocessor networks as the provably efficient fat-tree NoC topology [Lei85]. In the NoCs studied in this work, each relies on either a global common non-blocking memory or a ring-bus topology (a one-dimensional torus).

Using the ContinuousNN RITA specification and starting at the NoC level for study of heterogeneous simulation in this work, three implementations of in-use, cloud computing PE architectures are considered. The Amazon AWS M3 configuration instance processor[7] uses the Intel Xeon E5-2670 v2 (Ivy Bridge) processor. The E5-2670 uses the Intel QuickPath Interconnect (QPI) between chip sockets that has a basic unit of transfer of an 80-bit flit transferred in two clock cycles (four 20 bit transfers, two per

[6] http://accellera.org/downloads/standards/systemc

[7] http://aws.amazon.com/ec2/instance-types/

clock cycle). Internally to the E5-2670 the cores and memory are connected with bidirectional ring bus with buffered switches between the bidirectional ring buses as shown in Figures 4.1 and 4.2. Sockets have a full point-to-point mesh NoC[8]. For the E5-2670, 10 cores are used. The chip family can support between 4 to 12 cores.

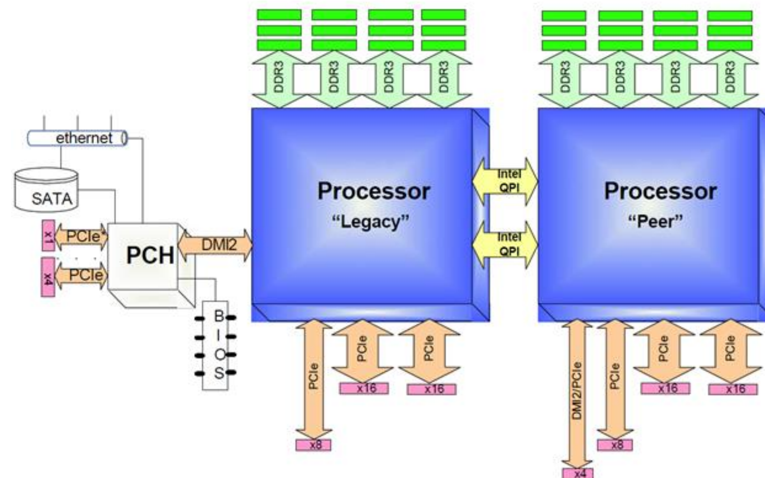


FIGURE 4.1: Chip Socket Interconnect Xeon E5-2670 v2 (Ivy Bridge)

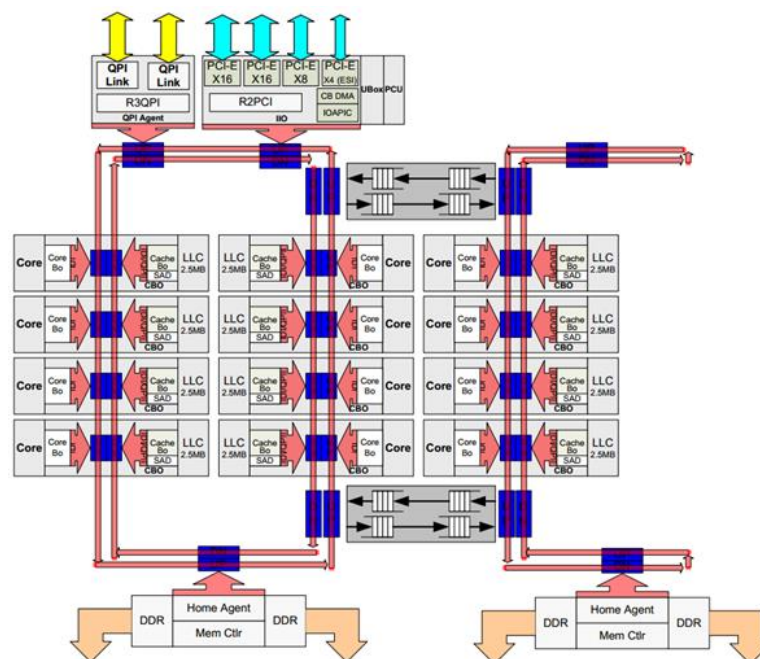


FIGURE 4.2: Core Interconnect Xeon E5-2670 v2 (Ivy Bridge)

The Amazon AWS G2 GPU configurations use the nVidia Kepler GK series processor with the same Intel Xeon E5-2670 v2 processor used to control the Kepler processor with communication between the two cards using PCI-e 3.0. The inclusion of a Kepler card adds a PE to the mesh network for chips as the GPU is truly a co-processor with the G2 configuration integrated into the cloud as an aggregate of a Xeon/Kepler card

[8] http://www.intel.com/content/dam/doc/white-paper/quick-path-interconnect-introduction-paper.pdf

set.
The Kepler GK processor internally closely resembles a vector processor (single instruction stream across multiple cores) and, for latency, is treated as a bus access to memory by the controlling CPU/GPP. This structure can be seen in Figure 4.3 which shows the die architecture of the Kepler GK110.

FIGURE 4.3: nVidia Kepler GK110 Architecture

In Figure 4.4 the Streaming Multiprocessor Architecture (SMX) provides the Kepler GK110 with an element to schedule 32 parallel threads (i.e. "warps"[9]). Each SMX features four warp schedulers and eight instruction dispatch units, allowing four warps to be issued and executed concurrently. The Kepler quad warp scheduler selects four warps, with two independent instructions per warp, dispatched at each cycle so the Kepler GK110 allows double precision instructions to be paired with other instructions[10]. As general cloud computing providers (Amazon, Google, and Microsoft) have yet to fully embrace FPGA, DSP, MIC (a.k.a. Intel Xeon Phi), and HSA processing power these architectures appear to be still too specialized for general use according to the trade press. While GPUs would also be in this list, there is a high enough commercial demand for graphics and GPU computation for Amazon to offer the AWS G2 configuration[11].

[9]In weaving cloth, the warp is the set of lengthwise threads that are held in tension on a frame or loom, hence the grouping name: threads are in a group called a warp.
[10]http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf
[11]http://www.hpcwire.com/2014/02/20/amazon-lead-highlights-hpc-cloud-progress/

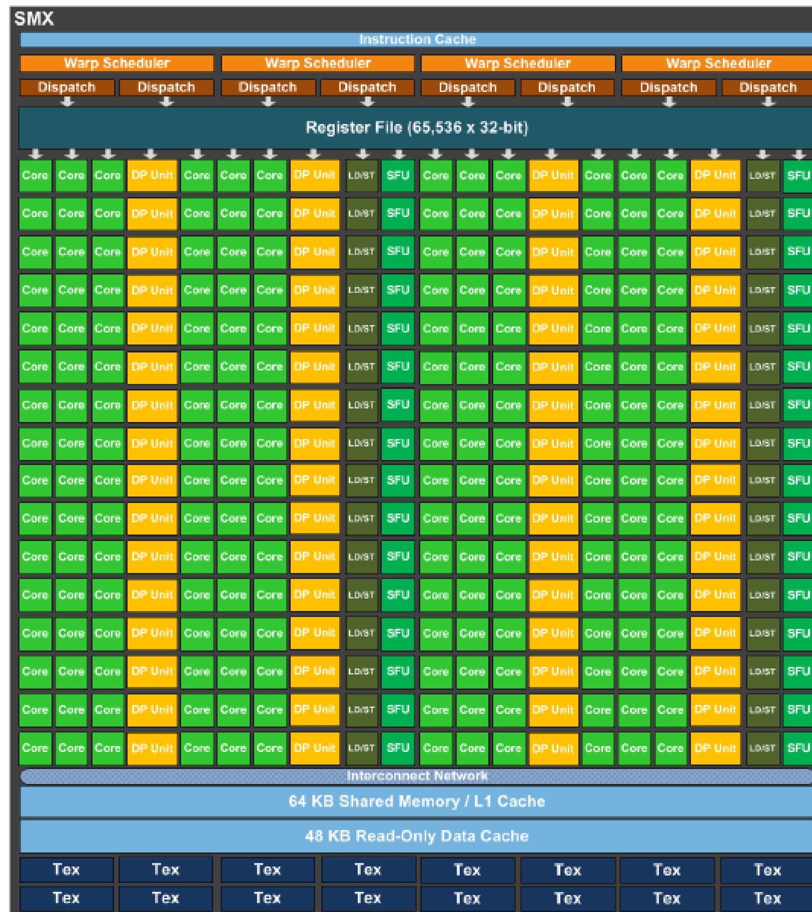


FIGURE 4.4: nVidia GK110 Streaming Multiprocessor (SMX) Cell

There is an exception. A French telecommunications company, Iliad S.A., has a subsidiary, Online.net, offering cloud computing using the ARM7 4-core Marvell ARMADA-XP® MV78460 SoC which uses a crossbar any-to-any processor bus interconnect topology as shown in Figure 4.5. This four-core processor is delivered as a single-board computer as "C1" server for Online.net. The MV78230 used is a server variation of the typical ARM7 embedded processor. Iliad explicitly choose this architecture for its low-power requirements as an IaaS. This offering uses no virtualization and users are assigned true dedicated hardware in contrast to the Amazon EC2, Google Compute Engine, and Microsoft Azure cloud offerings that do not guarantee dedicated systems. With the Iliad configuration an interesting side-effect is the claim of no "noisy-neighbors"[12] and this will be a contrasting environment for the other simulated PE environments.

[12]This effect is due to the shared infrastructure and activity of a virtual machine on a neighboring core on the same physical host leading to performance degradation of the other VMs on the same physical host. With neighboring VMs activated or deactivated at arbitrary times the process interrupts that occur for swapping on the physical host result in a negative impact in the actual performance of a resource.

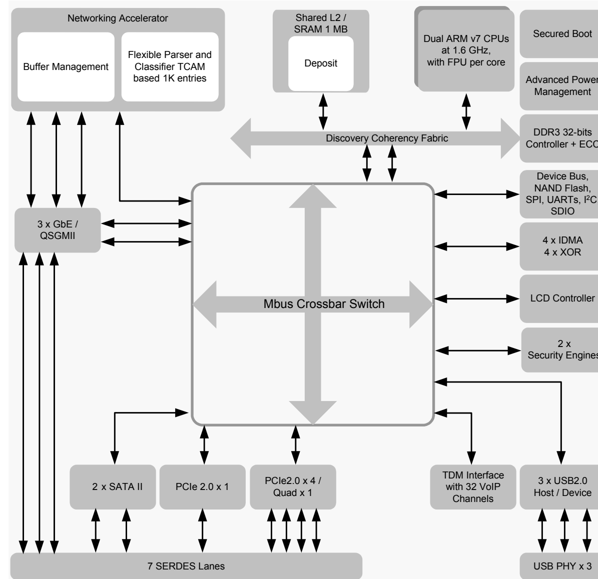


FIGURE 4.5: ARM7 Marvell ARMADA-XP® (MV78460) Architecture

4.2 Simulation Description

Working with OMNet++ network description language (NED)each of the PE architectures in §4.1 were described. See Appendix J for detailed code. Graphically each of the PEs are shown in Figures 4.6 to 4.9. The connections between all chips in the simulation uses a PCI-e 3.0 1 lane, 8Gbps theoretical, 7.88 measured bus. Internally the connections are shared L2 cache or, in the case of the Xeon chip, core-to-core connection is done using the QPI link. These connections, and their latency is described in the OMNeT++ NED code in Appendix J.

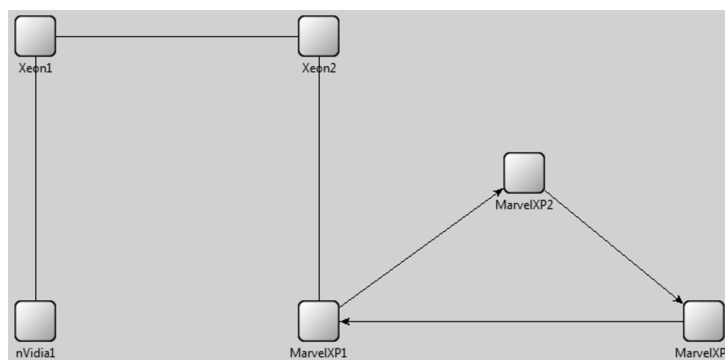


FIGURE 4.6: Top-level OMNeT++ Cluster Architecture

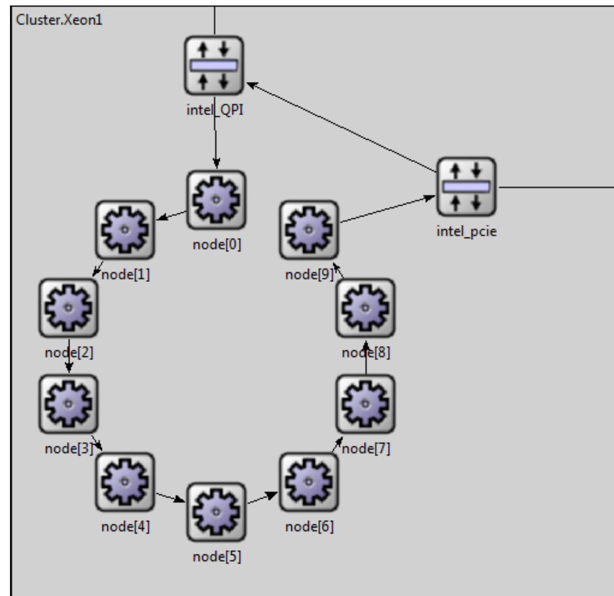


Figure 4.7: OMNeT++ Xeon Architecture Representation. Both Xeon1 and Xeon2 are identical and only Xeon1 is shown. The term "node" is used to indicate a core top-level view in a NoC

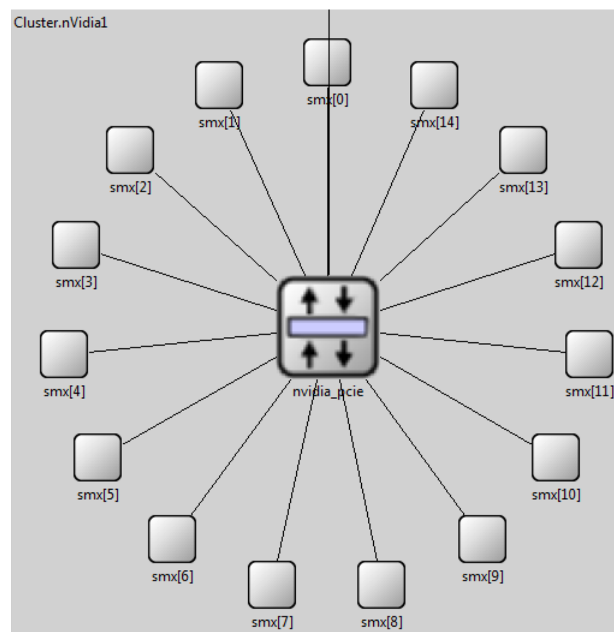


Figure 4.8: OMNeT++ nVidia Architecture Representation. The intermediary memory representation is subsumed in the SMX module

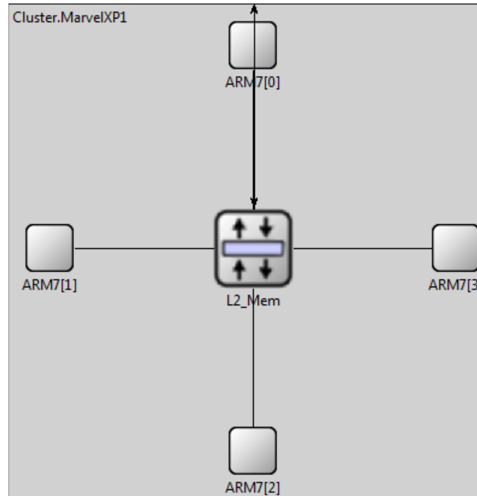


FIGURE 4.9: OMNeT++ MaxwellXP Architecture Representation. The intermediary memory representation is subsumed in the ARM7 module

In Figure 4.10 the messages used to control the operation of the condition-event matrix are shown. OMNeT++ can do a delayed send of a message (thus creating an event at a future time) but cannot quiesce and advance the simulation clock unless the finite state machine API calls are used. These API calls are discussed in §4.4.1 and §4.4.2 of the OMNeT++ manual. Since the simulation is at the NoC level, the network mesh simulation capability of OMNeT++ is used. This is similar to using structural description in VHDL which is at the RTL level rather than using a behavioral description which resembles procedural software programming. This network mesh feature is important to show recovered processing time from reduced messaging I/O operations. Simulating this uses three statistical functions. All functions use the OMNeT++ default Mersenne Twister random number generator by M. Matsumoto and T. Nishimura [MN98]. A *Chi* Square distribution is used for the *newRecord* message with 3 degrees of freedom[13] (k), μ = 2.89607239, σ = 2.75753474, an UCL of 5.65360713, and a LCL of 0.13853765 seconds. For the *newEpoch* message a non-negative, normal distribution was used with μ = 0.00051406991500 and σ = 0.00027594805711 seconds with the **AFTER** operator simulated with a LCL of 0.00030 seconds. The final, and externally visible, message in the simulation is *valueSend* which uses a dimensionless value to emulate valid values for delivery to the IP data network. This is shown in the `ContinuousNN.rit` listing in §3.4.8. The *valueSend* message control is simulated with a uniform distribution of $(0, 1]$ where the *lowerVal* variable is set to 0.4 and the *upperVal* variable is set to 0.6. These gating values were determined externally to the simulation and are in valid ranges and are comparable to values from Table 3.19. In Figures 4.11 to 4.13 message cascades for the Intel, ARM, and nVidia chips show message traversal. As these graphs are large and dense, counters were used to capture the unconstrained message processing and condition-event constrained message processing. The results are shown in §4.3.

[13] As k increases, the *Chi* Square distribution resembles a normal distribution. Thus k was kept at < 10 with $k = 3$ considered optimal.

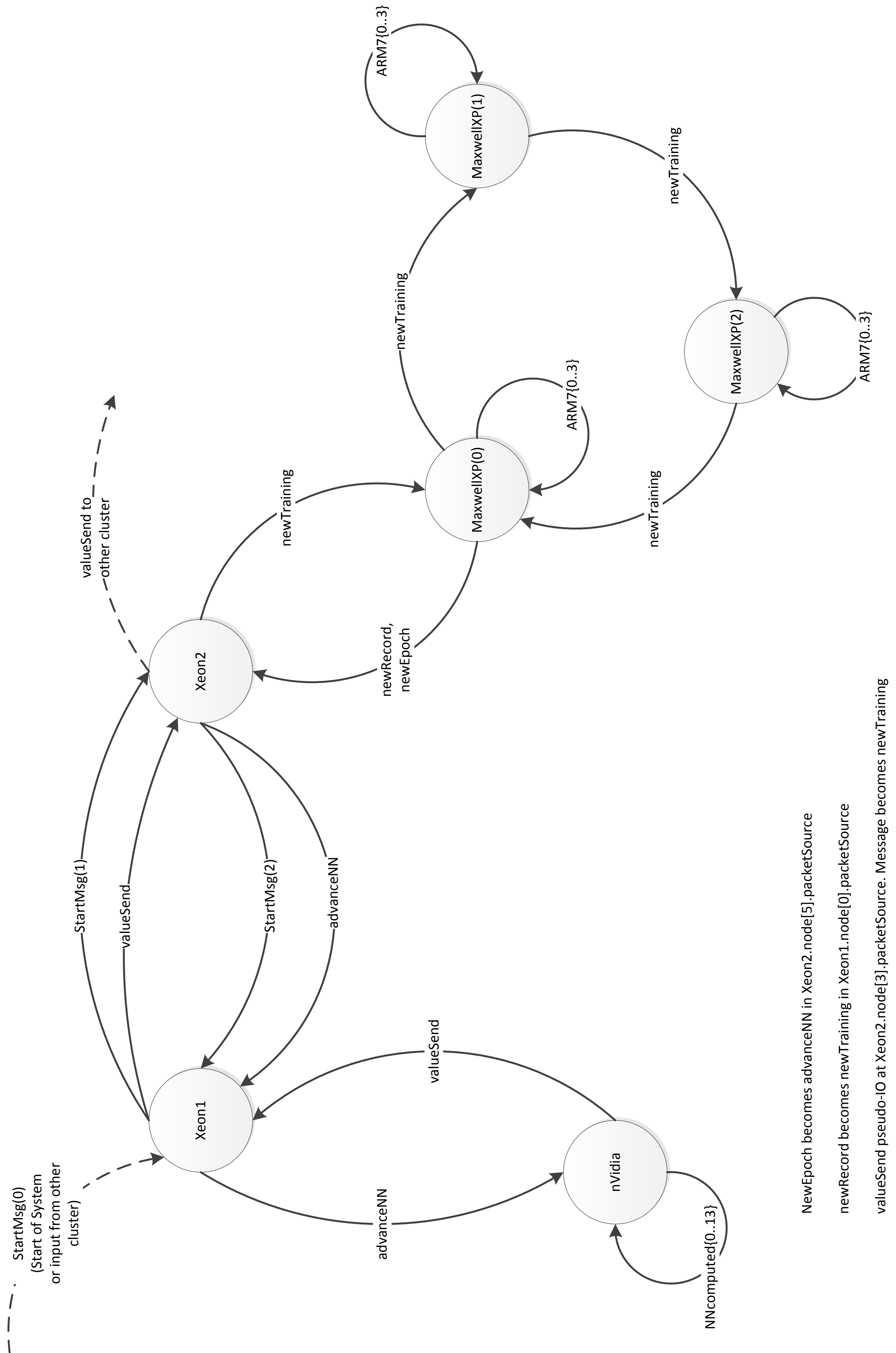


Figure 4.10: Message state transition for ContinuousNN

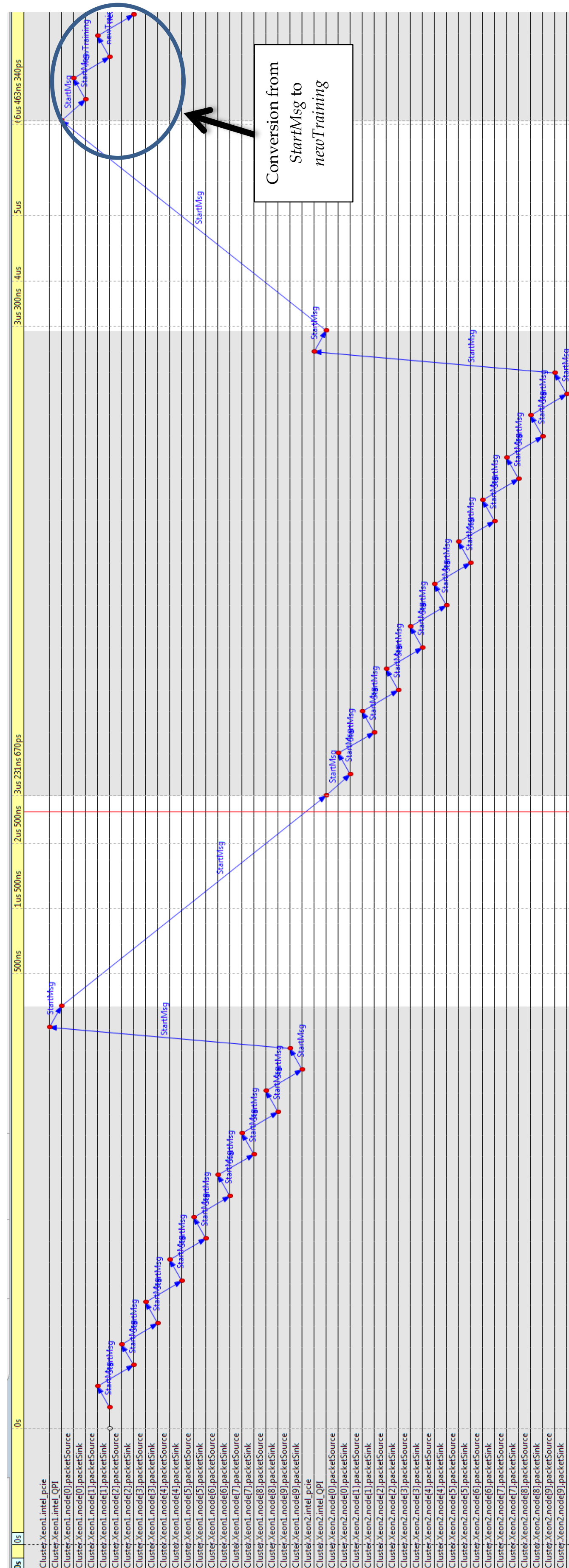


FIGURE 4.11: *StartMsg* Message Cascade

Figure 4.12: *newTraining* Message Cascade

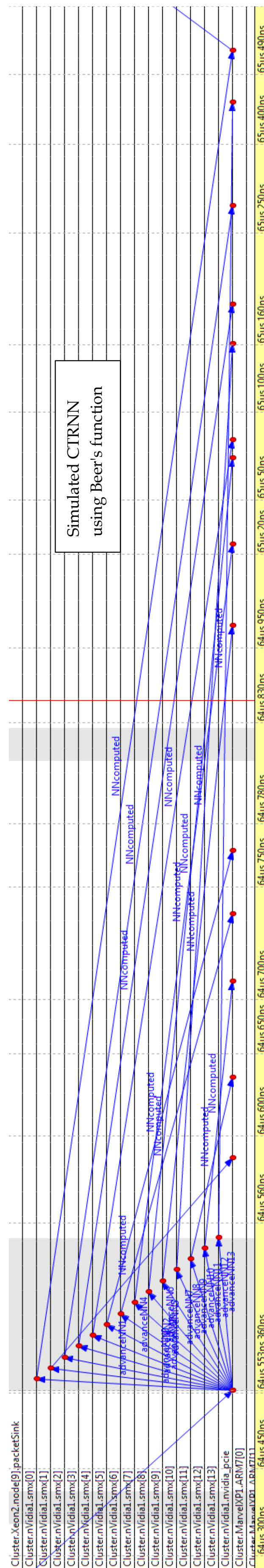


FIGURE 4.13: *computeNN* Message Cascade

4.3 Simulation Results

GIven that OMNeT++ is designed to produce the same stream of random numbers given a default seed, the simulation results shown in Tables 4.1 and 4.2 will remain unchanged regardless of the number of separate executions performed. This is by design for the OMNeT++ product thus giving identical output for reproducible executions. The time period of 5μsec (actual model time 4.9998676μsec) simulated real-time was specified as the “run until” time as the system had reached a steady-state execution and longer execution times were unnecessary. The use of the RITA condition-event method gave message and time reductions as shown in Tables 4.3 and 4.4. These percentage reductions are in line with results obtained by Wallace, McDonald, and Hague in work done in [WMH84]. An improvement in the amount of processing time recouped was expected but not at the same level as the avionics equipment used in the prior work. The nature of the heterogeneous PE architectures were expected to have an negative impact on the message transport between the PEs. After consideration of the two software environments and reviewing the simulation traces both environments use the condition-event method of message control and this would be the unifying basis giving similar message and time reductions.

TABLE 4.1: Unconstrained Message processing

Running in Express mode from event #1, t=0 ...	
Leaving Express mode at event #734206, t=0.00049998676	
Calling finish() methods of modules	
Cluster.nVidia1.nvidia_pcie: Number of valueSend messages sent:	3122
Cluster.MarvelXP1.L2_Mem: Number of newRecord messages sent:	4263
Cluster.MarvelXP1.L2_Mem: Number of newEpoch messages sent:	4263

TABLE 4.2: Condition-event Constrained Message processing

Running in Express mode from event #1, t=0 ...	
Leaving Express mode at event #734206, t=0.00049998676	
Calling finish() methods of modules	
Cluster.nVidia1.nvidia_pcie: Number of valueSend messages sent:	52
Cluster.MarvelXP1.L2_Mem: Number of newRecord messages sent:	250
Cluster.MarvelXP1.L2_Mem: Number of newEpoch messages sent:	241

TABLE 4.3: Percent Reduction in Messages

valueSend messages reduced by	98.334%
newRecord messages reduced by	94.136%
newEpoch messages reduced by	94.347%

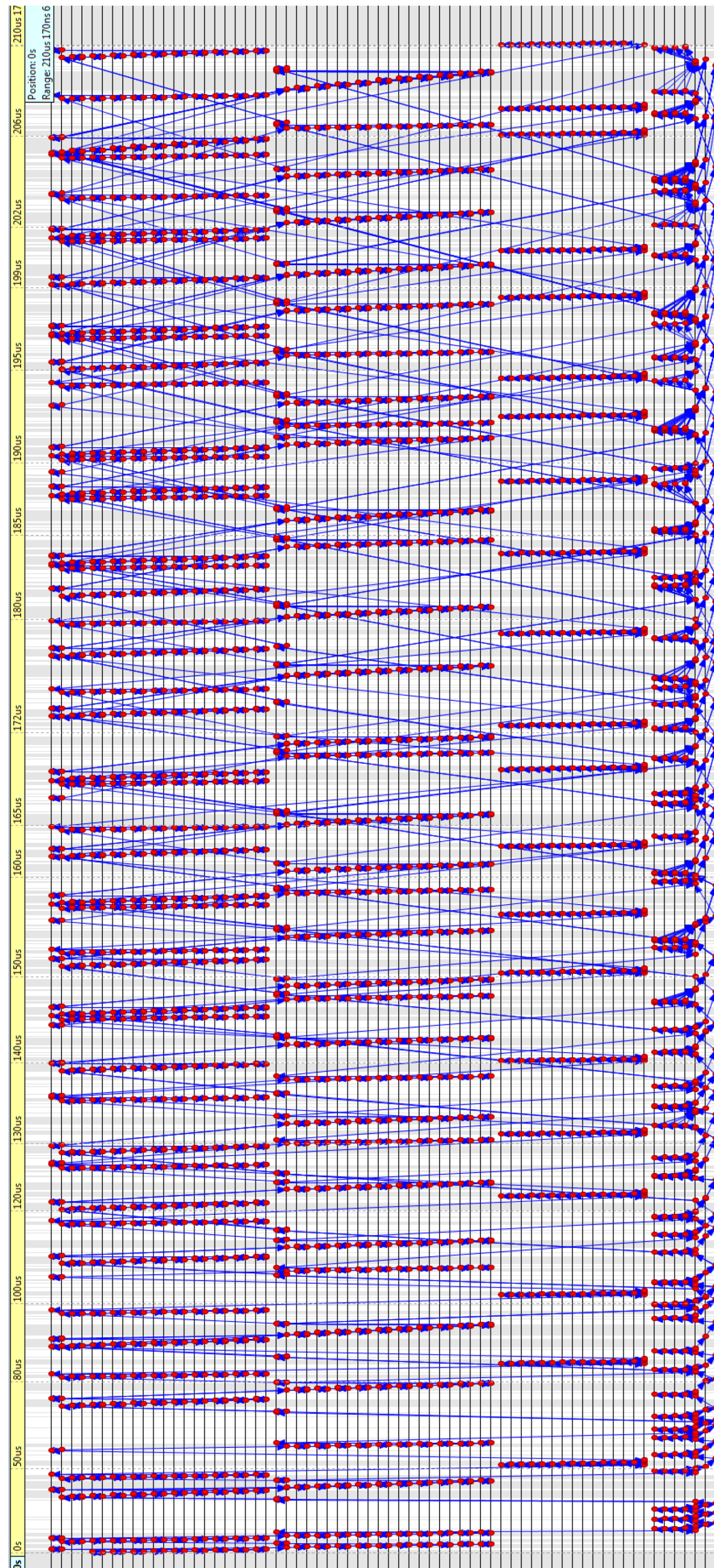


Figure 4.14: Unconstrained All Message Event Trace

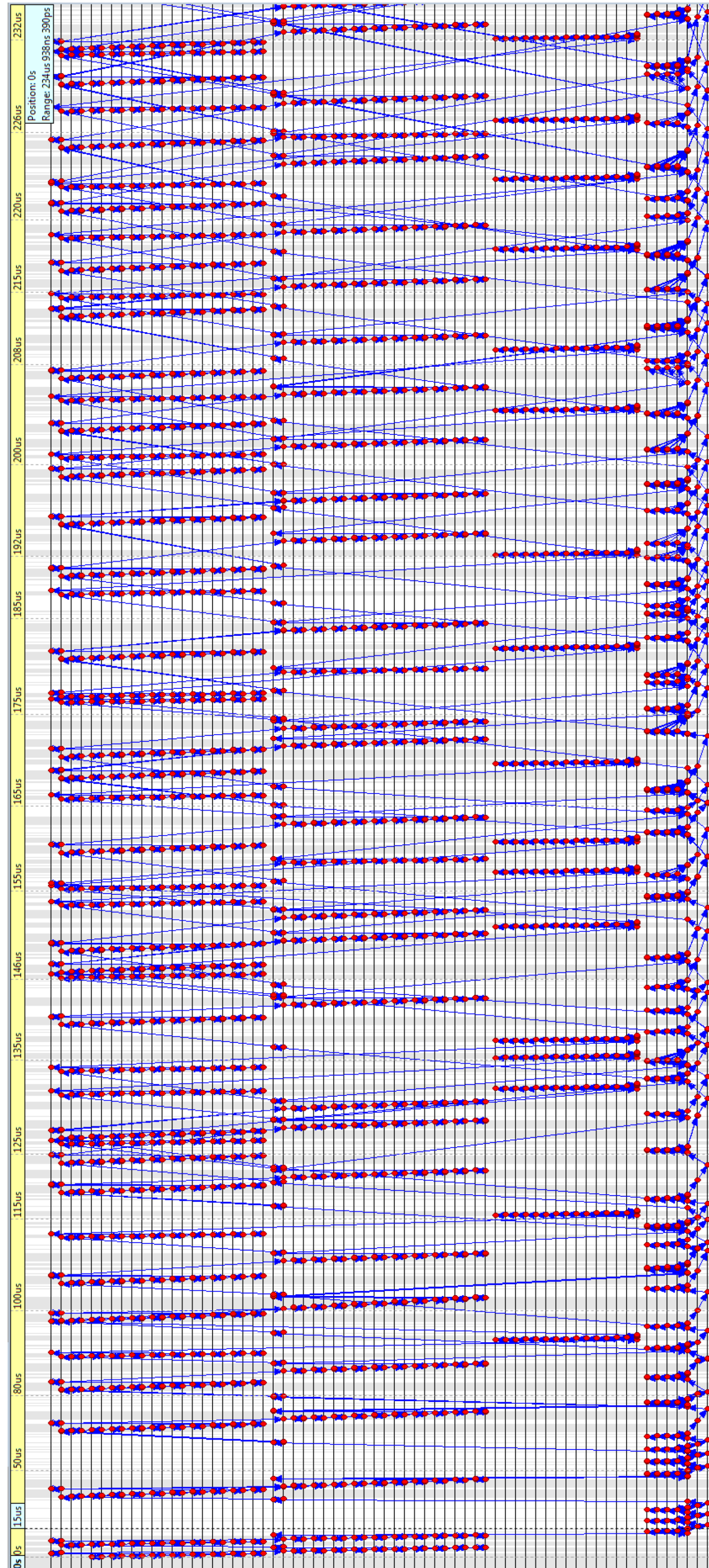


FIGURE 4.15: Condition-Event Constrained All Message Event Trace

Figures 4.14 and 4.15 show the highly overlapping event lifetimes in the model. This is important to note in the discussion of event times. Table 4.4 shows the comparison of message times between unconstrained and condition-event matrix constrained messages in the model. The event overlap causes summation of times to appear serially values greater that the total simulation time. Figure 4.16 shows the message lifetime and data transmission overlap for the *valueSend* message.

TABLE 4.4: Cumulative Message Times (seconds)

Simulation Type	Message	Cumulative Transition Time	Percent Reduction
Unconstrained	valueSend	0.020798200	—
	newRecord	0.029009200	—
	newEpoch	0.038415500	—
Constrained	valueSend	0.000347925	98.327%
	newRecord	0.001801480	93.790%
	newEpoch	0.002379040	93.807%

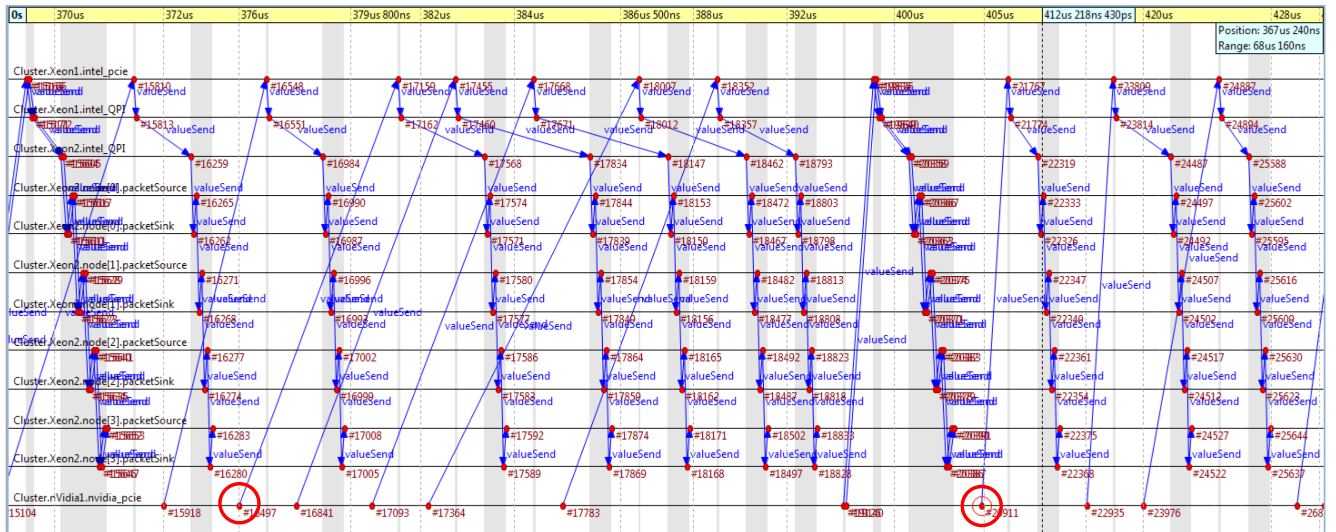


FIGURE 4.16: *valueSend* Message Overlap Event Trace

Thus, in the model, and in the real world as well, message lifetimes are highly overlapped and there is total time and work time. Reviewing a portion of the *valueSend* message cascade this overlap can be complex as shown in Figure 4.16 for event lifetimes from event 16497 up to, but not including, event 20911 (highlighted by the red circles). This overlap is expected in a multi-core chip and is also expected from a cluster implementation of multiple PEs per cluster. By comparing the number of message transmissions and the measured message life-times, it is straight-forward to evaluate the ratio of work saved by comparing the number of messages and the corresponding message latency time reduction to each other.

4.4 Analysis

This simulation demonstrated the successful interaction between software algorithms as mapped to PEs for a heterogeneous cluster using the ContinuousNN RITA condition-event controls for a continuous neural-network prediction algorithm and the significant message reduction that can be achieved by a properly configured condition-event matrix. The scope was limited to a cluster of 2 Xeon processors, a nVidia co-processor and a "mini-cluster" of 3 ARM7 processors. The heterogeneous environment is a good test environment for evaluating the objective of showing that once characteristics of an application are analyzed, and mapped to execution units, an allocation of PEs can be done with a resultant reduction in message load and an increase in application execution time while maintaining the safety and liveness of the system.

It was noted during development of this simulation focused on NoC that flit communication was specifically tied to the data channels of each processor type and as such the interaction between PEs became secondary. A trade was made to only capture the flow of data, rather than simulate flit communication. After examination of the HNoCS simulation based on OMNeT++, several of the HNoCS simulation constructs were found to be limited to only simulating flit processing within a processor and not extensible to clusters of heterogeneous chips. HNoCS was very useful as a reference model for the simulation created for this work. Referring back to Table 3.20, which contains a summary of $\mathcal{L}_{\mathcal{X}}$ empirical values, emphasis on NoC times would not be significant to demonstrate the time savings where the PCIe latencies dwarfed the NoC flit transmission times. Thus it was these PCIe times and message counts captured as output of the simulation; which, in real life, is where the optimization occurs in heterogeneous systems.

By varying the gating statistics for *valueSend*, *newRecord*, and *newEpoch* to allow more messages through (thus resulting in unconstrained message passing) did not give any special relation. The increases were linear and the slope could be adjusted at will by controlling the statistical guards like band pass filters. Other network parameters described in §3.4.6 are, over time, constant. Aberrations can occur which begin to affect the temporal aspect of the condition-event matrix. This has been recognized, discussed, and codified and would be a topic that requires a separate study.

From the timing analysis in the model, even with a 98% reduction in messages sent, the continuous neural network model did not suffer any degradation as the messages arrived well within the time period needed as compared to the times noted in Wallace, Turchenko, et.al.[WTS+13].

The model did have a high number of intra-processor messaging with *newTraining*, *advanceNN*, *NNcompute*{*0..13*}, and *ARM7*{*0..3*} providing the “other” process interaction. While these messages did take time, they were not significant in causing the condition-event matrix from performing as it was intended. Where “noisy neighbors” would cause issue is if there was a system blocking file or network I/O occurring during training of, or calculation of, the new neural network. This would be true with any co-processing and is not unique to this simulation.

Chapter 5

Summary, Conclusions, and Future Work

5.1 Summary

THIS thesis covered the topics of mathematics, integrated circuit design, cloud system management, programming, compiler design, and network-on-chip low-level to long-haul world-wide TCP/IP high-level networking. The core of work done in these topics considered the use of the Regulated Isomorphic Temporal Architecture (RITA) condition-event methods that achieved the goal of minimizing "chatter" between CE communications allowing for maximal application processing time. When using the word "chatter" it is important to note that it is not used as a dismissive. System "chatter" can be quite useful if it carries *information* semantics. All too often this communication is highly redundant event messages for "heart-beat" or "keep-alive" status or flow control messages that do not have any flow control effect. By examination and control of the conditions that create communication events, the actual information content increases as data volume decreases. Thus the work in this thesis was designing a regulation method to remove these redundant events from the most basic communication — messages between processors using a NoC — all the way up to the data packets exchanged between data-centers on a TCP/IP network — whether it be a LAN, CAN, or WAN — so only actual, *informational* data that needs processing interrupts an application for input or output.

After the initial overview of prior work in §1, §2 details the theoretical background needed to for the formal foundation that allows RITA to operate at any strata of computing. In §3 extensive work was done to provide a mapping from the theoretical to

available implementations. As the area of cluster computing, grid computing, and cloud computing are all rather synonymous, and each has its own jargon and preconceived framework, this thesis provided an agnostic method to differentiate the workflow control processing and the ontology for federated, distributed systems allowing mapping of the algorithm components across the computing fabric without use of too many currently fashionable terms tied to specific products in the market. In §4 a simulation of a cloud computing cluster was used to demonstrate and calculate the savings in processing time that RITA can provide. The OMNeT++ discrete event simulator used is a cycle accurate simulator that allowed messaging latencies to be captured for the highly-overlapped message transmission lifetimes. From the results of simulation the measurements taken demonstrate that the great reduction in "chatter" allowed increased processing time for application code.

5.2 Conclusions

THIS thesis has provided the following contributions to work on cloud system federated, distributed processing:

- A definition, mathematical basis, and usage for RITA
- A unique delayed-differential equation formula for heterogeneous processor cost functions
- The new syntax and semantics for a RITA description language
- New WorkFlow Control Pattern (WCP) and Decision-Point (DP) mapping algorithms for auto-partitioning applications across heterogeneous processing environments
- A new ontology for WCP and DP for RITA distributed flow-control for a semantic network
- Creation of decision-point algorithm definitions and DP scoring method to assign algorithms to CE types.
- A new extension of the Lotka-Volterra equations allowing predictive value assessments for establishment of steady state, maximal messaging in a RITA system

The concept of RITA has been in existence in the author's mind since 1984 and over the decades has matured to the work in this thesis. In each instantiation — avionics, middleware messaging, and cloud computing — the depth, formality, and general applicability to any federated, distributed application has matured. In each instance, the

reduction in message traffic has optimized processing. In §3.4.7 the novel use of Lotka-Volterra equations were extended to add predictive message rate capacity which had not been attempted in prior work.

This work has successfully demonstrated both theory and practice in achieving application optimization through large improvements (98% in the example chosen for the simulation) in application performance through a reduction in message "chatter" while not reducing application performance.

5.3 Future Research

FUTURE work from this thesis would include:

- Creating a multi-data center model using multiples of condition-event controlled clusters. This would require expansion of the model in §4 to simulate a full data center. This would require use of multiple simulation models that would have to be run in stages as the OMNeT++ system would be impractical to execute from the NoC to long-haul TCP/IP level in a single simulation.

- Perform an analysis to determine application or algorithm data sensitivity. While a DDE has been used to find the optimal message rate, this calculation would have to be used with other control-chart derived values for algorithm sensitivity to rate and data volume variations for general use case for RITA. This topic would be a quest for a unified optimization equation balancing the need for data input and how sparse data could be before an application or algorithm would not operate correctly.

- Currently there is no heterogeneous compiler using the RITA methods. Creating a general purpose compiler using the LLVM framework would be a long term (on the order of years), large effort (on the order of 10 persons) project.

- There is some fault tolerance in the methods used in RITA specifications where exceptions can be raised when data should have arrived or if data is out of range, but there needs to be a general fault tolerance mechanism for data loss in a RITA system or series of systems. In other words, the question is, "How 'lossy' or independent can a system be and still be considered effectively federated and can that that be captured in a RITA specification?"

- In §3.2.2 the workflow control patterns could lead to template classes to be used with RITA. Using the Go language from Google would be the best application

language as the both RITA and Go derive their semantics from *Communicating Sequential Processes* by C.A.R. Hoare.

These are the main topics that would make-up the bulk of future work. There are several smaller topics that would be interesting. One that could have helped immensely in developing programs used in this work would be an integrated RITA/Go IDE. While not a research topic it would be an important tool and would reduced the many hand-checks required to ensure that all semantic conditions were satisfied from the RITA specification.

Appendix A

RITA Language Syntax

RITA.G4

```
/*
** Regulated Isomorphic Temporal Architecture (RITA) grammar
** (c) 2000-2014 Richard Wallace
** Comments follow C language form.
** RITA is NOT a general purpose language.  It is a special purpose language.
*/
grammar RITA ;

rita_system
    : rita_unit+
    ;

rita_unit
    : with_use_stmt+ | system_stmt | control_stmt | comment
    ;

with_use_stmt
    : with_clause | use_clause
    ;

with_clause
    : 'with' QUOTE_CLAUSE ( ',' QUOTE_CLAUSE )* ';'
    ;

use_clause
    : 'use' QUOTE_CLAUSE ( ',' QUOTE_CLAUSE )* ';'
    ;

system_stmt
    : 'system' VALID_NAME '{' declarative_clause*
                              condition_clause+
                              guard_clause+
                              vector_clause+
                          '}'
    ;

declarative_clause
    : event_clause
    | data_clause
```

```
40      ;
41
42  event_clause
43      : 'event' '(' event_type ',' data_type ')' ':' VALID_NAME '(' value ')' ';'
44      ;
45
46  data_clause
47      : data_type ':' VALID_NAME ( ( '(' value ')' )
48      | ( '[' ( INTEGER | VALID_NAME ) ']' ) )? ';'
49      ;
50
51  condition_clause
52      : 'condition' VALID_NAME '{' statement+ '}'
53      ;
54
55  guard_clause
56      : 'guard' VALID_NAME '{' statement+ '}'
57      ;
58
59  vector_clause
60      : 'vector' VALID_NAME '{' guard_stmt condition_stmt+ result_stmt '}'
61      ;
62  guard_stmt
63      : 'guard' ':' VALID_NAME ';'
64      ;
65  condition_stmt
66      : 'condition' '(' bool_type ')' ':' ( VALID_NAME | 'null' ) ';'
67      ;
68  result_stmt
69      : 'result' ':'  ( expr | 'null' ) ';'
70      ;
71
72  control_stmt
73      : 'control' control_body
74      ;
75  control_body
76      : '{'
77            event_clause+
78            'begin' control_line+  'end'
79        '}'
80      ;
81  control_line
82      : VALID_NAME '<-' sys_or_event  (',' sys_or_event )* ';'
83  //    Since a system_name and event_name are both valid names, it will be
84  //    up to the production rules to assure the following syntax:
85  //    system_name <- system_name | event_name (, system_name | event_name)*
86      ;
87  sys_or_event
88      : VALID_NAME ( '(' value ')' )?
89  //   In the case of the valid name being a system name, there is no value part
90  //   but if the valid name is an event name, there is an optional value.
91      ;
92
93  comment
94      : LINE_COMMENT | BLOCK_COMMENT
95      ;
96
97  event_type
98      : 'setat'
99      | 'spike'
100     | 'trans'
101     ;
```

```
102
103 bool_type
104     : 'not'
105     | 'and'
106     | 'xor'
107     | 'or'
108     ;
109
110 data_type : 'bool'
111           | 'byte'
112           | 'int2'
113           | 'int4'
114           | 'int8'
115           | 'float4'
116           | 'float8'
117           | 'time'
118           | 'string'
119           ;
120
121 statement : if_stmt
122           | switch_stmt
123           | assignment_stmt
124           | return_stmt
125           | user_function_call
126           | break_stmt
127           | null_stmt
128           ;
129
130 if_stmt
131     : 'if' '(' expr ')' '{' statement+  '}' ( 'else' '{' statement+ '}' )*
132     ;
133
134 switch_stmt
135     : 'switch' VALID_NAME '{'  case_stmt+ '}'
136     ;
137 case_stmt
138     : 'case' value ':' statement+
139     ;
140
141 assignment_stmt
142     : VALID_NAME '=' expr ';'
143     ;
144
145 return_stmt
146     : 'return' VALID_NAME ';'
147     ;
148
149 user_function_call
150     : ( VALID_NAME '=' )? VALID_NAME ( '(' VALID_NAME | value ( ',' ( VALID_NAME | value ) )* ')' ) ';'
151     ;
152
153 break_stmt : 'break' VALID_NAME ';' ;
154
155 null_stmt : 'null' ';' ;
156
157 expr : VALID_NAME '(' exprList? ')' // func call like f(), f(x), f(1,2)
158      | expr '[' expr ']'            // array index like a[i], a[i][j]
159      | '-' expr                     // unary minus
160      | '!' expr                     // boolean not
161      | expr ('*' | '/') expr
162      | expr ('+' | '-') expr
163      | expr ('<=' | '<' | '>' | '>=' ) expr
```

```
164       | expr '&' expr // bitwise and
165       | expr '^' expr // bitwise exclusive or
166       | expr '|' expr // bitwise or
167       | expr ( '==' | '!=' ) expr    // equality comparison (lowest priority op)
168       | expr 'and' expr // logical and
169       | expr 'or'  expr // logical or
170       | VALID_NAME                   // variable reference
171       | value
172       | '(' expr ')'
173       ;
174
175 exprList : expr (',' expr)* ; // arg list
176
177 value
178     : INTEGER
179     | FLOAT
180     | TRUE_V //'TRUE'
181     | FALSE_V //'FALSE'
182     | QUOTE_CLAUSE
183     ;
184
185 //////////////////////////////////////
186
187 fragment
188     DIGIT : '0' .. '9' ;
189
190 fragment
191     LETTER : 'a' .. 'z' | 'A' ..'Z' ;
192
193 //////////////////////////////////////
194
195 TRUE_V
196     : [Tt][Rr][Uu][Ee] ;
197
198 FALSE_V
199     : [Ff][Aa][Ll][Ss][Ee] ;
200
201 QUOTE_CLAUSE
202     :  '"' (~["])*? '"'
203     ;
204
205 VALID_NAME
206     : LETTER | ( LETTER (LETTER | DIGIT | '_')* )
207     ;
208
209 BLOCK_COMMENT
210     : '/*' (~['*/'])* '*/' -> skip
211     ;
212
213 LINE_COMMENT
214     : '//' ~[\r\n]* -> skip
215     ;
216
217 FLOAT : DIGIT* '.' DIGIT*
218       |       '.' DIGIT+
219       ;
220
221 INTEGER : DIGIT+ ;
222
223 WS : [\u0020\t\r\n]+ -> skip ; //Skip space, tab, return, and new-line.
224
225 //
```

```
226 // The following are not implemented.
227 //
228 // For non-printable characters, special names are used:
229 //  \SOH, \STX, \ETX, \EOT, \ENQ, \ACK, \BEL, \BS,
230 //  \HT, \LF, \VT, \FF, \CR, \SO, \SI, \DLE,
231 //  \DC1, \DC2, \DC3, \DC4, \NAK, \SYN, \ETB, \CAN,
232 //  \EM, \SUB, \ESC, \FS, \GS, \RS, \US, \DEL
233 //
234 //  The escape characters for a double quote or back-slash in
235 //  quoted text
236 //
237 //  ESC : '\\"' | '\\\\' ;
238 //
```

RITA.G4

Appendix B

RITA Language Example Specification

In Listing B.1 the elements of a RITA *system, guard, vector,* and *resultant* are seen. The structure is detailed by line in Listing B.1:

- Lines 1-2: Linkage to application code functions that can run as detached processes.
- Lines 6-9: Declaration of events and variables for *TEST_SYSTEM_1*. Note the two events that have a named canonical event form, and their data type as an event occurrence has data associated with it.
- Lines 11-25: Declaration of conditions. Note that *INITIAL_CHECK* contains an application call to *user_function_1* that runs as a detached process while *user_function_2* is a blocking call for *FLOW*. The latter, while allowed, is not a preferred method of invoking application code.
- Lines 27-37: Declaration of guards. As $G \in C$, these look very much like conditions as they should. The primary usage is to short-circuit the evaluation of conditions in a vector. As such these are very simple gating controls.
- Lines 40-52: Declaration of vectors. Note the order of guard, condition(s), and result as these are the implementation of the condition event matrix. The boolean operation attached to each condition match the op_n in the condition event matrix. Given that there may be no result from an evaluation note the null condition in *VECTOR_2* as an example. Note the output from *VECTOR_1* is *SystemOn* and *null* in *VECTOR_2*. These two events stimulate further action in the RITA system.
- Lines 59-104: This is the second system in the RITA event system configuration. Of note is vector *System2_Looper* that has the event *Update* propagating a new

value of the event. The constructs in *TEST_SYSTEM_2* are like *TEST_SYSTEM_1* and their descriptions are not repeated.

- Lines 106-116: This is the RITA systems linkage. Note these are parallel statements as each system is a detached process. The symbol "`<-`" is read as, "Assign output from the right-hand system, or event, to the left hand system." Note that multiple right-hand systems may be specified thus, "`system1 <- system2, system3`."

```
1  with "../application/user_app.hpp";
2  use  "user_function_1", "user_function_2";
3
4  system TEST_SYSTEM_1
5  {
6    event(trans, bool) : SystemValue1(false);
7    int4        : Requests(0);
8    time        : FlowDuration("500ms");
9    float8      : FlowRate(0.0);
10
11   condition INITIAL_CHECK {
12     if( SystemOn == true ) {
13       user_function_1();
14       return INITIAL_CHECK_TRUE;
15     }
16     return INITIAL_CHECK_FALSE;
17   } // INITIAL_CHECK
18
19   condition FLOW {
20     FlowRate = user_function_2(FlowDuration);
21     if( FlowRate < 5 ){
22       return FLOW_TRUE; }
23     else {
24       return FLOW_FALSE; }
25   } // FLOW
26
27   guard SystemOn {
28     if( !SystemOn ) { break VECTOR_1; }
29     return SystemOn_TRUE;
30   } //SystemOn
31
32    guard Update {
33     if( Update <= 10 ) {
34       return Update_TRUE;
35     }
36     return Update_FALSE;
37   }
38
39   // SystemOn is true at start. Vector will activate.
40   vector VECTOR_1 {
41     guard          : SystemOn;
42     condition(and) : INITIAL_CHECK;
43     condition(or)  : FLOW;
44     result         : SystemOn1(true);
```

```
45    } // VECTOR_1
46
47    vector VECTOR_2 {
48      guard          : Update;
49      condition(and) : null;
50      condition(or)  : INITIAL_CHECK;
51      result         : CheckEvents(true);
52    } //VECTOR_2
53
54  } // TEST_SYSTEM_1
55
56  // ++
57  // TEST_SYSTEM_2
58  // --
59  system TEST_SYSTEM_2
60  {
61    int2 : update_value(0);
62    int2 : tmp_update(0);
63    int2 : Requests(0);
64
65    condition SetUpCheckEvents {
66      if( update_value <= 10 ){
67        tmp_update = update_value + 6;
68      }
69      tmp_update = 0;
70      return SetUpCheckEvents_TRUE;
71    }
72
73    condition AddUpdate {
74      update_value = update_value + 1;
75      return AddUpdate_TRUE;
76    }
77
78    guard SystemOn1 {
79      if( SystemOn1 == true) {
80        return SystemOn_TRUE; }
81      else {
82        break START_CheckEvents;}
83    }
84
85    guard CheckEvents {
86      if( Requests < 10) {
87        Requests = Requests + 1;
88        return CheckEvents_TRUE;
89      }
90      return CheckEvents_FALSE;
91    } //CheckEvents
92
93    vector START_CheckEvents {
94      guard          : SystemOn1;
95      condition(and) : SetUpCheckEvents;
96      result         : Update(tmp_update);
97    }
98
99    vector System2_Looper {
```

```
100      guard          : CheckEvents;
101      condition(and) : AddUpdate;
102      result         : Update( update_value + 1);
103    }
104  } // TEST_SYSTEM_2
105
106  control {
107    event(setat, bool) : SystemOn(false);
108    event(setat, bool) : SystemOn1(false);
109    event(spike, int4) : CheckEvents(0);
110    event(trans, int4) : Update(0);
111
112    begin
113      TEST_SYSTEM_1 <- SystemOn(true);
114      TEST_SYSTEM_2 <- TEST_SYSTEM_1;
115      TEST_SYSTEM_1 <- TEST_SYSTEM_2;
116    end
117  }
```

Listing B.1: RITA Code Fragment

Appendix C

Formula Elaboration

C.1 The Malthusian model

Thomas Robert Malthus FRS: 1766–1834, English clergyman, political economist and demographer. Malthus (1798) suggested a model for the birth and death rates proportional to the population:

$$B(p;t) = bp(t); \text{ and } D(p;t) = dp(t);$$

where b and d are constants, so

$$\frac{dp}{dt} = (b-d)p = \gamma p$$

where γ is a constant defined by: $\gamma = b - d$. This is the **growth rate**.
The population, p, at time t is solved by the equation:

$$p(t) = p(t_0)e^{\gamma(t-t0)}$$

This model is limited and it predicts an unbounded population increase for $\gamma > 0$. As show in Figure C.1.

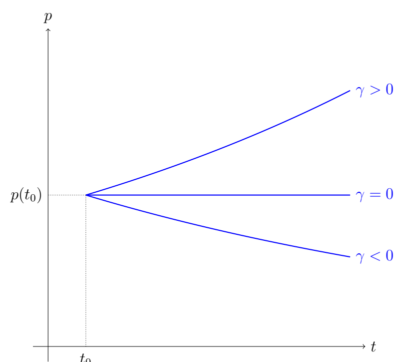


FIGURE C.1: Malthusian Growth Rate

Appendix D

Computing Element OWL XML

xr.xml

```
1  <?xml version="1.0"?>
2  <!DOCTYPE rdf:RDF [
3       <!ENTITY owl  "http://www.w3.org/2002/07/owl#" >
4       <!ENTITY xsd  "http://www.w3.org/2001/XMLSchema#" >
5     ]>
6  <rdf:RDF
7    xmlns:owl = "http://www.w3.org/2002/07/owl#"
8    xmlns:rdf = "http://www.w3.org/1999/02/22-rdf-syntax-ns#"
9    xmlns:rdfs= "http://www.w3.org/2000/01/rdf-schema#"
10   xmlns:xsd = "http://www.w3.org/2001/XMLSchema#">
11   <xs:simpleType name="memSizeKB">
12     <xs:restriction base="xs:int">
13       <xs:minInclusive value="1"/>
14       <rdfs:comment>Upper limit is a function of the system ability to
15         address memory based on word size.
16         xs:maxInclusive value=?????
17       </rdfs:comment>
18     </xs:restriction>
19   </xs:simpleType>
20   <owl:Ontology rdf:about="">
21     <rdfs:comment>Xrita is the eXtensible Regulated Isomorphic Temporal
22       Architecture. Created to support RITA communication between federated,
23       distributed computing elements.
24     </rdfs:comment>
25     <rdfs:label>Xrita Computing Element Ontology</rdfs:label>
26   </owl:Ontology>
27   <owl:Class rdf:ID="ComputingElement" />
28   <owl:Class rdf:ID="Network">
29     <rdfs:subClassof rdf:resouce="#ComputingElement"/>
30       <owl:Restriction>
31         <owl:onProperty rdf:resource="#networkType" />
32         <owl:allValuesFrom>
33           <owl:Class>
34             <owl:oneOf rdf:parseType="Collection">
35               <owl:Thing rdf:about="#CrayInterconnect" />
36               <owl:Thing rdf:about="#Custom" />
37               <owl:Thing rdf:about="#10-G" />
38               <owl:Thing rdf:about="#Gig-E" />
39               <owl:Thing rdf:about="#Infiniband" />
```

```
40            </owl:oneOf>
41          </owl:Class>
42        </owl:allValuesFrom>
43      </owl:Restriction>
44  </owl:Class>
45  <owl:Class rdf:ID="FPGA">
46    <rdfs:subClassof rdf:resource="#ComputingElement"/>
47  </owl:Class>
48  <owl:Class rdf:ID="DSP">
49    <rdfs:subClassof rdf:resource="#ComputingElement"/>
50      <owl:Restriction>
51        <owl:onProperty rdf:resource="#hasCores" />
52        <owl:cardinality rdf:datatype="&xsd;nonNegativeInteger">1</owl:cardinality>
53      </owl:Restriction>
54  </owl:Class>
55  <owl:Class rdf:ID="GPP">
56    <rdfs:subClassof rdf:resource="#ComputingElement"/>
57      <owl:Restriction>
58        <owl:onProperty rdf:resource="#hasCores" />
59        <owl:cardinality rdf:datatype="&xsd;nonNegativeInteger">1</owl:cardinality>
60      </owl:Restriction>
61  </owl:Class>
62  <owl:Class rdf:ID="GPU">
63    <rdfs:subClassof rdf:resource="#ComputingElement"/>
64      <owl:Restriction>
65        <owl:onProperty rdf:resource="#hasCores" />
66        <owl:cardinality rdf:datatype="&xsd;nonNegativeInteger">1</owl:cardinality>
67      </owl:Restriction>
68  </owl:Class>
69  <owl:Class rdf:ID="HSA">
70    <rdfs:subClassof rdf:resource="#ComputingElement"/>
71      <owl:Restriction>
72        <owl:onProperty rdf:resource="#hasCores" />
73        <owl:cardinality rdf:datatype="&xsd;nonNegativeInteger">1</owl:cardinality>
74      </owl:Restriction>
75  </owl:Class>
76  <owl:Class rdf:ID="HSA">
77    <rdfs:subClassof rdf:resource="#ComputingElement"/>
78      <owl:Restriction>
79        <owl:onProperty rdf:resource="#hasCores" />
80        <owl:cardinality rdf:datatype="&xsd;nonNegativeInteger">1</owl:cardinality>
81      </owl:Restriction>
82  </owl:Class>
83  <owl:ObjectProperty rdf:ID="hasCEDescriptor">
84    <rdf:type rdf:resource="&owl;TransitiveProperty" />
85    <rdfs:domain rdf:resource="#ComputingElement" />
86    </owl:ObjectProperty>
87  <owl:ObjectProperty rdf:ID="#hasCPU">
88    <rdf:type rdf:resource="&owl;FunctionalProperty" />
89    <rdfs:subPropertyOf rdf:resource="#hasCEDescriptor" />
90  </owl:ObjectProperty>
91  <owl:ObjectProperty rdf:ID="#hasCores">
92    <rdf:type rdf:resource="&owl;FunctionalProperty" />
93    <rdfs:subPropertyOf rdf:resource="#hasCPU" />
94    <rdfs:range rdf:resource="#MIC" />
95    <rdfs:range rdf:resource="#HSA" />
96    <rdfs:range rdf:resource="#GPU" />
97    <rdfs:range rdf:resource="#GPP" />
98    <rdfs:range rdf:resource="#DSP" />
99    <owl:Restriction>
100     <owl:onProperty rdf:resource="#hasCores" />
101     <owl:minCardinality rdf:datatype="&xsd;nonNegativeInteger">
```

```
102         1
103       </owl:minCardinality>
104     </owl:Restriction>
105   </owl:ObjectProperty>
106   <owl:ObjectProperty rdf:ID="#hasMemory">
107     <rdf:type rdf:resource="&owl;FunctionalProperty" />
108     <rdfs:subPropertyOf rdf:resource="#hasCEDescriptor" />
109     <owl:minCardinality rdf:datatype="&xsd;nonNegativeInteger">
110       #memSizeKB
111     </owl:minCardinality>
112   </owl:ObjectProperty>
113   <owl:ObjectProperty rdf:ID="#hasL1Cache">
114     <rdf:type rdf:resource="&owl;FunctionalProperty" />
115     <rdfs:subPropertyOf rdf:resource="#hasMemory" />
116   </owl:ObjectProperty>
117   <owl:ObjectProperty rdf:ID="#hasL2Cache">
118     <rdf:type rdf:resource="&owl;FunctionalProperty" />
119     <rdfs:subPropertyOf rdf:resource="#hasMemory" />
120   </owl:ObjectProperty>
121   <owl:ObjectProperty rdf:ID="#hasL3Cache">
122     <rdf:type rdf:resource="&owl;FunctionalProperty" />
123     <rdfs:subPropertyOf rdf:resource="#hasMemory" />
124   </owl:ObjectProperty>
125   <owl:ObjectProperty rdf:ID="#hasRAM">
126     <rdf:type rdf:resource="&owl;FunctionalProperty" />
127     <rdfs:subPropertyOf rdf:resource="#hasMemory" />
128   </owl:ObjectProperty>
129   <owl:ObjectProperty rdf:ID="#hasNetwork">
130     <rdf:type rdf:resource="&owl;FunctionalProperty" />
131     <rdfs:subPropertyOf rdf:resource="#hasCEDescriptor" />
132     <owl:Restriction>
133       <owl:onProperty rdf:resource="#hasNetwork" />
134       <owl:minCardinality rdf:datatype="&xsd;nonNegativeInteger">
135         1
136       </owl:minCardinality>
137     </owl:Restriction>
138   </owl:ObjectProperty>
139   <owl:DatatypeProperty rdf:ID="networkType">
140     <rdfs:domain rdf:resource="#Network" />
141     <rdfs:range  rdf:resource="#Network" />
142   </owl:DatatypeProperty>
143 </rdf:RDF>
```

xr.xml

Appendix E

VXDL BNF

VXDL_BNF.xml

```
1  <vxdl query> ::= "virtual grid" <name> [<time-to-use>] "{" <vg> "}"
2
3  <name> ::= <string>
4
5  <time-to-use> ::= "start" <date-time> "for" <total-time>
6
7  <date-time> ::= <date> " " <time>
8
9  <total-time> ::= <number>
10
11 <vg> ::= (<resource> | <group>)*
12
13 <resource> ::= "resource" "(" <name> ")" "{"
14   ["function" <elementary-functions>]
15   ["parameters" <resource-parameters>]
16   ["software" <software-list>]
17   ["anchor" <location>] "}"
18
19 <group> ::= "group" "(" <name> ")" "{""size" <value-number>[
20   "function" <elementary-functions>["anchor" <location>][<vg>] "}"
21
22 <value-number> ::= "(" <number> "," <number> ")" |
23   "(" "min" <number> ")" | "(" "max" <number> ")"
24
25 <value-freq> ::= "(" "min" <number> "GHz" ")"
26   | "(" "max" <number> "GHz" ")"
27   | "(" <number> "GHz" "," <number> "GHz" ")"
28
29 <value-mem> ::= "(" "min" <number> <men-unit> ")"
30   | "(" "max" <number> <men-unit> ")"
31   | "(" <number> <men-unit> "," <number> <men-unit> ")"
32
33 <value-band> ::= "(" "min" <number> <band-unit> ")"
34   | "(" "max" <number> <band-unit> ")"
35   | "(" <number> <band-unit> "," <number><band-unit> ")"
36
37 <value-lat> ::= "(" "min" <number> <lat-unit> ")"
38   | "(" "max" <number> <lat-unit> ")"
39   | "(" <number> <lat-unit> "," <number> <lat-unit> ")"
```

```
40
41  <men-unit> ::= "MB" | "GB" | "TB"
42
43  <band-unit> ::= "Kb/s" | "Mb/s" | "Gb/s"
44
45  <lat-unit> ::= "us" | "ms" | "s"
46
47  <location> ::= <string>
48
49  <elementary-functions> ::= <function> ("," <function> )*
50
51  <function> ::= "endpoint" | "aquisition" | "storage"
52    | "computing" | "visualization" | "network_sensor"
53    | "router" "(" "ports" <ports> ")"
54
55  <ports> ::= <number>
56
57  <resource-parameters> ::= <parameters> ("," <parameters>)*
58
59  <parameters> ::= "cpu_frequency" <value-freq>
60    | "cpu_mips" <value-number> | "hd_size" <value-mem>
61    | "memory_ram" <value-mem>
62    | "vms_per_node" <value-number>
63    | "cpu_processors" <value-number>
64
65  <software-list> ::= <software> ("," <software>)*
66
67  <software> ::= <string>
68    ["virtual topology" <name> "{" <links> "}"]
69
70  <links> ::= (<link>)+
71
72  <link> ::= "link" "(" <name> ")" "{" <link-parameters> "}"
73
74  <link-parameters> ::= <link-parameter>
75    ("," <link-parameter>)*
76
77  <link-parameter> ::= "bandwidth" <value-band>
78    | "latency" <value-lat>
79    | "between" "[" <components-links> "]"
80    | "direction" <direction>
81
82  <direction> ::= "uni" | "bi"
83
84  <components-links> ::= <pair> ("," <pair>)*
85
86  <pair> ::= "(" <component> "," <component> ")"
87
88  <component> ::= <name> | <name> "port" <number>
89    ["virtual timeline" <name> "{" (<timeline>)+ "}"]
90
91  <timeline> ::= <time-name> "=" (<start> | <after>)
92
93  <time-name> ::= <string>
94
95  <start> ::= "start" "(" <components-list> ")" [<until>]
96
97  <after> ::= "after" "(" <time-name-list> ")" <start>
98
99  <components-list> ::= <component-name>
100   ("," <component-name>)*
101
```

```
102 <component-name> ::= <name>
103
104 <time-name-list> ::= <time-name> ("," <time-name>)*
105
106 <until> ::= (<computation> | <transfer>)+
107
108 <computation> ::= "computation" "(" <total-time> ")"
109
110 <transfer> ::= "transfer" "(" <value-mem> ")"
```

VXDL_BNF.xml

Appendix F

Tower of Hanoi in Java and Go

F.1 Java Implementation

TowerOfHanoi.java

```
1  import java.io.*;
2  import java.lang.*;
3  import java.util.*;
4
5  class TowerOfHanoi
6  {
7    static int movecount = 0;
8    static public void Solve2DiscsTOH(Stack source, Stack temp, Stack dest)
9    {
10     temp.push(source.pop());
11     movecount++;
12     PrintStacks();
13     dest.push(source.pop());
14     movecount++;
15     PrintStacks();
16     dest.push(temp.pop());
17     movecount++;
18     PrintStacks();
19   }
20
21   static public int SolveTOH(int nDiscs, Stack source, Stack temp, Stack dest)
22   {
23     if (nDiscs <= 4)
24     {
25       if ((nDiscs % 2) == 0)
26       {
27         Solve2DiscsTOH(source, temp, dest);
28         nDiscs = nDiscs - 1;
29         if (nDiscs == 1)
30         return 1;
31
32         temp.push(source.pop());
33         movecount++;
34         PrintStacks();
35         //new source is dest, new temp is source, new dest is temp;
```

```
36          Solve2DiscsTOH(dest, source, temp);
37          dest.push(source.pop());
38          movecount++;
39          PrintStacks();
40          //new source is temp, new temp is source, new dest is dest;
41          SolveTOH(nDiscs, temp, source, dest);
42        }
43        else
44        {
45          if (nDiscs == 1)
46          return -1;
47          Solve2DiscsTOH(source, dest, temp);
48          nDiscs = nDiscs - 1;
49          dest.push(source.pop());
50          movecount++;
51          PrintStacks();
52          Solve2DiscsTOH(temp, source, dest);
53        }
54        return 1;
55      }
56      else if (nDiscs >= 5)
57      {
58        SolveTOH(nDiscs - 2, source, temp, dest);
59        temp.push(source.pop());
60        movecount++;
61        PrintStacks();
62        SolveTOH(nDiscs - 2, dest, source, temp);
63        dest.push(source.pop());
64        movecount++;
65        PrintStacks();
66        SolveTOH(nDiscs - 1, temp, source, dest);
67      }
68      return 1;
69    }
70
71    static public Stack A = new Stack();
72    static public Stack B = new Stack();
73    static public Stack C = new Stack();
74
75    static public void PrintStacks()
76    {
77      if (countA != A.size() ||
78      countB != B.size() ||
79      countC != C.size())
80      {
81        int diffA = A.size() - countA;
82        int diffB = B.size() - countB;
83        int diffC = C.size() - countC;
84        if (diffA == 1)
85        {
86          if (diffB == -1)
87          System.out.print("Move Disc " + A.peek() + " From B To A");
88          else
89          System.out.print("Move Disc " + A.peek() + " From C To A");
90        }
91        else if (diffB == 1)
92        {
93          if (diffA == -1)
94          System.out.print("Move Disc " + B.peek() + " From A To B");
95          else
96          System.out.print("Move Disc " + B.peek() + " From C To B");
97        }
```

```
98        else //if (diffC == 1)
99        {
100         if (diffA == -1)
101         System.out.print("Move Disc " + C.peek() + " From A To C");
102         else
103         System.out.print("Move Disc " + C.peek() + " From B To C");
104       }
105       countA = A.size();
106       countB = B.size();
107       countC = C.size();
108       System.out.println();
109     }
110
111     PrintStack(A);
112     System.out.print(" , ");
113     PrintStack(B);
114     System.out.print(" , ");
115     PrintStack(C);
116     System.out.print(" , ");
117   }
118
119   static int countA = 0;
120   static int countB = 0;
121   static int countC = 0;
122
123   static public void PrintStack(Stack s)
124   {
125     System.out.print(s.toString());
126   }
127
128   public static void main(String[] args)
129   {
130     try
131     {
132       while (true)
133       {
134         System.out.print("\nEnter the number of discs (-1 to exit): ");
135
136         int maxdisc = 0;
137         String inpstring = "";
138
139         InputStreamReader input = new InputStreamReader(System.in);
140         BufferedReader reader = new BufferedReader(input);
141         inpstring = reader.readLine();
142
143         movecount = 0;
144         maxdisc = Integer.parseInt(inpstring);
145         if (maxdisc == -1)
146         {
147           System.out.println("Good Bye!");
148           return;
149         }
150         if (maxdisc <= 1 || maxdisc >= 10)
151         {
152           System.out.println("Enter between 2 - 9");
153           continue;
154         }
155         for (int i = maxdisc; i >= 1; i--)
156         A.push(i);
157         countA = A.size();
158         countB = B.size();
159         countC = C.size();
```

```
160         PrintStacks();
161         SolveTOH(maxdisc, A, B, C);
162         System.out.println("Total Moves = " + movecount);
163         while (C.size() > 0)
164         C.pop();
165       }
166     }
167     catch (Exception e)
168     {
169       e.printStackTrace();
170     }
171   }
172 }
```

TowerOfHanoi.java

F.2 Go Implementation

hanoi.go

```
1  package main
2
3  import (
4          "flag"
5          "fmt"
6  )
7
8  const (
9          A = iota
10         B
11         C
12 )
13
14 type stack struct {
15         Name  string
16         Discs []int
17 }
18
19 func (s *stack) pop() int {
20         stack_length := len(s.Discs)
21         disc := s.Discs[stack_length-1]
22         s.Discs = s.Discs[0 : stack_length-1]
23         return disc
24 }
25
26 func (s *stack) push(disc int) {
27         s.Discs = append(s.Discs, disc)
28 }
29
30 var (
31         discs  = flag.Int("discs", 7, "Number of discs to use.")
32         moves  = 0
33         stacks []*stack
34 )
35
36 func init() {
37         flag.Parse()
38 }
```

```
39
40  func main() {
41          fmt.Printf("Solving Towers of Hanoi for %d discs\n", *discs)
42
43          // Create the stacks
44          stacks = make([]*stack, 3, 3)
45          stacks[A] = &stack{"A", make([]int, 0, *discs)}
46          stacks[B] = &stack{"B", make([]int, 0, *discs)}
47          stacks[C] = &stack{"C", make([]int, 0, *discs)}
48
49          // Create the discs
50          for i := 0; i < *discs; i++ {
51                  stacks[A].Discs = append(stacks[A].Discs, i)
52          }
53
54          hanoi(*discs, A, C)
55  }
56
57  func via_stack(src, dst int) int {
58          return A + B + C - src - dst
59  }
60
61  func hanoi(d, src, dst int) {
62          via := via_stack(src, dst)
63          if d > 1 {
64                  hanoi(d-1, src, via)
65          }
66
67          moves++
68          stacks[dst].push(stacks[src].pop())
69          fmt.Printf("Move #%d: %s -> %s\n", moves, stacks[src].Name, stacks[dst].Name)
70
71          if d > 1 {
72                  hanoi(d-1, via, dst)
73          }
74  }
```

hanoi.go

Appendix G

Example Optimization Report for Intel OpenMP*

optimization output

```
1
2   Begin optimization report for:
3   exafmm_kernel<1L>::M2L_V(std::valarray<std::valarray<real>> &, const
4   std::valarray<real> *, const std::valarray<std::valarray<real>> &,
5   std::size_t)
6
7       Report from: Vector optimizations [vec]
8
9
10  LOOP BEGIN at ../src/exafmm.cpp(41,2)
11      remark #15388: vectorization support: reference F64 has aligned access
12  [ ../src/exafmm.cpp(42,14) ]
13      remark #15388: vectorization support: reference F64 has aligned access
14  [ ../src/exafmm.cpp(42,14) ]
15      remark #15388: vectorization support: reference F64 has aligned access
16  [ ../src/exafmm.cpp(42,14) ]
17      remark #15388: vectorization support: reference F64 has aligned access
18  [ ../src/exafmm.cpp(42,14) ]
19      remark #15388: vectorization support: reference F64 has aligned access
20  [ ../src/exafmm.cpp(42,14) ]
21      remark #15388: vectorization support: reference F64 has aligned access
22  [ ../src/exafmm.cpp(42,14) ]
23      remark #15388: vectorization support: reference _M_data has aligned
24  access [ ../src/exafmm.cpp(57,12) ]
25      remark #15399: vectorization support: unroll factor set to 4
26      remark #15301: SIMD LOOP WAS VECTORIZED
27      remark #15448: unmasked aligned unit stride loads: 6
28      remark #15449: unmasked aligned unit stride stores: 1
29      remark #15475: --- begin vector loop cost summary ---
30      remark #15476: scalar loop cost: 104
31      remark #15477: vector loop cost: 81.000
32      remark #15478: estimated potential speedup: 4.400
33      remark #15479: lightweight vector operations: 21
34      remark #15488: --- end vector loop cost summary ---
35  LOOP END
36
37  LOOP BEGIN at ../src/exafmm.cpp(41,2)
38  <Remainder>
39      remark #15388: vectorization support: reference F64 has aligned access
40  [ ../src/exafmm.cpp(42,14) ]
41      remark #15388: vectorization support: reference F64 has aligned access
42  [ ../src/exafmm.cpp(42,14) ]
43      remark #15388: vectorization support: reference F64 has aligned access
44  [ ../src/exafmm.cpp(42,14) ]
45      remark #15388: vectorization support: reference F64 has aligned access
46  [ ../src/exafmm.cpp(42,14) ]
47      remark #15388: vectorization support: reference F64 has aligned access
48  [ ../src/exafmm.cpp(42,14) ]
49      remark #15388: vectorization support: reference F64 has aligned access
50  [ ../src/exafmm.cpp(42,14) ]
51      remark #15388: vectorization support: reference _M_data has aligned
52  access [ ../src/exafmm.cpp(57,12) ]
53      remark #15301: REMAINDER LOOP WAS VECTORIZED
54  LOOP END
55
56  LOOP BEGIN at ../src/exafmm.cpp(41,2)
57  <Remainder>
```

```
58   LOOP END
59
60   LOOP BEGIN at       ../src/exafmm.cpp(91,4)
61          remark       #15388:       vectorization       support: reference F64 has aligned access
62   [       ../src/exafmm.cpp(92,17) ]
63          remark       #15388:       vectorization       support: reference _M_data has aligned
64   access              [       ../src/exafmm.cpp(92,8)       ]
65          remark       #15399:       vectorization       support: unroll       factor set to       4
66          remark       #15301:       SIMD LOOP       WAS       VECTORIZED
67          remark       #15449:       unmasked aligned unit       stride stores: 2
68          remark       #15475:       ---       begin       vector loop       cost summary ---
69          remark       #15476:       scalar loop       cost:       9
70          remark       #15477:       vector loop       cost:       6.000
71          remark       #15478:       estimated       potential       speedup: 5.000
72          remark       #15479:       lightweight       vector operations: 6
73          remark       #15488:       ---       end       vector loop       cost summary ---
74   LOOP END
75
76   LOOP BEGIN at       ../src/exafmm.cpp(91,4)
77   <Remainder>
78          remark       #15388:       vectorization       support: reference F64 has aligned access
79   [       ../src/exafmm.cpp(92,17) ]
80          remark       #15388:       vectorization       support: reference _M_data has aligned
81   access              [       ../src/exafmm.cpp(92,8)       ]
82          remark       #15301:       REMAINDER       LOOP WAS VECTORIZED
83   LOOP END
84
85   LOOP BEGIN at       ../src/exafmm.cpp(91,4)
86   <Remainder>
87   LOOP END
88
89   LOOP BEGIN at       ../src/exafmm.cpp(110,6)
90          remark       #15388:       vectorization       support: reference F64 has aligned access
91   [       ../src/exafmm.cpp(111,7) ]
92          remark       #15388:       vectorization       support: reference F64 has aligned access
93   [       ../src/exafmm.cpp(111,7) ]
94          remark       #15388:       vectorization       support: reference F64 has aligned access
95   [       ../src/exafmm.cpp(111,7) ]
96          remark       #15388:       vectorization       support: reference F64 has aligned access
97   [       ../src/exafmm.cpp(111,7) ]
98          remark       #15388:       vectorization       support: reference F64 has aligned access
99   [       ../src/exafmm.cpp(111,7) ]
100         remark       #15388:       vectorization       support: reference F64 has aligned access
101  [       ../src/exafmm.cpp(111,7) ]
102         remark       #15399:       vectorization       support: unroll       factor set to       4
103         remark       #15301:       SIMD LOOP       WAS       VECTORIZED
104         remark       #15448:       unmasked aligned unit       stride loads:       4
105         remark       #15449:       unmasked aligned unit       stride stores: 2
106         remark       #15475:       ---       begin       vector loop       cost summary ---
107         remark       #15476:       scalar loop       cost:       17
108         remark       #15477:       vector loop       cost:       14.000
109         remark       #15478:       estimated       potential       speedup: 4.180
110         remark       #15479:       lightweight       vector operations: 14
111         remark       #15488:       ---       end       vector loop       cost summary ---
112  LOOP END
113
114  LOOP BEGIN at       ../src/exafmm.cpp(110,6)
115  <Remainder>
116         remark       #15388:       vectorization       support: reference F64 has aligned access
117  [       ../src/exafmm.cpp(111,7) ]
118         remark       #15388:       vectorization       support: reference F64 has aligned access
119  [       ../src/exafmm.cpp(111,7) ]
120         remark       #15388:       vectorization       support: reference F64 has aligned access
121  [       ../src/exafmm.cpp(111,7) ]
122         remark       #15388:       vectorization       support: reference F64 has aligned access
123  [       ../src/exafmm.cpp(111,7) ]
124         remark       #15388:       vectorization       support: reference F64 has aligned access
125  [       ../src/exafmm.cpp(111,7) ]
126         remark       #15388:       vectorization       support: reference F64 has aligned access
127  [       ../src/exafmm.cpp(111,7) ]
128         remark       #15301:       REMAINDER       LOOP WAS VECTORIZED
129  LOOP END
130
131  LOOP BEGIN at       ../src/exafmm.cpp(110,6)
132  <Remainder>
133  LOOP END
134
135  LOOP BEGIN at       ../src/exafmm.cpp(119,4)
136         remark       #15388:       vectorization       support: reference F64 has aligned access
137  [       ../src/exafmm.cpp(120,7) ]
138         remark       #15388:       vectorization       support: reference _M_data has aligned
139  access              [       ../src/exafmm.cpp(120,7) ]
140         remark       #15388:       vectorization       support: reference F64 has aligned access
141  [       ../src/exafmm.cpp(121,7) ]
142         remark       #15388:       vectorization       support: reference _M_data has aligned
143  access              [       ../src/exafmm.cpp(121,7) ]
144         remark       #15301:       SIMD LOOP       WAS       VECTORIZED
145         remark       #15448:       unmasked aligned unit       stride loads:       2
146         remark       #15449:       unmasked aligned unit       stride stores: 2
147         remark       #15475:       ---       begin       vector loop       cost summary ---
148         remark       #15476:       scalar loop       cost:       11
149         remark       #15477:       vector loop       cost:       1.500
150         remark       #15478:       estimated       potential       speedup: 7.330
151         remark       #15479:       lightweight       vector operations: 6
152         remark       #15488:       ---       end       vector loop       cost summary ---
153  LOOP END
154
```

```
155  LOOP BEGIN at        ../src/exafmm.cpp(119,4)
156  <Remainder>
157  LOOP END
158
159  LOOP BEGIN at        /usr/include/c++/4.9.0/bits/valarray_array.h(207,2)        inlined
160  into ../src/exafmm.cpp(124,2)
161          remark        #15527:        loop was not vectorized: function        call to        free cannot        be
162  vectorized               [       ../src/new.cpp(29,2) ]
163  LOOP END
164  ===========================================================================
165
166  Begin        optimization report        for: exafmm_kernel<1L>::cart2sph(real        &, real        &,
167  real        &, std::valarray<real> *)
168
169                  Report from: Vector        optimizations        [vec]
170
171
172  LOOP BEGIN at        /usr/include/c++/4.9.0/bits/valarray_after.h(298,4)        inlined
173  into ../src/exafmm.cpp(130,25)
174  <Peeled>
175  LOOP END
176
177  LOOP BEGIN at        /usr/include/c++/4.9.0/bits/valarray_after.h(298,4)        inlined
178  into ../src/exafmm.cpp(130,25)
179          remark        #15388:        vectorization        support: reference __s.30798 has aligned
180  access               [        /usr/include/c++/4.9.0/bits/valarray_after.h(299,6)        ]
181          remark        #15388:        vectorization        support: reference __s.30798 has aligned
182  access               [        /usr/include/c++/4.9.0/bits/valarray_after.h(299,6)        ]
183          remark        #15399:        vectorization        support: unroll        factor set to        4
184          remark        #15301:        REVERSED LOOP        WAS        VECTORIZED
185          remark        #15442:        entire loop        may        be executed        in remainder
186          remark        #15448:        unmasked aligned unit        stride loads:        1
187          remark        #15475:        ---        begin        vector loop        cost summary ---
188          remark        #15476:        scalar loop        cost:        10
189          remark        #15477:        vector loop        cost:        7.000
190          remark        #15478:        estimated        potential        speedup: 4.510
191          remark        #15479:        lightweight        vector operations: 7
192          remark        #15488:        ---        end        vector loop        cost summary ---
193  LOOP END
194
195  LOOP BEGIN at        /usr/include/c++/4.9.0/bits/valarray_after.h(298,4)        inlined
196  into ../src/exafmm.cpp(130,25)
197  <Remainder>
198          remark        #15388:        vectorization        support: reference __s.30798 has aligned
199  access               [        /usr/include/c++/4.9.0/bits/valarray_after.h(299,6)        ]
200          remark        #15388:        vectorization        support: reference __s.30798 has aligned
201  access               [        /usr/include/c++/4.9.0/bits/valarray_after.h(299,6)        ]
202          remark        #15301:        REMAINDER        LOOP WAS VECTORIZED
203  LOOP END
204
205  LOOP BEGIN at        /usr/include/c++/4.9.0/bits/valarray_after.h(298,4)        inlined
206  into ../src/exafmm.cpp(130,25)
207  <Remainder>
208  LOOP END
209  ===========================================================================
```

optimization output

Appendix H

High Speed TCP Variants

Various types of recent TCP implementations (since 2000) define changes in the size of a congestion window by requiring data from the network. Such congestion control protocols are divided into two groups, explicit and implicit.
List from http://kb.pert.geant.net/PERTKB/TcpHighSpeedVariants.

Explicit congestion control protocols that use explicit feedback from the router:

- Source Quench
- Explicit Congestion Notification (ECN) published by K. Ramakrishnan, S. Floyd, D. Black
- eXplicit Congestion Protocol (XCP) developed by Dina Katabi from MIT Computer Science and Artificial Intelligence Lab
- Rate Control Protocol (RCP) developed by Nandita Dukkaipati from Stanford University's Computer Systems Laboratory
- Quick-Start TCP, focusing on permission to use a large Initial Window

Implicit congestion control protocols that reply on implicit measurements of congestion such as loss or delay:

- HS-TCP (HighSpeed TCP) by Sally Floyd et al. uses modified AIMD parameters when the congestion window gets larger than some boundary.
- H-TCP by Doug Leith et al. from the Hamilton Institute
- TCP Vegas from the University of Arizona
- TCP Westwood proposed by UCLA Computer Science Department
- TCP Westwood+ from C3LAB at Politecnico de Bari
- Compound TCP by Microsoft (included in Microsoft Vista)

- FAST from Caltech
- BIC (Binary Increase Congestion Control)
- CUBIC
- Scalable TCP by Tom Kelly
- TCP Fusion by Kazumi Kaneko et al.
- YeAH-TCP from University of Roma
- Layered TCP
- SABUL
- Sync-TCP, Michele Weigle's Ph.D. work at UNC

Appendix I

Lotka–Volterra

The MATLAB code used for the Lotka–Volterra with Carrying Capacity graphs in §3.4.6.

RITA_PredPrey.m

```
1   function RITA_PredPrey()
2
3   % For this work, a redefinition of terms is done so the model works for
4   % latency:
5   % repPrey     -> System resource recovery rate (%) for processed messages.
6   % PredCoef    -> Message resource use coefficient.
7   % PredMort    -> Message resource recovery after receipt.
8   % repPredPrey -> Message processing creation rate per application running
9   %                on a single processor.
10  % m           -> Carrying capacity for messages.
11  %
12  % Initial conditions:
13  % Prey                  -> Message resources.
14  % Predator -> Initial number of messages created.
15
16      repPrey      =  1.00;   % Intrinsic rate of prey population increase
17      PredCoef     =  0.01;   % Predation rate coefficient
18      PredMort     =  1.00;   % Preditor mortality rate
19      repPredPrey  =  0.0195; % Reproduction rate of predators per 1 prey eaten
20      m            = 150;     % Carrying capacity
21      lags  = [ 1 ];   % Vvector of Taus (prior state).
22      tspan = [0 500]; % time span to integrate over (unitless)
23
24      sol = dde23(@ddefun,lags,@Myhistory,tspan);
25
26      y1 = sol.y(1,:); % note the y-solution is a row-wise matrix.
27      y2 = sol.y(2,:);
28      figure;
29      plot(y1,y2)
30      xlabel('Message Resources');
31      ylabel('Messages');
32
33      figure;
34      plot(sol.x,y1,sol.x,y2);
35      xlabel('Time');
36      legend 'Message Resources' 'Messages';
37
38      function  dYdt = ddefun(t,Y,Z)
39      % We define the function for the delay. the Y variable is the same as you
40      % should be used to from an ordinary differential equation. Z is the values
41      % of all the delayed variables.
42
43      % Using the computed latencies from Table 2.20, a random coefficient is
44      % added to the predation coefficient allowing latency to effect consumption
45      % of resources.  This value is not added to the lags array above as that
46      % would only produce a (T - t) from further in the past and does not
47      % account for accrued latency in the system.  As the true latency is only
48      % in the hundredths of a second the effect is not too great for the system
49      % that operating at per second granularity.
50          a = 1.10026025121950E-01;
51          b = 1.24518052195120E-01;
52          r = (b-a) * rand(1,1) + a;
53          y1 = Y(1); % Prey
54          y2 = Y(2); % Preditor
55          y1_tau1 = Z(1,1); %History prey value
56          y2_tau1 = Z(2,1); %History predator value
57          dy1dt = repPrey * y1 * ( 1 - y1 / m ) - (PredCoef * r) * y1 * y2;
58          dy2dt = repPredPrey * y1_tau1 * y2_tau1 - PredMort * y2;
59          dYdt = [dy1dt; dy2dt];
60      end %ddefun()
61
62      function y = Myhistory(t)
```

```
63       % A history function is needed where the dx/dt equals zero.  These are the
64       % initial conditions for y1 = prey (message resources) and y2 = predator
65       % (messages).
66          y = [ 100; 10 ];
67       end %Myhistory()
68
69   end %RITA_PredPrey()
```

RITA_PredPrey.m

Appendix J

OMNeT++ NED Descriptions

The OMNeT++ NED descriptions used for the heterogeneous PE simulation in §4.

Cluster.ned

```
//
// Xeon message Source Interface
//
moduleinterface Source_interface
{
    parameters:
                 // int srcId; // must be globally unique
     @display("i=block/source");
    gates:
        output out;
}

//
// Xeon message Sink Interface
//
moduleinterface Sink_interface
{
    parameters:
    @display("i=block/sink");
    gates:
        input in;
}

simple PacketSource like Source_interface
{
    @display("i=block/source");
    gates:
        output out;
        input crossbar;
}

simple PacketSink like Sink_interface
{
    parameters:
    @display("i=block/sink");
    gates:
        input in;
        output crossbar;
}

simple Intel_PCIe
{
    parameters:
        @display("i=block/layer");
    gates:
        input  prev;
        output next;
        inout  pcie;
}

simple Intel_QPI
{
    parameters:
        @display("i=block/layer");
    gates:
        input  prev;
        output next;
        inout  qpi;
}

simple nVidia_PCIe
{
```

```
63      parameters:
64          @display("i=block/layer");
65      gates:
66          inout b[]; // memory buffer, L2 or shading RAM.
67  }
68
69  simple nVidia_SMX
70  {
71      gates:
72          inout g;
73  }
74
75  simple ARM_L2
76  {
77      parameters:
78          @display("i=block/layer");
79      gates:
80          inout g[];
81          output out;
82          input  in;
83  }
84
85  simple ARMChip
86  {
87      gates:
88          inout g;
89  }
90
91
92  import RITA.PE.ARM.ARMChip;
93  import RITA.PE.ARM.ARM_L2;
94  import RITA.PE.Intel.Intel_PCIe;
95  import RITA.PE.Intel.Intel_QPI;
96  import RITA.PE.nVidia.nVidia_PCIe;
97  import RITA.PE.nVidia.nVidia_SMX;
98  import RITA.PE.sinks.PacketSink;
99  import RITA.PE.sources.PacketSource;
100 import ned.DelayChannel;
101 import ned.IdealChannel;
102
103 module Node_Xeon
104 {
105     parameters:
106         @display("i=block/cogwheel;bgb=160,198");
107     gates:
108         input prev[];
109         output next[];
110     types:
111     submodules:
112         packetSource: PacketSource {
113             @display("p=55,54");
114         }
115         packetSink: PacketSink {
116             @display("p=111,124");
117         }
118     connections allowunconnected:
119         packetSource.out --> next++;
120         packetSink.in <-- prev++;
121         packetSource.crossbar <-- packetSink.crossbar; // This models the ring bus in the Xeon chip
122 }
123
124 module Chip_Xeon
125 {
126     @display("bgb=392,381");
127     gates:
128         inout PCIe_port[10];
129         inout QPI_port[10];
130
131     types:
132         //
133         // Data rate for PCI-e 3.0 1 lane, 8Gbps theoretical, 7.88 measured.
134         // Latency (delay) is from gammer forums.  Range is 1000ns - 2000ns, e.g. 1-2ms
135         // For now, OMNeT++ will not allow mixing of IdealChannel and DatarateChannel,
136         // so mask this for now.
137         //        channel IODataRate extends DatarateChannel
138         //        {
139         //            datarate = 7.88Gbps;
140         //            delay = 1500ns;    // PCI is about 400ns;
141         //            ber = 1e-10;
142         //        }
143         channel IODataRate extends DelayChannel
144         {
145             delay = 1500ns;
146         }
147
148     submodules:
149         node[10]: Node_Xeon {
150             @display("p=31,124,ri,100,100");
151         }
152         intel_pcie: Intel_PCIe {
153             @display("p=291,115");
154         }
155         intel_QPI: Intel_QPI {
156             @display("p=129,38");
157         }
158     connections allowunconnected:
159         // As a design can not have two or more datarate channels on a connection path between two simple modules
```

```
160         // use IdealChannel (see OMNeT++ Manual). Use IdealChannel for internal (node to node) communication.
161         node[0].next++ --> IdealChannel --> node[1].prev++;
162         node[1].next++ --> IdealChannel --> node[2].prev++;
163         node[2].next++ --> IdealChannel --> node[3].prev++;
164         node[3].next++ --> IdealChannel --> node[4].prev++;
165         node[4].next++ --> IdealChannel --> node[5].prev++;
166         node[5].next++ --> IdealChannel --> node[6].prev++;
167         node[6].next++ --> IdealChannel --> node[7].prev++;
168         node[7].next++ --> IdealChannel --> node[8].prev++;
169         node[8].next++ --> IdealChannel --> node[9].prev++;
170         node[9].next++ --> IdealChannel --> intel_pcie.prev;
171
172         intel_pcie.pcie <--> IODataRate <--> PCIe_port[0];
173         intel_pcie.next --> IdealChannel --> intel_QPI.prev;
174
175         intel_QPI.qpi <--> IODataRate <--> QPI_port[0];
176         intel_QPI.next --> IdealChannel --> node[0].prev++;
177 }
178
179 module Chip_nVidia
180 {
181     @display("bgb=374,304;bgl=5");
182     gates:
183         inout port[2];
184     types:
185         //
186         // Data rate for PCI-e 3.0 1 lane, 8Gbps theoretical, 7.88 measured.
187         // Latency (delay) is from gammer forums.  Range is 1000ns - 2000ns, e.g. 1ms
188         // For now, OMNeT++ will not allow mixing of IdealChannel and DatarateChannel,
189         // so mask this for now.
190         //       channel IODataRate extends ned.DatarateChannel
191         //        {
192         //            datarate = 7.88Gbps;
193         //            delay = 1500ns;    // PCI is about 400ns;
194         //        }
195         channel IODataRate extends DelayChannel
196         {
197             delay = 1500ns;
198         }
199
200     submodules:
201         smx[15]: nVidia_SMX {
202             @display("p=20,20,ri,100,100");
203         }
204         nvidia_pcie: nVidia_PCIe {
205             @display("p=120,120");
206         }
207     connections allowunconnected:
208         smx[0].g <--> nvidia_pcie.b++;
209         smx[1].g <--> nvidia_pcie.b++;
210         smx[2].g <--> nvidia_pcie.b++;
211         smx[3].g <--> nvidia_pcie.b++;
212         smx[4].g <--> nvidia_pcie.b++;
213         smx[5].g <--> nvidia_pcie.b++;
214         smx[6].g <--> nvidia_pcie.b++;
215         smx[7].g <--> nvidia_pcie.b++;
216         smx[8].g <--> nvidia_pcie.b++;
217         smx[9].g <--> nvidia_pcie.b++;
218         smx[10].g <--> nvidia_pcie.b++;
219         smx[11].g <--> nvidia_pcie.b++;
220         smx[12].g <--> nvidia_pcie.b++;
221         smx[13].g <--> nvidia_pcie.b++;
222         smx[14].g <--> nvidia_pcie.b++;
223
224         nvidia_pcie.b++ <--> IODataRate <--> port[0];
225 }
226
227 module Chip_ARM
228 {
229     @display("bgl=2;bgb=374,306");
230     gates:
231         input in[];
232         output out[];
233         inout PCIe[100];
234
235     types:
236         channel L2DataRate extends ned.DatarateChannel
237         {
238             datarate = 1.6Gbps;
239             delay = 400ns;
240         }
241         channel IODataRate extends DelayChannel
242         {
243             delay = 1500ns;
244         }
245
246     submodules:
247         ARM7[4]: ARMChip {
248             @display("p=20,20,ri,100,100");
249         }
250         L2_Mem: ARM_L2 {
251             @display("p=120,120");
252         }
253
254     //
255     // Have to allow for memory access in this tightly connected ARM chip.  Searching the web found:
256     // <https://www.ruhr-uni-bochum.de/integriertesysteme/emuco/files/System_Level_Benchmarking_Analysis_of_the_Cortex_A9_MPCore.pdf>
```

```
257        // and
258        // <http://www.bsdcan.org/2014/schedule/attachments/281_2014_arm_superpages-paper.pdf>
259        // that gives results of around 400ns "average" value.  While it is a ARM"x" and not an ARM7, ARM7 data is hard to find.
260        //
261        connections allowunconnected:
262            in++ --> L2_Mem.in;
263            L2_Mem.g++ <--> L2DataRate <--> ARM7[0].g;
264            L2_Mem.g++ <--> L2DataRate <--> ARM7[1].g;
265            L2_Mem.g++ <--> L2DataRate <--> ARM7[2].g;
266            L2_Mem.g++ <--> L2DataRate <--> ARM7[3].g;
267            L2_Mem.g++ <--> IODataRate <--> PCIe[0];
268            L2_Mem.out --> L2DataRate --> out++;
269
270    }
271    //-----------------------------------------------------
272
273    network Cluster
274    {
275        @display("bgl=6;bgb=598,253");
276        types:
277            //  See https://software.intel.com/en-us/blogs/2014/01/28/
278            //  memory-latencies-on-intel-xeon-processor-e5-4600-and-e7-4800-product-families
279            channel QPIDataRate extends ned.DatarateChannel
280            {
281                datarate = 25.6Gbps;
282                delay = 231.67ns;
283            }
284            channel IODataRate extends DelayChannel
285            {
286                delay = 1500ns; //PCIe data rate
287            }
288        submodules:
289            Xeon1: Chip_Xeon {
290                @display("p=124,46");
291            }
292            Xeon2: Chip_Xeon {
293                @display("p=248,46");
294            }
295            nVidia1: Chip_nVidia {
296                @display("p=124,171");
297            }
298
299            MarvelXP1: Chip_ARM {
300                @display("p=248,171");
301            }
302            MarvelXP2: Chip_ARM {
303                @display("p=338,106");
304            }
305            MarvelXP3: Chip_ARM {
306                @display("p=417,171");
307            }
308        connections allowunconnected:
309            Xeon1.QPI_port[0] <--> QPIDataRate <--> Xeon2.QPI_port[0];
310            Xeon1.PCIe_port[0] <--> IODataRate <--> nVidia1.port[0];
311            Xeon2.PCIe_port[0] <--> IODataRate <--> MarvelXP1.PCIe++;
312            MarvelXP1.out++ --> IODataRate --> MarvelXP2.in++;
313            MarvelXP2.out++ --> IODataRate --> MarvelXP3.in++;
314            MarvelXP3.out++ --> IODataRate --> MarvelXP1.in++;
315    }
```

Cluster.ned

Bibliography

[ABB+12] Joshua Auerbach, David F. Bacon, Ioana Burcea, Perry Cheng, Stephen J. Fink, Rodric Rabbah, and Sunil Shukla. A compiler and runtime for heterogeneous computing. In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, pages 271–276, New York, NY, USA, 2012. ACM. URL: http://doi.acm.org/10.1145/2228360.2228411.

[ABC+06] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006. URL: http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html.

[ABCR10] Joshua Auerbach, David F. Bacon, Perry Cheng, and Rodric Rabbah. Lime: A java-compatible and synthesizable language for heterogeneous architectures. *SIGPLAN Notices*, 45(10):89–108, October 2010. URL: http://dx.doi.org/10.1145/1932682.1869469.

[ABD+09] Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine Yelick. A view of the parallel computing landscape. *Commun. ACM*, 52(10):56–67, October 2009. URL: http://doi.acm.org/10.1145/1562764.1562783.

[ADMX+12] Ra'ed Al-Dujaily, Terrence Mak, Fei Xia, Alexandre Yakovlev, and Maurizio Palesi. Embedded transitive closure network for runtime deadlock detection in networks-on-chip. *Parallel and Distributed Systems, IEEE Transactions on*, 23(7):1205–1215, July 2012. URL: http://dx.doi.org/10.1109/TPDS.2011.275.

[ADR14] Anthony Auer, Juergen Dingel, and Karen Rudie. Concurrency control generation for dynamic threads using discrete-event systems. *Science of Computer Programming*, 82(0):22 – 43, 2014. URL: http://dx.doi.org/10.1016/j.scico.2013.01.007.

[AF15] Tor M. Aamodt and Wilson W. L. Fung. Gpgpu-sim 3.x manual. Wiki, 2015. URL: http://gpgpu-sim.org/manual/index.php/GPGPU-Sim_3.x_Manual.

[AFB12] Tor M. Aamodt, Wilson W. L. Fung, and Andrew Boktor. Presentation slides, 2012. GPGPU-Sim simulator is sponsored by Tor Aamodt. URL: http://www.gpgpu-sim.org/.

[AFG+09] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. Above the clouds: A berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, Feb 2009. URL: http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.html.

[AFY+14] Rachata Ausavarungnirun, Chris Fallin, Xiangyao Yu, Kevin Chang, Greg Nazario, Reetuparna Das, Gabriel H. Loh, and Onur Mutlu. Improving energy efficiency of hierarchical rings via deflection routing. Technical Report 2014-002, Carnegie Mellon University, April 2014. URL: https://www.ece.cmu.edu/~safari/tr/tr-2014-002.pdf.

[AGM+90] S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215:403–410, 1990.

[Alw04] Vivek Alwayn. *Optical Network Design and Implementation*. Cisco Press, 2004.

[Ama14] Amazon. Amazon EC2 Spot Price History, 2014. URL: http://aws.typepad.com/aws/2011/07/ec2-spot-pricing-now-specific-to-each-availability-zone.html.

[APR05] Md. Imran Alman, Manjusha Pandey, and Siddharth S. Rautaray. A comprehensive survey on cloud computing. *International Journal of Information Technology and Computer Science(IJITCS)*, 7(2), 2105. URL: http://www.mecs-press.org/ijitcs/ijitcs-v7-n2/v7n2-9.html.

[Asm87] Søeren Asmussen. *Applied Probability and Queues*. Wiley, 1987.

[Bad13] Ashesh Badani. Openshift and openstack: A match made in the cloud, 2013. URL: http://www.redhat.com/about/news/archive/2013/2/openshift-and-openstack-a-match-made-in-the-cloud.

[BDH+10a] Andre R. Brodtkorb, Christopher Dyken, Trond R. Hagen, Jon M. Hjelmervik, and Olaf O. Storaasli. State-of-the-art in heterogeneous computing. *Sci. Program.*, 18(1):1–33, January 2010. URL: http://dl.acm.org/citation.cfm?id=1804799.1804800.

[BDH+10b] Andre R. Brodtkorb, Christopher Dyken, Trond R. Hagen, Jon M. Hjelmervik, and Olaf O. Storaasli. State-of-the-art in heterogeneous computing. *Scientific Programming*, 18(1):1–33, 2010. URL: http://dl.acm.org/citation.cfm?id=1804799.1804800.

[Bee95] Randall D. Beer. On the dynamics of small continuous-time recurrent neural networks. *Adaptive Behavior*, 3(4):469–509, January 1995. URL: http://dx.doi.org/10.1177/105971239500300405.

[Ben62] Václav E. Beneš. On rearrangeable three-stage connecting networks. *The Bell System Technical Journal*, 41(5):1481–1492, Sept 1962. URL: http://dx.doi.org/10.1002/j.1538-7305.1962.tb03990.x.

[BGDG+10] Andreas Berl, Erol Gelenbe, Marco Di Girolamo, Giovanni Giuliani, Hermann De Meer, Minh Quan Dang, and Kostas Pentikousis. Energy-efficient cloud computing. *The Computer Journal*, 53(7):1045–1051, 2010. URL: http://dx.doi.org/10.1093/comjnl/bxp080.

[BGH92] Dimitri P Bertsekas, Robert G Gallager, and Pierre Humblet. *Data networks*. Prentice-Hall International, 2nd edition, 1992.

[BHS09] Franz Baader, Ian Horrocks, and Ulrike Sattler. Description logics. In Steffen Staab and Rudi Studer, editors, *Handbook on Ontologies*, International Handbooks on Information Systems, pages 21–43. Springer Berlin Heidelberg, 2009. URL: http://dx.doi.org/10.1007/978-3-540-92673-3_1.

[Bir12] Kenneth P. Birman. Network perspective. In *Guide to Reliable Distributed Systems*, Texts in Computer Science, pages 101–143. Springer London, 2012. URL: http://dx.doi.org/10.1007/978-1-4471-2416-0_4.

[BJ70] G. Box and G. Jenkins. *Time Series Analysis: Forecasting and Control*. John Wiley and Sons, Hoboken, NJ, 1970.

[BKHH78] C. Gordon Bell, A. Kotok, T. N. Hastings, and R. Hill. The Evolution of the DECsystem 10. *Communications of the ACM*, 21(1):44–63, January 1978. URL: http://doi.acm.org/10.1145/359327.359335.

[BLHL01] Tim Berners-Lee, James Hendler, and Ora Lassila. The semantic web. *Scientific American*, 284(5):34–43, May 2001. URL: http://www.sciam.com/article.cfm?articleID=00048144-10D2-1C70-84A9809EC588EF21.

[BS13] Pierre Bomel and Marc Sevaux. Parallel deadlock detection and recovery for networks-on-chip dedicated to diffused computations. In *2013 Euromicro Conference on Digital System Design (DSD)*, pages 29–36, Sept 2013. URL: http://dx.doi.org/10.1109/DSD.2013.14.

[Bui09] Hai-Lam Bui. Survey and comparison of event query languages using practical examples. Diplomarbeit/diploma thesis, Institute of Computer Science, LMU, Munich, 2009. URL: http://www.pms.ifi.lmu.de/publikationen/#DA_Hai-Lam.Bui.

[Bun13] Chris Bunch. *Automated Configuration and Deployment of Applications in Heterogeneous Cloud Environments*. PhD thesis, Dept. Computer Science, UC Santa Barbara, 2013. URL: http://www.cs.ucsb.edu/research/tech_reports/reports/2013-02.pdf.

[BVF+12] Peter Bailis, Shivaram Venkataraman, Michael J Franklin, Joseph M Hellerstein, and Ion Stoica. Probabilistically bounded staleness for practical partial quorums. *Proceedings of the VLDB Endowment*, 5(8):776–787, 2012.

[BWF+96] F. Berman, R. Wolski, S. Figueira, J. Schopf, and G. Shao. Application-level scheduling on distributed heterogeneous networks. In *Supercomputing, 1996. Proceedings of the 1996 ACM/IEEE Conference on*, pages 39–39, 1996. URL: URL:http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1392910&isnumber=30314.

[CA12] Andrew S. Cassidy and Andreou G. Andreas. Beyond amdahl's law: An objective function that links multiprocessor performance gains to delay and energy. *IEEE Transactions on Computers*, 61(8):1110–1126, Aug 2012. URL: http://dx.doi.org/10.1109/TC.2011.169.

[Car11] Giulio Caravagna. *Formal modeling and simulation of biological systems with delays*. PhD thesis, Universitá di Pisa, 2011. URL: http://www.di.unipi.it/~caravagn/papers/caravagna-phd-thesis.pdf.

[CBA+05] Brent N. Chun, Philip Buonadonna, Alvin Auyoung, Chaki Ng, David C. Parkes, Jeffrey Shneidman, Alex C. Snoeren, and Amin Vahdat. Mirage: A microeconomic resource allocation system for sensornet testbeds. In *In Proceedings of the 2nd IEEE Workshop on Embedded Networked Sensors*, 2005.

[CcC+06] Donald Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Greg Seidman, Michael Stonebraker, Nesime Tatbul, and Stanley B. Zdonik. Monitoring streams - a new class of data management applications. In *VLDB*, pages 215–226. Morgan Kaufmann, 2006.

[CEvA11] Mani K. Chandy, Opher Etzion, and Rainer von Ammon. 10201 Executive Summary and Manifesto – Event Processing. In K. Mani Chandy, Opher Etzion, and Rainer von Ammon, editors, *Event Processing*, number 10201 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2011. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany. URL: http://drops.dagstuhl.de/opus/volltexte/2011/2985.

[CGB+11] B. Chandramouli, J. Goldstein, R. Barga, M. Riedewald, and I. Santos. Accurate latency estimation in a distributed event processing system. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pages 255–266, April 2011. URL: http://dx.doi.org/10.1109/ICDE.2011.5767926.

[CGS09] Adrian M Caulfield, Laura M Grupp, and Steven Swanson. Gordon: using flash memory to build fast, power-efficient clusters for data-intensive applications. *ACM Sigplan Notices*, 44(3):217–228, 2009.

[CHB+09] B. Chapman, Lei Huang, E. Biscondi, E. Stotzer, A. Shrivastava, and Alan Gatherer. Implementing openmp on a high performance embedded multicore mpsoc. In *IEEE International Symposium on Parallel Distributed Processing, 2009 (IPDPS 2009)*, pages 1–8, 2009. URL: http://dx.doi.org/10.1109/IPDPS.2009.5161107.

[Chu71] L.O. Chua. Memristor-the missing circuit element. *IEEE Transactions on Circuit Theory*, 18(5):507–519, Sep 1971. URL: http://dx.doi.org/10.1109/TCT.1971.1083337.

[CKK03] David Codl, Jaroslav Kačer, and Tomáš Koutný. Comparison-evaluation of Java-based discrete-time simulation tools. In *Proceedings of the 37th International Conference MOSIS-2003: Modelling and Simulation of Systems*, pages 125–130. Faculty of Information Technology of VUT

Brno, Department of Computer Science of FEEI VŠB-TU Ostrava, ASU, CSSS, Eurosim, and SCS, 2003. URL: http://www.kiv.zcu.cz/j-sim/Articles/Files/MOSIS2003-CodlKacerKoutny.PDF.

[CKP+93] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. Logp: Towards a realistic model of parallel computation. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '93, pages 1–12, 1993. URL: http://doi.acm.org/10.1145/155332.155333.

[CLB14] YuFeng Chen, ZhiWu Li, and Kamel Barkaoui. Maximally permissive liveness-enforcing supervisor with lowest implementation cost for flexible manufacturing systems. *Information Sciences*, 256(0):74 – 90, 2014. URL: http://dx.doi.org/10.1016/j.ins.2013.07.021.

[CLdMA12] Érika Cota, Marcelo Lubaszewski, and Alexandre de Morais Amory. *Reliability, Availability and Serviceability of Networks-on-Chip*. Springer, 2012. URL: www.springer.com.

[Clo53] Charles Clos. A study of non-blocking switching networks. *The Bell System Technical Journal*, 32(2):406–424, March 1953. URL: http://dx.doi.org/10.1002/j.1538-7305.1953.tb01433.x.

[Coh82] Jacob Willem Cohen. *The Single Server Queues*. Amsterdam: North-Holland, 1982.

[CS61] David Roxbee Cox and Walter Smith. *Queues*. London: Mathuen & Co, 1961.

[CS13] Tommaso Cucinotta and Aram Santogidis. Cloudnetsim-simulation of real-time cloud computing applications. In *Proceedings of the 4th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems*, 2013. URL: http://retis.sssup.it/waters2013/.

[CSJN05] Junwei Cao, Daniel P. Spooner, Stephen A. Jarvis, and Graham R. Nudd. Grid load balancing using intelligent agents. *Future Generation Computer Systems*, 21(1):135 – 149, 2005. URL: http://dx.doi.org/10.1016/j.future.2004.09.032.

[DC01] Parviz Doulai and Warren Cahill. Short-term price forecasting in electric energy market. In *2001 International Power Engineering Conference 17-19 May 2001, Singapore*, volume 21, pages 29–29, 2001. URL: http://dx.doi.org/10.1109/MPER.2001.4311253.

[DC14] Sheng Di and Franck Cappello. Gloudsim: Google trace based cloud simulator with virtual machines. *Software: Practice and Experience*, 2014. URL: http://dx.doi.org/10.1002/spe.2303.

[dCRN13] Otávio M. de Carvalho, Eduardo Roloff, and Philippe O. A. Navaux. A survey of the state-of-the-art in event processing. In *WSPPD 2013 - XI Workshop on Parallel and Distributed Processing*. Institute of Informatics, Porto Alegre, Brazil, 2013. URL: http://inf.ufrgs.br/gppd/wsppd/2013/papers/wsppd2013_submission_14.pdf.mod.pdf.

[DGM+11] Ron Dror, J. P. Grossman, Kenneth Mackenzie, Brian Towles, Edmond Chow, John Salmon, Cliff Young, Joseph Bank, Brannon Batson, David Shaw, Jeffrey S. Kuskin, Richard H. Larson, Mark A. Moraes, and David E. Shaw. Overcoming communication latency barriers in massively parallel scientific computation. *IEEE Micro*, 31(3):8–19, May 2011. URL: http://dx.doi.org/10.1109/MM.2011.38.

[DHR88] Roger Duke, Ian Hayes, and Gordon Rose. Verification of a cyclic retransmission protocol. Technical Report 92, University of Queensland, Queensland, Australia, 1988.

[Dij68] Edsger W. Dijkstra. Cooperating sequential processes. In F. Genuys, editor, *Programming Languages: NATO Advanced Study Institute*, pages 43–112. Academic Press, 1968.

[DS04] Mike Dean and Guus (Editors) Schreiber. Owl web ontology language reference, February 2004. URL: http://www.w3.org/TR/2004/REC-owl-ref-20040210/.

[DT03] William Dally and Brian Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[DWH14] Preethi P. Damodaran, Stefan Wallentowitz, and Andreas Herkersdorf. Distributed cooperative shared last-level caching in tiled multiprocessor system on chip. In *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*, pages 1–4, March 2014. URL: http://dx.doi.org/10.7873/DATE2014.096.

[EBB+11] Michael Eckert, François Bry, Simon Brodt, Olga Poppe, and Steffen Hausmann. A CEP Babelfish: Languages for Complex Event Processing and querying surveyed. In *Reasoning in Event-Based Distributed Systems*, volume 347 of *Studies in Computational Intelligence*, pages 47–70.

Springer Berlin Heidelberg, 2011. URL: http://dx.doi.org/10.1007/978-3-642-19724-6_3.

[Eil69] Samuel Eilon. Letter to the editor—a simpler proof of $L = \lambda W$. *Operations Research*, 17(5):915–917, 1969. URL: http://pubsonline.informs.org/doi/abs/10.1287/opre.17.5.915.

[Elm86] Ahmed K. Elmagarmid. A survey of distributed deadlock detection algorithms. *ACM SIGMOD Record*, 15(3):37–45, September 1986. URL: http://doi.acm.org/10.1145/15833.15837.

[FBK+12] Andrew D. Ferguson, Peter Bodik, Srikanth Kandula, Eric Boutin, and Rodrigo Fonseca. Jockey: Guaranteed job latency in data parallel clusters. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 99–112. ACM, 2012. URL: http://doi.acm.org/10.1145/2168836.2168847.

[FC14] Wu-chun Feng and Kirk Cameron. The green500 supercomputers site, 2014. URL visited 2014-04-12. URL: http://www.green500.org/.

[Fei10] Dror Feitelson. Parallel workloads archive, 2010. URL visited 2014-04-12. URL: http://www.cs.huji.ac.il/labs/parallel/workload/.

[Fin13] FindTheBest. Find the best, 2013. URL: http://cloud-computing.findthebest.com/.

[FJ05] M. Frigo and S.G. Johnson. The design and implementation of fftw3. *Proceedings of the IEEE*, 93(2):216–231, Feb 2005. URL: http://dx.doi.org/10.1109/JPROC.2004.840301.

[FK97] Ian Foster and Carl Kesselman. Globus: A metacomputing infrastructure toolkit. *International Journal of High Performance Computing Applications*, 11(2):115–128, 1997. URL: http://hpc.sagepub.com/content/11/2/115.abstract.

[Fly66] Michael J. Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12):1901–1909, Dec 1966. URL: http://dx.doi.org/10.1109/PROC.1966.5273.

[For13] Distributed Management Task Force. Open virtualization format specification, 2013. URL: http://dmtf.org/standards/ovf.

[Fou12] Open Grid Fourm. Distributed resource management application api version 2 (drmaa), 2012. URL: http://www.gridforum.org/standards/.

[FRZ14] Nick Feamster, Jennifer Rexford, and Ellen Zegura. The road to sdn: An intellectual history of programmable networks. *SIGCOMM Comput. Commun. Rev.*, 44(2):87–98, April 2014. URL: http://doi.acm.org/10.1145/2602204.2602219.

[FSH13] Irina Fedotova, Eduard Siemens, and Hao Hu. A high-precision time handling library. In *ICNS 2013 : The Ninth International Conference on Networking and Services*, pages 193–199. IARIA, 3 2013.

[FSS07] Alexandra Fedorova, Margo Seltzer, and Michael D. Smith. Improving performance isolation on chip multiprocessors via an operating system scheduler. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, PACT '07, pages 25–38, Washington, DC, USA, 2007. IEEE Computer Society. URL: http://dx.doi.org/10.1109/PACT.2007.40.

[FTL+02] James Frey, Todd Tannenbaum, Miron Livny, Ian Foster, and Steven Tuecke. Condor-G: A computation management agent for multi-institutional grids. *Cluster Computing*, 5(3):237–246, 2002. URL: http://dx.doi.org/10.1023/A%3A1015617019423.

[FTR13] Andrew Flahive, David Taniar, and Wenny Rahayu. Ontology as a service (oaas): a case for sub-ontology merging on the cloud. *The Journal of Supercomputing*, 65(1):185–216, 2013. URL: http://dx.doi.org/10.1007/s11227-011-0711-4.

[FWB07] Xiaobo Fan, Wolf-Dietrich Weber, and Luiz Andre Barroso. Power provisioning for a warehouse-sized computer. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 13–23, New York, NY, USA, 2007. ACM. URL: http://doi.acm.org/10.1145/1250662.1250665.

[Gal08] Antony Galton. Temporal logic. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Stanford University, 2008. URL: http://plato.stanford.edu/archives/fall2008/entries/logic-temporal/.

[GBGN93] R.B. Grove, D.S. Blickstein, K.D. Glossop, and W.B. Noyce. Gem optimizing compilers for alpha axp systems. In *Compcon Spring '93, Digest of Papers.*, pages 465–473, Feb 1993. URL: http://dx.doi.org/10.1109/CMPCON.1993.289715.

[GBL10] Cristian Grozea, Zorana Bankovic, and Pavel Laskov. Fpga vs. multi-core cpus vs. gpus: Hands-on experience with a sorting application. In *Facing the Multicore-Challenge*, volume 6310 of *Lecture Notes in Computer Science*, pages 105–117. Springer Berlin Heidelberg, 2010. URL: http://dx.doi.org/10.1007/978-3-642-16233-6_12.

[GBRW13] Richard Geary, Robert Bassett, Phil Rothwell, and James Williams. Cloud awards, 2013. URL: http://www.cloud-awards.com/.

[GL85] Ronald I. Greenberg and Charles E. Leiserson. Randomized routing on fat-trees. Technical Report ADA166965, MIT, Cambridge Lab, September 1985. Defense Technical Information Center. URL: http://www.dtic.mil/get-tr-doc/pdf?AD=ADA166965.

[Goo13] Google. Cloud computing providers, 2013. URL: http://en.wikipedia.org/wiki/Category:Cloud_computing_providers.

[Gop92] Kondalsamy Gopalsamy. *Stability and oscillations in delay differential equations of population dynamics*. Springer, 1992.

[Gre89] Ronald I. Greenberg. *Efficient Interconnection Schemes for VLSI and Parallel Computation*. PhD thesis, Electrical Engineering and Computer Science Dept., MIT, 1989. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.49.295&rep=rep1&type=pdf.

[GSA90] Randall L. Geiger, Noel R. Strader, and P. E. Allen. *VLSI design techniques for analog and digital circuits*. McGraw-Hill series in electrical engineering, 1990.

[GSTH08] Donald Gross, John F. Shortle, James M. Thompson, and Carl M. Harris. *Fundamentals of Queueing Theory*. Wiley-Interscience, New York, NY, USA, 4th edition, 2008.

[GvdHG+13] Mattijs Ghijsen, Jeroen van der Ham, Paola Grosso, Cosmin Dumitru, Hao Zhu, Zhiming Zhao, and Cees de Laat. A semantic-web approach for modeling computing infrastructures. *Computers and Electrical Engineering*, 39(8):2553 – 2565, 2013. URL: http://dx.doi.org/10.1016/j.compeleceng.2013.08.011.

[GWVP+13] Ginés D. Guerrero, Richard M. Wallace, José L. Vázquez-Poletti, José M. Cecilia, José M. García, Daniel Mozos, and Horacio Pérez-Sánchez. A performance/cost model for a cuda drug discovery application on physical

and public cloud infrastructures. *Concurrency and Computation: Practice and Experience*, 2013. URL: http://dx.doi.org/10.1002/cpe.3117.

[Hat13] Red Hat. Openshift origin cartridge developer's guide, 2013. URL: http://openshift.github.io/documentation/oo_cartridge_developers_guide.html.

[Hem09] Nicole Hemsoth. Déjà vu all over again – lessons from the past offer insight into hpc advancements of today; a conversation with steve wallach of convey computer. *HPC Wire*, 2009. URL: http://archive.hpcwire.com/hpcwire/2009-11-16/d_j_vu_all_over_again.html.

[Hew08] Carl Hewitt. Orgs for scalable, robust, privacy-friendly client cloud computing. *IEEE Internet Computing*, 12(5):96–99, 2008. URL: http://doi.ieeecomputersociety.org/10.1109/MIC.2008.107.

[Hil09] David Hilley. Cloud computing: A taxonomy of platform and infrastructure-level offerings. Technical report, Georgia Institute of Technology, 2009. URL: http://hdl.handle.net/1853/34402.

[HK11] C.N. Höfer and G. Karagiannis. Cloud computing services: taxonomy and comparison. *Journal of Internet Services and Applications*, 2(2):81–94, 2011. URL: http://dx.doi.org/10.1007/s13174-011-0027-x.

[HKKB13] Jan Heisswolf, Ralf König, Martin Kupper, and Jürgen Becker. Providing multiple hard latency and throughput guarantees for packet switching networks on chip. *Computers & Electrical Engineering*, 39(8):2603 – 2622, 2013. URL: http://dx.doi.org/10.1016/j.compeleceng.2013.06.005.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1985.

[Hol59a] C.S. Holling. The components of predation as revealed by a study of small-mammal predation of the european pine sawfly. *The Canadian Entomologist*, 91(5):293–320, 1959. URL: http://dx.doi.org/10.4039/Ent91293-5.

[Hol59b] C.S. Holling. Some characteristics of simple types of predation and parasitism. *The Canadian Entomologist*, 91(7):385–398, 1959. URL: http://dx.doi.org/10.4039/Ent91385-7.

[HP11] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 5th edition, 2011.

[IBY+07] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys '07: Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 59–72. ACM, 2007. URL: http://doi.acm.org/10.1145/1272996.1273005.

[IEE88] IEEE. Ieee standard vhdl language reference manual. *IEEE Std 1076-1987*, 1988. URL: http://dx.doi.org/10.1109/IEEESTD.1988.122645.

[IEE09] IEEE. Ieee standard vhdl language reference manual. *IEEE Std 1076-2008 (Revision of IEEE Std 1076-2002)*, 2009. URL: http://dx.doi.org/10.1109/IEEESTD.2009.4772740.

[IEE12] IEEE. Iec 62531:2012(e) (ieee std 1850-2010): Standard for property specification language (psl). *IEC 62531:2012(E) (IEEE Std 1850-2010)*, 2012. URL: http://dx.doi.org/10.1109/IEEESTD.2012.6228486.

[Int04] Intel. Ia-pc hpet (high precision event timers) specification. Technical report, Intel, 10 2004. URL: http://www.intel.com/content/dam/www/public/us/en/documents/technical-specifications/software-developers-hpet-spec-1-0a.pdf.

[Int14] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual*, 2014. URL: http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf.

[Jew67] William S. Jewell. A simple proof of: $L = \lambda W$. *Operations Research*, 15(6):1109–1116, 1967. URL: http://pubsonline.informs.org/doi/abs/10.1287/opre.15.6.1109.

[JH11] Chris Jackson and Simon J. Hollis. A deadlock-free routing algorithm for dynamically reconfigurable networks-on-chip. *Microprocessors and Microsystems*, 35(2):139 – 151, 2011. Special issue on Network-on-Chip Architectures and Design Methodologies. URL: http://dx.doi.org/10.1016/j.micpro.2010.09.004.

[JM03] Neil Johnson and Alan Mycroft. Combined code motion and register allocation using the value state dependence graph. In Görel Hedin,

editor, *Compiler Construction*, volume 2622 of *Lecture Notes in Computer Science*, pages 1–16. Springer Berlin Heidelberg, 2003. URL: http://dx.doi.org/10.1007/3-540-36579-6_1.

[Joh14] Benjamin C. Johnstone. Bandwidth requirements of gpu architectures. Masters thesis, Rochester Institute of Technology, December 2014. URL: http://scholarworks.rit.edu/theses/8296/.

[JPB14] A. Jain, R. Parikh, and V. Bertacco. High-radix on-chip networks with low-radix routers. In *Computer-Aided Design (ICCAD), 2014 IEEE/ACM International Conference on*, pages 289–294, Nov 2014. URL: http://dx.doi.org/10.1109/ICCAD.2014.7001365.

[JTB11] Bahman Javadi, Ruppa K. Thulasiram, and Rajkumar Buyya. Statistical modeling of spot instance prices in public cloud environments. In *UCC*, pages 219–228, 2011. URL: http://doi.ieeecomputersociety.org/10.1109/UCC.2011.37.

[KBV00] Thilo Kielmann, Henri E. Bal, and Kees Verstoep. Fast measurement of logp parameters for message passing platforms. In *Proceedings of the 15 IPDPS 2000 Workshops on Parallel and Distributed Processing*, IPDPS '00, pages 1176–1183, 2000. URL: http://dl.acm.org/citation.cfm?id=645612.662667.

[KF14] Dimitris Kontoudis and Panayotis Fouliras. A survey of models for computer networks management. *IJCNC: International journal of Computer Networks & Communications*, 6(3), May 2014. URL: http://dx.doi.org/10.5121/ijcnc.2014.6313.

[KK08] Sukhun Kang and R. Kumar. Magellan: A search and machine learning-based framework for fast multi-core design space exploration and optimization. In *Design, Automation and Test in Europe, 2008. DATE '08*, pages 1432–1437, March 2008. URL: http://dx.doi.org/10.1109/DATE.2008.4484875.

[KKM+14] Hanjoon Kim, Gwangsun Kim, Seungryoul Maeng, Hwasoo Yeo, and John Kim. Transportation-network-inspired network-on-chip. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, pages 332–343, Feb 2014. URL: http://dx.doi.org/10.1109/HPCA.2014.6835943.

[KM08] Fred Kröger and Stephan Merz. *Temporal Logic and State Systems. Texts in Theoretical Computer Science. An EATCS Series*. Springer Publishing Company, Incorporated, 1st edition, 2008.

[KMS14] Jan Komenda, Tomáš Masopust, and JanH. Schuppen. Coordination control of discrete-event systems revisited. *Discrete Event Dynamic Systems*, pages 1–30, 2014. URL: http://dx.doi.org/10.1007/s10626-013-0179-x.

[KNP11] M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)*, volume 6806 of *LNCS*, pages 585–591. Springer, 2011.

[Koo08] Jonathan G Koomey. Worldwide electricity used in data centers. *Environmental Research Letters*, 3(3), 2008.

[Kos12] Piêgas Guilherme Koslovski. *Dynamically provisioned virtual infrastructures : specification, allocation and execution*. Phd thesis, Ecole Normale Supérieure de Lyon, ENS LYON (NNT:2011ENSL0631, tel-00661619), 2012. URL: https://tel.archives-ouvertes.fr/tel-00661619.

[KPCa09] Guilherme Piegas Koslovski, PascaleVicat-Blanc Primet, and Andrea Schwertner Charão. VXDL: Virtual resources and interconnection networks Description Language. In *Networks for Grid Applications*, volume 2 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 138–154. Springer Berlin Heidelberg, 2009. URL: http://dx.doi.org/10.1007/978-3-642-02080-3_15.

[Kri63] Saul A. Kripke. Semantical analysis of Modal Logic I normal modal propositional calculi. *Mathematical Logic Quarterly*, 9(5-6):67–96, 1963. URL: http://dx.doi.org/10.1002/malq.19630090502.

[KRM08] Taeho Kgil, David Roberts, and Trevor Mudge. Improving nand flash based disk caches. In *Computer Architecture, 2008. ISCA'08. 35th International Symposium on*, pages 327–338. IEEE, 2008.

[KS11] Jaeyong Kang and Kwang-Mong Sim. Ontology and search engine for cloud computing system. In *2011 International Conference on System Science and Engineering (ICSSE)*, pages 276–281, June 2011. URL: http://dx.doi.org/10.1109/APSCC.2010.44.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978. URL: http://dl.acm.org/citation.cfm?doid=359545.359563.

[Lam83] Leslie Lamport. What good is temporal logic? In R.E.A. Mason, editor, *Information Processing 83, Proceedings of the IFIP 9th World Computer Congress*, pages 657–668. Elsevier Science Publishers B.V. (North-Holland), 1983.

[Lea09] Neal Leavitt. Complex-event processing poised for growth. *Computer*, 42(4):17–20, 2009. URL: http://doi.ieeecomputersociety.org/10.1109/MC.2009.109.

[Lei85] Charles E. Leiserson. Fat-trees: Universal networks for hardware-efficient supercomputing. *IEEE Transactions on Computers*, C-34(10):892–901, Oct 1985. URL: http://dx.doi.org/10.1109/TC.1985.6312192.

[LFSZ12a] Yong Li, Dan Feng, Zhan Shi, and Zhao Zhang. Disk array performance prediction with cart-mars hybrid models. In *Asia-Pacific Magnetic Recording Conference, 2012 Digest*, pages 1–4. IEEE, Oct 2012.

[LFSZ12b] Yong Li, Dan Feng, Zhan Shi, and Zhao Zhang. Disk array performance prediction with cart-mars hybrid models. In *APMRC, 2012 Digest*, pages 1–4, Oct 2012.

[LG08] John D.C. Little and Stephen C. Graves. Little's law. In *Building Intuition*, volume 115 of *International Series in Operations Research & Management Science*, pages 81–100. Springer US, 2008. URL: http://dx.doi.org/10.1007/978-0-387-73699-0_5.

[LGH10] Youming Li, Ardian Greca, and James Harris. On dijkstra's algorithm for deadlock detection. In Khaled Elleithy, editor, *Advanced Techniques in Computing Sciences and Software Engineering*, pages 385–387. Springer Netherlands, 2010. URL: http://dx.doi.org/10.1007/978-90-481-3660-5_66.

[LKV14] D.J. Lingenfelter, A. Khurshudov, and D.M. Vlassarev. Efficient disk drive performance model for realistic workloads. *Magnetics, IEEE Transactions on*, 50(5):1–9, May 2014. URL: http://dx.doi.org/10.1109/TMAG.2013.2294942.

[Lot10] Alfred J. Lotka. Contribution to the theory of periodic reactions. *Journal of Physical Chemistry*, 14(3):271—-274, 1910. URL: http://pubs.acs.org/doi/pdf/10.1021/j150111a004.

[LS11] Muthusamy Lakshmanan and Dharmapuri Vijayan Senthilkumar. *Dynamics of Nonlinear Time-Delay Systems*. Springer Series in Synergetics. Springer, 2011.

[LSZ09] S. Leo, F. Santoni, and G. Zanetti. Biodoop: Bioinformatics on hadoop. In *Parallel Processing Workshops, 2009. ICPPW '09; International Conference on*, pages 415–422, 2009. URL: http://dx.doi.org/10.1109/ICPPW.2009.37.

[LTM+11] Fang Liu, Jin Tong, Jian Mao, Robert Bohn, John Messina, Lee Badger, and Dawn Leaf. Nist cloud computing reference architecture. Special Publication 500-292, NIST, 2011. URL: http://www.nist.gov/publication-portal.cfm.

[LTW14] Glenn K. Lockwood, Mahidhar Tatineni, and Rick Wagner. Sr-iov: Performance benefits for virtualized interconnects. In *Proceedings of the 2014 Annual Conference on Extreme Science and Engineering Discovery Environment*, XSEDE '14, 2014. URL: http://doi.acm.org/10.1145/2616498.2616537.

[Luc02] David C Luckham. *The power of events*. Addison-Wesley, Reading, MA, 2002.

[LXW+11] Weichen Liu, Jiang Xu, Xiaowen Wu, Yaoyao Ye, Xuan Wang, Wei Zhang, M. Nikdast, and Zhehui Wang. A noc traffic suite based on real applications. In *VLSI (ISVLSI), 2011 IEEE Computer Society Annual Symposium on*, pages 66–71, July 2011. URL: http://dx.doi.org/10.1109/ISVLSI.2011.49.

[LZFD10] Liang Li, Xingjun Zhang, Jinghua Feng, and Xiaoshe Dong. mplogp: A parallel computation model for heterogeneous multi-core computer. In *IEEE/ACM International 10th Conference on Cluster, Cloud and Grid Computing (CCGrid), 2010*, pages 679–684, May 2010. URL: http://dx.doi.org/10.1109/CCGRID.2010.60.

[LZG+14] Jun Li, MengChu Zhou, Tao Guo, Yahui Gan, and Xianzhong Dai. Robust control reconfiguration of resource allocation systems with petri nets and integer programming. *Automatica*, 50(3):915 – 923, 2014. URL: http://dx.doi.org/10.1016/j.automatica.2013.12.015.

[May73] Robert M May. Stability in randomly fluctuating versus deterministic environments. *American Naturalist*, pages 621–650, 1973.

[MBA+14] A.W. Malik, K. Bilal, K. Aziz, D. Kliazovich, N. Ghani, S.U. Khan, and R. Buyya. Cloudnetsim++: A toolkit for data center simulations in omnet++. In *High-capacity Optical Networks and Emerging/Enabling Technologies (HONET), 2014 11th Annual*, pages 104–108, Dec 2014. URL: http://dx.doi.org/10.1109/HONET.2014.7029371.

[Met12] Cade Metz. The secret history of openstack, the free cloud software that's changing everything, 2012. URL: http://www.wired.com/wiredenterprise/2012/04/openstack/all/.

[MFP06] Gero Mühl, Ludger Fiege, and Peter Pietzuch. *Distributed Event-Based Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[MJL11] Yong Beom Ma, Sung Ho Jang, and Jong Sik Lee. Ontology-based resource management for cloud computing. In NgocThanh Nguyen, Chong-Gun Kim, and Adam Janiak, editors, *Intelligent Information and Database Systems*, volume 6592 of *Lecture Notes in Computer Science*, pages 343–352. Springer Berlin Heidelberg, 2011. URL: http://dx.doi.org/10.1007/978-3-642-20042-7_35.

[MN98] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul.*, 8(1):3–30, January 1998. URL: http://doi.acm.org/10.1145/272991.272995.

[NdSPC13] N. Nikitin, J. de San Pedro, and J. Cortadella. Architectural exploration of large-scale hierarchical chip multiprocessors. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 32(10):1569–1582, Oct 2013. URL: http://dx.doi.org/10.1109/TCAD.2013.2272539.

[NnVPC+12] Alberto Núñez, Jose Luis Vázquez-Poletti, AgustinC. Caminero, Gabriel G. Castañé, Jesus Carretero, and Ignacio M. Llorente. icancloud: A flexible and scalable cloud infrastructure simulator. *Journal of Grid Computing*, 10(1):185–209, 2012. URL: http://dx.doi.org/10.1007/s10723-012-9208-5.

[NS07] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 89–100, 2007. URL: http://doi.acm.org/10.1145/1250734.1250746.

[NVI14] NVIDIA. *CUDA C PROGRAMMING GUIDE*, 2014. URL: http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf.

[OM13] Umit Y. Ogras and Radu Marculescu. Literature survey. In *Modeling, Analysis and Optimization of Network-on-Chip Communication Architectures*, volume 184 of *Lecture Notes in Electrical Engineering*, pages 9–32. Springer Netherlands, 2013. URL: http://dx.doi.org/10.1007/978-94-007-3958-1_2.

[PKL08] Kevin Pedretti, Suzanne Kelly, and Michael Levenhagen. Summary of multi-core hardware and programming model investigations. Technical Report SAND2008-3205, Sandia National Laboratories, May 2008. URL: http://www.sandia.gov/~ktpedre/papers/multicore-tech.pdf.

[PST+97] Karin Petersen, Mike J. Spreitzer, Douglas B. Terry, Marvin M. Theimer, and Alan J. Demers. Flexible update propagation for weakly consistent replication. *SIGOPS Oper. Syst. Rev.*, 31(5):288–301, October 1997. URL: http://doi.acm.org/10.1145/269005.266711.

[QJB+14] Zhiliang Qian, Da-Cheng Juan, P. Bogdan, Chi-Ying Tsui, D. Marculescu, and R. Marculescu. A comprehensive and accurate latency model for network-on-chip performance analysis. In *Design Automation Conference (ASP-DAC), 2014 19th Asia and South Pacific*, pages 323–328, Jan 2014. URL: http://dx.doi.org/10.1109/ASPDAC.2014.6742910.

[Reg10] Giles Reger. Rule-based runtime verification in a multicore system setting. Master's thesis, Engineering and Physical Sciences, 2010. URL: http://studentnet.cs.manchester.ac.uk/resources/library/thesis_abstracts/MSc10/FullText/RegerGiles.pdf.

[RHvdAM06] N. Russell, A.H.M. ter Hofstede, W.M.P. van der Aalst, and N. Mulyar. Workflow control-flow patterns: A revised view. Technical report, Business Process Management (BPM) Center, 2006. URL: http://www.workflowpatterns.com/documentation/documents/BPM-06-22.pdf.

[RKL+05] Dumitru Roman, Uwe Keller, Holger Lausen, Jos de Bruijn, Rubén Lara, Michael Stollberg, Axel Polleres, Cristina Feier, Cristoph Bussler, and Dieter Fensel. Web service modeling ontology. *Applied Ontology*, 1(1):77–106, January 2005. URL: http://dl.acm.org/citation.cfm?id=1412350.1412357.

[RL97] R. R. Lawrence. Using neural networks to forecast stock market prices. *University of Manitoba*, 1997.

[RMB+09] Dara Rahmati, Srinivasan Murali, Luca Benini, Federico Angiolini, Giovanni De Micheli, and Hamid Sarbazi-Azad. A method for calculating hard qos guarantees for networks-on-chip. In *Proceedings of the 2009 International Conference on Computer-Aided Design*, ICCAD '09, pages 579–586, 2009. URL: http://doi.acm.org/10.1145/1687399.1687507.

[RS96] M. Raynal and M. Singhal. Logical time: capturing causality in distributed systems. *Computer*, 29(2):49–56, 1996. URL: http://dx.doi.org/10.1109/2.485846.

[RSS07] Szabolcs Rozsnyai, Josef Schiefer, and Alexander Schatten. Concepts and models for typing events for event-based systems. In *Proceedings of the 2007 inaugural international conference on Distributed event-based systems*, DEBS '07, pages 62–70, New York, NY, USA, 2007. ACM. URL: http://doi.acm.org/10.1145/1266894.1266904.

[RW94] Chris Ruemmler and John Wilkes. An introduction to disk drive modeling. *Computer*, 27(3):17–28, 1994.

[RYC+13] Christopher J. Rossbach, Yuan Yu, Jon Currey, Jean-Philippe Martin, and Dennis Fetterly. Dandelion: A compiler and runtime for heterogeneous systems. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 49–68, New York, NY, USA, 2013. ACM. URL: http://doi.acm.org/10.1145/2517349.2522715.

[SDQR10] Jörg Schad, Jens Dittrich, and Jorge-Arnulfo Quiané-Ruiz. Runtime measurements in the cloud: Observing, analyzing, and reducing variance. *Proceedings of the VLDB Endowment*, 3(1-2):460–471, 2010. URL: http://dx.doi.org/10.14778/1920841.1920902.

[SDS14] Erich Strohmaier, Jack Dongarra, and Horst Simon. The top500 supercomputer site, 2014. URL visited 2014-04-12. URL: http://www.top500.org/.

[SF09] Edi Shmueli and Dror G. Feitelson. On simulation and design of parallel-systems schedulers: Are we doing the right thing? *IEEE Trans. Parallel Distrib. Syst.*, 20(7):983–996, July 2009. URL: http://dx.doi.org/10.1109/TPDS.2008.152.

[SGWVP15] Mehdi Sheikhalishahi, Lucio Grandinetti, Richard M. Wallace, and José Luis Vázquez-Poletti. Autonomic resource contention-aware scheduling. *Software: Practice and Experience*, 45(2):161–175, 2015. URL: http://dx.doi.org/10.1002/spe.2223.

[SI09] Konstantin Serebryany and Timur Iskhodzhanov. Threadsanitizer: Data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications*, WBIA '09, pages 62–71, 2009. URL: http://doi.acm.org/10.1145/1791194.1791203.

[Sim12] Kwang Mong Sim. Agent-based cloud computing. *Services Computing, IEEE Transactions on*, 5(4):564–577, 2012. URL: http://dx.doi.org/10.1109/TSC.2011.52.

[Sin89] Mukesh Singhal. Deadlock detection in distributed systems. *Computer*, 22(11):37–48, Nov 1989. URL: http://dx.doi.org/10.1109/2.43525.

[SLG11] Mehdi Sheikhalishahi, Ignacio M. Llorente, and Lucio Grandinetti. Energy aware consolidation policies. In *International Conference on Parallel Computing*, pages 109–116. IOS Press, 30 August-2 September 2011.

[SLPSCG11] I. Sánchez-Linares, H. Pérez-Sánchez, J.M. Cecilia, and J.M. García. Bindsurf: A fast blind virtual screening methodology on gpus. In *Network Tools and Applications in Biology (NETTAB 2011), Clinical Bioinformatics*, pages 95–97, 2011.

[SMJ02] Peter Spyns, Robert Meersman, and Mustafa Jarrar. Data modelling versus ontology engineering. *SIGMOD Rec.*, 31(4):12–17, December 2002. URL: http://doi.acm.org/10.1145/637411.637413.

[SMS14] Jay Smith, Anthony A. Maciejewski, and Howard Jay Siegel. Maximizing stochastic robustness of static resource allocations in a periodic sensor driven cluster. *Future Generation Computer Systems*, 33(0):1 – 10, 2014. URL: http://dx.doi.org/10.1016/j.future.2013.10.001.

[Sos11] Barrie Sosinsky. *Cloud Computing Bible*. Wiley Publishing, 1st edition, 2011.

[Spi92] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 2nd edition, 1992.

[SRHZ14] WaselulHaque Sadid, Laurie Ricker, and Shahin Hashtrudi-Zad. Robustness of synchronous communication protocols with delay for decentralized

discrete-event control. *Discrete Event Dynamic Systems*, pages 1–18, 2014. URL: http://dx.doi.org/10.1007/s10626-014-0184-8.

[SRM+13] Ciprian Seiculescu, Dara Rahmati, Srinivasan Murali, Hamid Sarbazi-Azad, Luca Benini, and Giovanni De Micheli. Designing best effort networks-on-chip to meet hard latency constraints. *ACM Trans. Embed. Comput. Syst.*, 12(4):108:1–108:23, July 2013. URL: http://doi.acm.org/10.1145/2485984.2485996.

[SSMS08] Vladimir Shestak, Jay Smith, Anthony A. Maciejewski, and Howard Jay Siegel. Stochastic robustness metric and its use for static resource allocations. *Journal of Parallel and Distributed Computing*, 68(8):1157 – 1173, 2008. URL: http://dx.doi.org/10.1016/j.jpdc.2008.01.002.

[SVL07] David Sheldon, Frank Vahid, and Stefano Lonardi. Interactive presentation: Soft-core processor customization using the design of experiments paradigm. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '07, pages 821–826, San Jose, CA, USA, 2007. EDA Consortium. URL: http://dl.acm.org/citation.cfm?id=1266366.1266541.

[SWG+14] Mehdi Sheikhalishahi, Richard M. Wallace, Lucio Grandinetti, José Luis Vázquez-Poletti, and Francesca Guerriero. A multi-capacity queuing mechanism in multi-dimensional resource scheduling. In Florin Pop and Maria Potop-Butucaru, editors, *Adaptive Resource Management and Scheduling for Cloud Computing*, Lecture Notes in Computer Science, pages 9–25. Springer International Publishing, 2014. URL: http://dx.doi.org/10.1007/978-3-319-13464-2_2.

[SWG+15] Mehdi Sheikhalishahi, Richard M. Wallace, Lucio Grandinetti, José Luis Vázquez-Poletti, and Francesca Guerriero. A multi-dimensional job scheduling. *Future Generation Computer Systems*, 2015. URL: http://www.sciencedirect.com/science/article/pii/S0167739X1500076X.

[SZS+03] Stan Zdonik Sbz, Stan Zdonik, Michael Stonebraker, Mitch Cherniack, Ugur C Etintemel, Magdalena Balazinska, and Hari Balakrishnan. The aurora and medusa projects. *IEEE Data Engineering Bulletin*, 26, 2003.

[TBDSG11] V. Turchenko, P. Beraldi, F. De Simone, and L. Grandinetti. Short-term stock price prediction using mlp in moving simulation mode. In *Intelligent Data Acquisition and Advanced Computing Systems (IDAACS), 2011 IEEE 6th International Conference on*, volume 2, pages 666–671, 2011. URL: http://dx.doi.org/10.1109/IDAACS.2011.6072853.

[TD86] Satish K. Tripathi and Andrzej Duda. Time-dependent analysis of queueing systems. *Information Systems Operations Research (INFOR)*, 24(3):199–219, 1986.

[TLL+11] Xiaoyong Tang, Kenli Li, Guiping Liao, Kui Fang, and Fan Wu. A stochastic scheduling algorithm for precedence constrained tasks on grid. *Future Generation Computer Systems*, 27(8):1083 – 1091, 2011. URL: http://dx.doi.org/10.1016/j.future.2011.04.007.

[TST+14] Volodymyr Turchenko, Vladyslav Shults, Iryna Turchenko, Richard Wallace, Sheikhalishahi Mehdi, José Vázquez-Poletti, and Lucio Grandinetti. Spot price prediction for cloud computing using neural networks. *International Journal of Computing*, 12(4):348–359, 2014. URL: http://computingonline.net/index.php/computing/article/view/615.

[UAM01] Mustafa Uysal, Guillermo A Alvarez, and Arif Merchant. A modular, analytical throughput model for modern disk arrays. In *Modeling, Analysis and Simulation of Computer and Telecommunication Systems, 2001. Proceedings. Ninth International Symposium on*, pages 183–192. IEEE, 2001.

[Vac05] John R. Vacca. *Optical Networking Best Practices Handbook*. John Wiley & Sons, 2005.

[vdAtHKB03] W. M. P. van der Aalst, A. H. M. ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow patterns. *Distrib. Parallel Databases*, 14(1):5–51, July 2003. URL: http://dx.doi.org/10.1023/A:1022883727209.

[vdHeDŁZ13] Jeroen van der Ham (editor), Freek Dijkstra, Roman Łapacz, and Jason Zurawski. Network markup language base schema version 1 (gfd.206), Sep 2013. Open Grid Forum. URL: http://www.ogf.org:8080/gf/docs/?final.

[VDY05] Richard Vuduc, James W Demmel, and Katherine A Yelick. Oski: A library of automatically tuned sparse matrix kernels. *Journal of Physics: Conference Series*, 16(1), 2005. URL: http://dx.doi.org/10.1088/1742-6596/16/1/071.

[Vog09] Werner Vogels. Eventually consistent. *Communications of the ACM*, 52(1):40–44, January 2009. URL: http://doi.acm.org/10.1145/1435417.1435432.

[Vol26] V. Volterra. Variazioni e fluttuazioni del numero d'individui in specie animali conviventi. *Accademia Nazionale dei Lincei, Rome*, 2:31–113, 1926.

[Wal00] Richard M. Wallace. Regulated isomorphic temporal architecture, 2000. URL: http://emendo-ex-erratum.org/papers/RITA.pdf.

[WCB10] B. Wickremasinghe, R.N. Calheiros, and R. Buyya. Cloudanalyst: A cloudsim-based visual modeller for analysing cloud computing environments and applications. In *24th IEEE International Conference on Advanced Information Networking and Applications (AINA)*, pages 446–452, April 2010. URL: http://dx.doi.org/10.1109/AINA.2010.32.

[WCES94] Daniel Weise, Roger F. Crew, Michael Ernst, and Bjarne Steensgaard. Value dependence graphs: Representation without taxation. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '94, pages 297–310, 1994. URL: http://doi.acm.org/10.1145/174675.177907.

[WD98] R. Clint Whaley and Jack J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, pages 1–27. IEEE Computer Society, 1998. URL: http://dx.doi.org/10.1109/SC.1998.10004.

[WG84] William Waite and Gerhard Goos. *Compiler construction*. Texts and monographs in computer science. Springer, 1984.

[WJM08] Wayne Wolf, Ahmed Amine Jerraya, and Grant Martin. Multiprocessor system-on-chip (mpsoc) technology. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(10):1701–1713, Oct 2008. URL: http://dx.doi.org/10.1109/TCAD.2008.923415.

[WLLD05] Chuliang Weng, Minglu Li, Xinda Lu, and Qianni Deng. An economic-based resource management framework in the grid context. In *Cluster Computing and the Grid, 2005. CCGrid 2005. IEEE International Symposium on*, volume 1, pages 542–549 Vol. 1, 2005. URL: http://dx.doi.org/10.1109/CCGRID.2005.1558601.

[WMH84] Richard M. Wallace, James E. McDonald, and Duane O. Hague. A data-driven operating system for data-driven architectures of real-time systems, 1984. URL: http://emendo-ex-erratum.org/papers/RMW_DH_JM_DDOS-DDA.zip.

[WMV14] Richard Wallace, Patrick Martin, and José Luis Vázquez-Poletti. Regulated Condition-Event Matrices for Cloud Environments. *Scalable Computing: Practice and Experience*, 15(2):169–185, 2014. URL: http://www.scpe.org.

[WTS+13] R.M. Wallace, V. Turchenko, M. Sheikhalishahi, I. Turchenko, V. Shults, J. L. Vázquez-Poletti, and L. Grandinetti. Applications of neural-based spot market prediction for cloud computing. In *Intelligent Data Acquisition and Advanced Computing Systems (IDAACS), 2013 IEEE 7th International Conference on*, volume 02, pages 710–716, Sept 2013. URL: http://dx.doi.org/10.1109/IDAACS.2013.6663017.

[WVC+11] Richard Wallace, Bogdan Vacaliuc, Dwight Clayton, Bruce Chaffins, Olaf Storaasli, Dave Strenski, and Dan Poznanovic. Consideration of the tms320c6678 multi-core dsp for power efficient high performance computing. Technical Report ORNL-Pub-28647, Oak Ridge National Laboratory, Feb 2011. Original URL was http://info.ornl.gov/sites/publications/Files/Pub28647.pdf. URL: https://www.dropbox.com/s/56t9z9lav5ir45f/ORNL-Pub28647.pdf?dl=0.

[XHTH13] Yi Xie, J. Hu, S. Tang, and X. Huang. A forward—backward algorithm for nested hidden semi-markov model and application to network traffic. *The Computer Journal*, 56(2):229–238, February 2013. URL: http://dx.doi.org/10.1093/comjnl/bxs124.

[XHX+13] Yi Xie, Jiankun Hu, Yang Xiang, Shui Yu, Shensheng Tang, and Yu Wang. Modeling oscillation behavior of network traffic by nested hidden markov model with variable state-duration. *IEEE Transactions on Parallel and Distributed Systems*, 24(9):1807–1817, Sept 2013. URL: http://dx.doi.org/10.1109/TPDS.2012.268.

[XJJP01] Jianxin Xiong, Jeremy Johnson, Robert Johnson, and David Padua. Spl: A language and compiler for dsp algorithms. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01, pages 298–308. ACM, 2001. URL: http://dx.doi.org/10.1145/378795.378860.

[YBDS08] L. Youseff, M. Butrico, and D. Da Silva. Toward a unified ontology of cloud computing. In *Grid Computing Environments Workshop, 2008. GCE '08*, pages 1–10, Nov 2008. URL: http://dx.doi.org/10.1109/GCE.2008.4738443.

[YXMZ12] Yi Yang, Ping Xiang, M. Mantor, and Huiyang Zhou. Cpu-assisted gpgpu on fused cpu-gpu architectures. In *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on*, pages 1–12, Feb 2012. URL: http://doi.ieeecomputersociety.org/10.1109/HPCA.2012.6168948.

[ZBF10] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 129–142, New York, NY, USA, 2010. ACM. URL: http://doi.acm.org/10.1145/1736020.1736036.

[ZKK12] Eitan Zahavi, Isaac Keslassy, and Avinoam Kolodny. Distributed adaptive routing for big-data applications running on data center networks. In *Proceedings of the Eighth ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '12, pages 99–110, 2012. URL: http://doi.acm.org/10.1145/2396556.2396578.