UNIVERSITY OF CALIFORNIA
RIVERSIDE

Accelerating Synchronous Many-Core Networks on FPGAs

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Bailey Alan Miller

June 2014

Dissertation Committee:

Dr. Frank Vahid, Chairperson
Dr. Tony Givargis
Dr. Philip Brisk
Dr. Sheldon Tan

The Dissertation of Bailey Alan Miller is approved:

_____

_____

_____

_____
Committee Chairperson

University of California, Riverside

ACKNOWLEDGMENTS

I'd like to first thank my advisor, Dr. Frank Vahid, for his incredible mentorship over the past 5 years. I was just a quiet kid in the back of an undergraduate course, and with your guidance I somehow stumbled into a PhD. Thank you for sharing your wisdom, for your friendship, and for being gentle when reviewing those early first paper drafts.

To my parents, Alan and Mary Jane. For some reason, probably because it kept me quiet, you let me play on my computer for hours on end as a child. Because of this philosophy, I was able to find my passion for computers and engineering that otherwise may never have been. You encouraged me to grow, and inspired merit through hard work in every pursuit. This thesis and the degree that it represents would not have been possible without your love and support. You are paragons for everyone around you, and I love you both.

To my family, including especially my siblings Colin, Andrew, Scott, and Janelle, and their families. Thank you for being supportive, despite perhaps wondering why school should take a person more than years. I look forward to spending more time with you all.

Finally, to my wife Tricia, whom cares little about acknowledgements but is deserving of them anyways. For 5 years you have been a constant pillar of support, always encouraging when times were hard. I feel like the past 5 years have been a constant battle for us. We've experienced awesome and inspiring events like having a

beautiful wedding. We've toured some of the world and had so many great experiences. We've also each struggled with finding our own direction and purpose. Luckily, we have one another for guidance through those hard times and the inevitable graduate-student periods of self-doubt. You are always there to pick me up when I struggle, and I hope to be there as much for you also. Without you, this journey would have been impossible. Thank you and I love you.

Some of the work contained within this dissertation has been previously published by the Association of Computer Machinery (ACM) and the Institute of Electrical and Electronics Engineers (IEEE).

DEDICATION

Dedicated to my wife Tricia.

ABSTRACT OF THE DISSERTATION

Accelerating Synchronous Many-Core Networks on FPGAs

by

Bailey Alan Miller

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, June 2014
Dr. Frank Vahid, Chairperson

Applications running on custom architectures with hundreds of specialized processing elements (PEs) on field-programmable gate arrays (FPGAs) can gain 10x or greater speedups versus desktop or embedded processors. Applications benefiting from such many-core PE networks include applications with non-centralized communication and memory requirements, like real-time emulation of physical systems, video streaming applications, signal processing, and more. Synchronous many-core networks (syncMCs) consist of synchronized many-core PE networks that execute distributed computations in parallel, keeping each PE in the network in lockstep through a regular, periodic communication phase.

An FPGA has limited logic and routing resources on which to place a synchMC network. Large or densely connected synchMCs can overwhelm the available resources and severely impact the resulting FPGA circuit frequency, and thus application performance. This dissertation focuses primarily on methods for successfully implementing large synchMCs, consisting of hundreds of PEs and thousands of interconnects, on an FPGA. An approach is described that considers the natural structure of a physical model, such as a mesh, ring, or cube, to perform a graph embedding of a synchMC on a 2-dimensional grid of physical PE regions. The described PE placement techniques reduce critical path length and allows previously unroutable designs able to complete place-and-route. In addition, an automated approach to reduce the number of wires in a synchMC allows applications of arbitrary size and complexity to fit within the resource constraints of a target FPGA. Time-multiplexed communication for synchMCs is introduced, and a greedy scheduler, a heuristic scheduler, and an integer linear program scheduler are described. Finally, an approach for exploring the configuration space of a synchMC executing real-time emulation of a physical model is described. Using the described synthesis approaches, synchMCs enable the fastest emulation of physical models on a moderately priced platform, executing 15x faster than a desktop PC, 26x faster than a GPU, 9x faster than a network-on-chip (NoC), and 9x faster than a circuit produced via high-level synthesis (HLS).

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1. SYNCHRONOUS MANY-CORE (SYNCHMC) NETWORKS

1.1 INTRODUCTION

Many-core architectures, consisting of hundreds to thousands of small processing elements (PEs), are powerful platforms for executing applications with extensive parallelism. Examples include iterative numerical systems solvers and high-performance streaming applications [1][18]. In general, many-core architectures are often a good fit for applications that can be modeled as Kahn process networks (KPNs) [26]. Previous research has introduced many-core PE networks that are *synchronized*, referred to as synchronous many-core networks (synchMCs), having a compute phase that executes a sequential process within each PE, followed by an update phase that resolves any data dependencies between PEs [22]. SynchMCs are able to emulate complex physical systems 2x-100x faster than other embedded platforms, and faster than a desktop PC (15x), GPU (26x), or high-level synthesized circuits (9x).

SynchMCs are particularly fitted for applications adhering to the synchronous data flow (SDF) computation model, a variant of the KPN model wherein the nodes of a network periodically fire to generate and consume tokens [31]. SDF-based applications can be mapped to synchMCs by considering the nodes of the SDF as the synchMC PEs, the connections between SDF nodes as the point-to-point communication architecture of

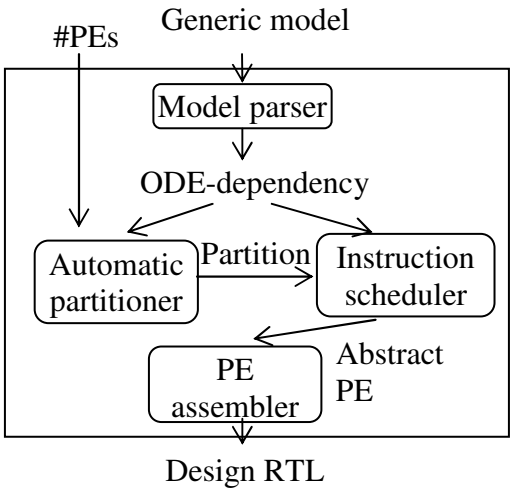

Figure 1-1: PE compiler structure. A model is translated into an ODE dependency graph. Equations are partitioned to PEs, the PEs are scheduled, and RTL is produced.

the synchMC, and the tokens as synchMC messages passed between PEs during each iteration. While this work focuses on the emulation of physical models as the driving example application, synchMCs are applicable to other domains and applications besides numerical solvers, especially those framed within the SDF model.

To accelerate physical model emulation using synchMCs, researchers previously introduced a lightweight programmable general ODE-solver PE for a field-programmable gate array (FPGA), where the PE was optimized for solving tens of ODEs [20]. The work included a new synthesis tool, referred to as the ***PE compiler***, that automatically maps thousands of ODEs onto a statically-scheduled custom network of tens or hundreds of PEs, each PE solving a subset of ODEs [22]. Because physical systems are typically comprised of numerous physical objects communicating locally with neighboring objects,

physical systems represent an excellent match for FPGAs, which excel at performing numerous parallel computations with local communication.

The PE compiler has the structure shown in Figure 1-1. A text description of the equations define the model are supplied in the form of a subset of MML [38]. A parser extracts the equations from the description and determines the ODEs of the model. Equations are then partitioned to a number of PEs using a simulated annealing heuristic that primarily attempts to reduce the number of wires in the design while balancing computation cost across the network. Once the annealing is complete, the computation of each PE is scheduled, the global communication tasks are added to the end of the schedule, and RTL is generated.

The compiler schedules synchronized iterations of the equations, separating computation and communication in to two unique stages. The first stage is an *evaluation* stage, in which each PE computes its assigned variables. The second stage is an *update* stage, in which data dependencies between PEs are resolved. For example, if a PE2 computes x' = y+y, and y is computed in PE3, then PE3 must update PE2 with the value of y in the current iteration. All PEs are synchronized such that their schedule lengths are identical, and the update phase of the network occurs simultaneously for all PEs.

More details on the PE architecture and compiler are available in past research manuscripts [20][21][22].

1.2 RELATED WORK

Previous researchers have introduced many-core networks as an alternative architecture for high-performance computing. Advantages of many-core networks include reduced power usage, and throughput potential that increases linearly with the number of cores [10]. The past decade has seen work towards creating programming frameworks and operating systems focused on many-core networks, such as Corey [9], Barrelfish [50], Tessellation [33], Helios [44], and others. Modern many-core architectures networks are a derivative of past research efforts on *systolic arrays*, which are (typically) homogeneous arrays of small processors called cells that communicate only with neighboring cells [27].

This work focuses on an *application-specific* many-core network, instead of generalized many-core frameworks and/or operating systems. In an application-specific network, the architecture and scheduling has been produced for a specific instance of an application. Furthermore, the architecture is based around small and simple cores with distributed memories local to each core—thus, the architecture has no global shared memory and all sharing between cores must be scheduled as explicit tasks. Perhaps the most similar work to ours is GraphStep [12], an architecture and framework for executing sparse graph-based applications as many-core networks in FPGAs. GraphStep achieves 10x-20x speedups for a knowledge base search application, similar to speedups in our previous work. Other similar work includes AsAP (Asynchronous Array of simple Processors) [56][64], which consists programmable processors connected via a reconfigurable mesh network and implemented as an ASIC. In contrast to synchMCs,

each processor can be individually clocked via a globally asynchronous locally synchronous (GALS) approach. By targeting ASIC platforms AsAP achieves ~1GHz clock frequencies compared to the ~300MHz clock frequencies of synchMCs on FPGAs. Thus AsAP has faster performance and higher throughput, but higher cost and less flexibility in terms of being able to set the number of processors on the chip, or add additional logic or debug capabilities to the platform.

This dissertation focuses on physical model emulation as a driving example, thus we give some background here on that domain. Languages have been introduced for modeling physical systems, such as MML [38], SBML [23], and CellML [34]. Tools have been built that can simulate physical models, such as Matlab [36], and LabView [43]. These tools usually aim to produce accurate results rather than real-time emulation. The tools described in this paper use a subset of MML to capture model equations as high-level textual descriptions, which are then parsed and compiled to many-core networks.

Many case studies using FPGAs to speedup physical model emulation have been conducted. For example, Yoshimi [63] obtained 100x speedups of a fine-grained biological emulation compared to a single-core processor, and showed why multi-core processors were not suitable. de Pimentel [49] proposed an FPGA solution to emulate a heart-lung system model in real-time, while a PC required 1.5 hours to simulate 60 seconds of the same model. Their FPGA performance was estimated by a theoretical optimal formula, rather than via an implementation. Osana developed the ReCSiP [46] tool to generate chemical models on FPGAs using the SBML language. The crossbar

communication structure used in ReCSiP may not scale to larger models. Those previous efforts mostly used manual design approaches to implement the models on FPGAs.

Chapter 2. PLACEMENT OF STRUCTURED APPLICATIONS ON FPGAS USING SYNCHMCS

Fast physical model simulations are required in various domains, including biomedical engineering, physics, chemistry, and much more. A physical model represents some observable physical phenomena, usually as a set of normal, partial differential, or ordinary differential equations. The set of equations can be solved using time-stepping equation solvers.

In the cyber-physical system domain, previous work uses physical models to interact with and test devices such as ventilators [39], pacemakers [24], and unmanned aerial vehicles [17]. Using physical model simulations for testing can be preferable over the actual physical environment when such an environment is difficult, expensive, or dangerous to create or use. Physical models may also be more accurate than physical analogs, e.g., a balloon may capture some of the behavior of a lung, but may not be able to accurately model various lung diseases.

Our previous research has been able to speed up physical model simulation up to three orders of magnitude versus multi-core desktop processors, by partitioning physical model computation across hundreds of processing elements in a synchMC, each PE optimized to execute time-stepping equation solvers [20].

Many physical models share the same natural structure as the corresponding physical system. For example, a Weibel lung model [61] utilizes a binary tree structure because

the lung physiology itself is a tree in which the trachea is the root and where gas exchange occurs at the leaves. Similarly, atrial cell models utilize a three-dimensional mesh structure to simulate the propagation of electrical signals across tissues of cardiac cells [65]. Equations of the physical system are grouped naturally, e.g., the volume and pressure of a lung branch have data dependencies and thus should ideally be placed within the same PE to minimize communication costs. Generally, the natural structure of a physical model provides an optimal grouping of equations that minimizes communication costs.

A key contribution of this work is utilizing the natural structure of simulated physical model during placement of a PE network onto an FPGA. By using graph embedding techniques that have been extensively researched in graph theory literature, the structure of the physical model can be embedded onto a two-dimensional grid of PE elements on

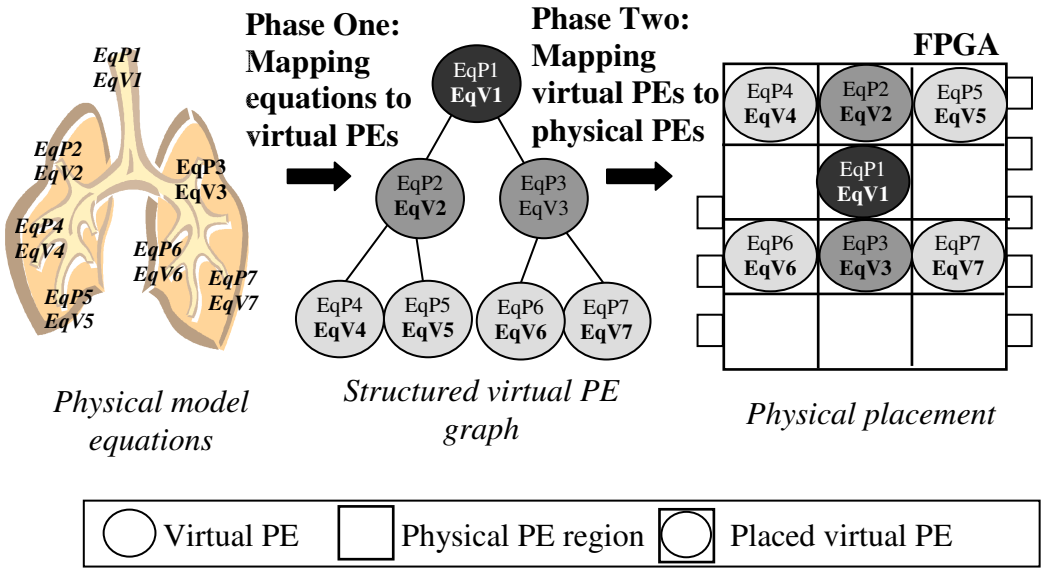

Figure 2-1: Two-phase approach of mapping equations to a structured graph of virtual PEs, and mapping virtual PEs to an FPGA with graph embedding.

an FPGA. By performing graph embeddings, the resulting circuit incurs less communication cost and enables higher circuit frequencies, translating to faster execution of physical models. A secondary contribution is the definition of a simulated annealing approach that provides cost and neighbor functions for minimizing distances between PEs placed on a grid of physical regions on a FPGA, used for unknown model structures and also for evaluating the first contribution.

Figure 2-1 details a two-phase approach for embedding a physical model onto an FPGA. The first phase maps the physical model equations to a structured virtual PE graph. A structured virtual PE graph has virtual PE nodes that contain groups of equations, have connections to other virtual PE nodes, and is structured in the form of the physical model. Physical placement can then be performed by defining physical PE regions where virtual PEs may be mapped, and then either applying the appropriate graph embedding algorithm or using a general simulated annealing approach to perform the mapping. In the right side of Figure 2-1, a graph embedding algorithm maps a binary tree to a two-dimensional grid by placing the root in a physical PE region in middle of the grid, and expanding the child subtrees out in different directions.

2.1 PHYSICAL MODEL STRUCTURES

Physical models often have a natural structure associated with a corresponding layout in the physical world. Consider a human lung, which begins at the trachea and splits into nearly identical left and right lobes. Each lung contains more than twenty additional splits as the airway passage diameters decrease and eventually are able to

support blood-gas exchange alveoli. The lung has thus often been modeled as a binary tree of twenty or more generations such that gas flow at the trachea can be used to compute the pressure and volume of internal branches [61]. Similarly, cell models that simulate electrical activity across heart atrium walls utilize a three-dimensional mesh structure to allow neighboring cells to propagate signals. Figure 2-2 shows some examples of physical models and their corresponding structures, which are described below.

Weibel lung: The classical binary tree shaped lung model, in which an inlet flow at the root of the tree is used to compute volume and pressure at lower branches [61]. Each node of the tree computes the volume V and flow F of the corresponding branch:

$$\frac{dV_i}{dt} = F_{parent} \cdot C_1 + V_i \cdot C_2 + F_i$$
$$\frac{dF_i}{dt} = V_i \cdot C_3 - F_i \cdot C_4 - V_{R\_child} \cdot C_5 - V_{L\_child} \cdot C_6$$

Atrial heart cells: A 3-dimensional mesh of cells, where each cell propagates signals to its neighbors [65]. v_i is the membrane potential of cell i and is computed by the following equation.

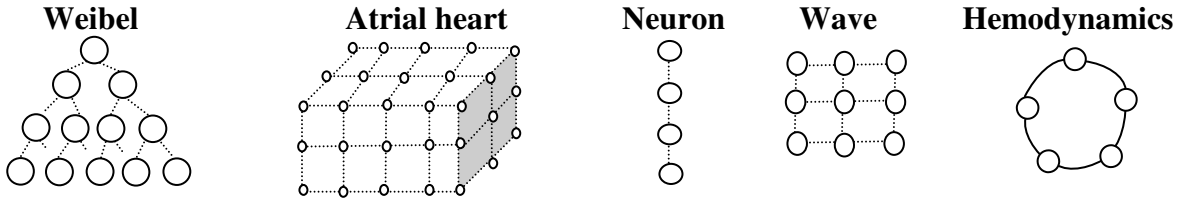

Figure 2-2: Various physical models and graphs of their representative structures.

$$\frac{dv_i}{dt} = (c_1 + (v_{x1} + v_{x2} + v_{y1} + v_{y2} + v_{z1} + v_{z2} - c_3 \cdot v_i) \cdot c_2) \cdot c_3$$ *Neuron synapses:* A 1-dimensional array of cells that simulates the firing of neuron synapses. *s* is the synaptic variable, *v* is the membrane potential, and *w* is a channel gating variable [54].

$$\frac{dv_i}{dt} = c_1 \cdot v_i + w_i - c_2 \cdot (v_i - c_3) \cdot (s_{i-1} + s_{i+1})$$
$$\frac{dw_i}{dt} = c_1 \cdot w_i - v_i$$
$$\frac{ds_i}{dt} = c_1 \cdot (1 - s_i) \cdot (v_i - c_2) - c_3 \cdot s_i$$

Wave: A wave model has a two-dimensional mesh network structure and is often used to model the propagation of sound, acoustics, etc [39]. The amplitude of the signal at node *i* is given by:

$$\frac{du_i}{dt} = c_1 \cdot (u_{x1} + u_{x2} + u_{y1} + u_{y2}) + c_2 \cdot u_i$$

Hemodynamics: A model that simulates the circulation of the human body, and includes submodels for the left/right heart ventricle and pulmonary/systemic tissues [59]. The hemodynamic model is arranged in a circular structure. Since there are many different types of equations to model this system, we omit the detailed descriptions here.

Large physical models such as those described above can be partitioned to hundreds of PEs in a network to achieve very fast simulation speeds. By maintaining the structure associated with the physical model during physical placement of PEs onto an FPGA, the routing overhead between PEs can be minimized. The natural structure of a physical model typically uses an optimally minimal number and length of wires, because only local communication between cells, lung branches, etc. is required. Previous work in

physical model simulation attempted to recover the physical model structure via heuristic annealing algorithms, after having converted the specification of the physical model's equations to an equation dependency graph [20]. However, finding the globally optimal solution for physical models containing thousands of equations and hundreds of PEs is not feasible with this approach. Instead of attempting to recover structures with heuristics, we propose to preserve the connections as they were modeled so as to minimize communication cost.

2.2 PHASE 1: MAPPING EQUATIONS TO VIRTUAL PES

Given the specification of a physical model that enumerates the physical model equations, a map must be built that groups equations into a structured virtual PE graph *G* that maintains the structure of the physical model. Equations must first be partitioned to a structured virtual PE graph of unconstrained size. Second, the graph must be reduced in size via folding to fit into available resources of the *target platform*. The target platform, which is typically an FPGA but could be an ASIC, is the device that the circuit will be placed on. There are limited resources on the target platform, thus folding is necessary for physical models whose structured virtual PE graphs exceed the size of the target platform.

2.2.1. PARTITIONING EQUATIONS

Let $G=(v,e)$, where $v=\{v_1,v_2\ldots,v_n\}$ is a set of n vertices and $e=\{e_1,e_2,\ldots,e_k\}$ is a set of k edges between vertices in v. Let $E=\{E_1,E_2,\ldots,E_m\}$ be the set of equations defined in the specification of the physical model. The set of vertices v represent virtual PEs, which may have equations from E allocated to them. The set of edges e represent communication channels between virtual PEs. If an edge $e_i=(v,u)$ exists, then there exists dependencies between the equations hosted in v and u. The graph G and its nodes and edges are defined by the structure of the physical model; a three-level binary-tree shaped Weibel lung model thus would have a graph that contains:

G_v={v1,v2,v3,v4,v5,v6,v7}

G_e={(v1,v2),(v1,v3),(v2,4),(v2,v5),(v3,v6),(v3,v7)})

Each equation E_i can be allocated to a vertex v_i in G according to a surjective mapping function $f : E \rightarrow G_v$. The function f depends on the structure of G, and maps groups of equations that represent the same physical element, e.g., a lung branch or atrial cell, to a single vertex. The result of applying the map function f to each equation yields a structured virtual PE graph G which maintains the basic structure of the physical model, and where each vertex (virtual PE) contains equations that represent some physical element of the physical model.

2.2.2. FOLDING

A physical model may be very large – a Weibel model with 11 generations contains 4000 differential equations. In order to meet the physical constraints of using a real

platform when mapping virtual PEs to physical PEs, the virtual PE graph *G* must first be scaled down. We perform *graph folding* on *G* by applying a homomorphic folding function φ that maps the larger graph to a smaller, more compact version *G'* while preserving the structure of *G.* In particular, φ maps *G* to *G'*, where the size *n* of the vertex set of *G'* is less than or equal to the number of supportable PEs in the target platform *S*; $\varphi : G \rightarrow G' \mid G'_n < S$. φ must also maintain the topology of *G* in *G*' by either maintaining an existing edge of *G* in *G',* or by merging the equations of vertex $a \in G$ into vertex $b \in G'$ such that the length of any edge connected to the merged vertices is constant. Informally, structures that are symmetric can generally be folded by cutting the graph into two subgraphs, and merging vertices that share the same position in each subgraph. Folding of graphs has been previously explored in graph theory literature [1][14][60]. Aleliunas [1]and Ellis [14] utilized folding techniques in order reduce the aspect ratio of rectangular graphs into forms that could be embedded onto a two-dimensional grid. Other work has developed algorithms for folding strongly balanced hypertrees in order to embed them into hypercube structures [60].

The exact definition of φ depends on the physical model structure. Different physical models can reuse the same folding functions as long as their structures match, thus a folding function for each structure type must be identified. A potential pitfall of folding is that structured virtual PE graph sizes tended to be reduced by halves, potentially creating a situation where almost half of the physical PE regions of the target platform are empty. One solution is to simply manually merge the final few virtual PEs if the size constraint of the target platform is only slightly less than the size of the structured virtual PE graph.

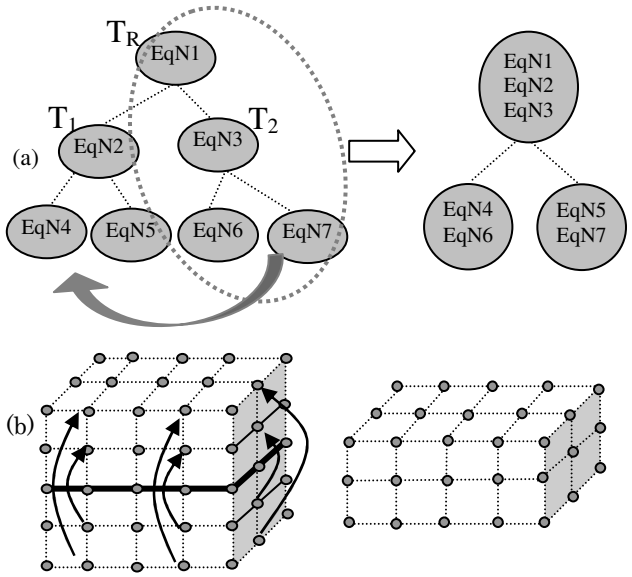

Figure 2-3: Folding graphs: (a) binary tree (b) 3-dimensional mesh.

The following section provides examples that target binary tree physical models, describing the mapping function *f* and folding function φ which result in the generation of a structured virtual PE graph.

2.2.3. LUNG MODEL EXAMPLE

A small Weibel lung model with three generations of bifurcating airways is structured as a binary tree with 2^3-1 = 7 branches, or fourteen interdependent differential equations for computing the pressure and volume of each branch. Let the set of equations *E* in the specification of the physical model be ordered such that the first *l* equations compute the volume and pressure of the root node, the next *l* equations compute the left child of the root, followed by *l* equations for the right child of the root, and so on. Equations can thus be initially partitioned to vertices in *G* via $f(e_i) = i / l$. The left side of

Figure 2-3(a) shows a representative structured PE graph, where EqNx represents the equations allocated to each node.

Consider if the target platform for the three-generation Weibel lung model is an FPGA that contains only enough resources for three PEs. Since each vertex in the graph represents a virtual PE that must eventually be physically placed, an excess of four PEs will not fit into the device. The graph can be folded as shown in the right side of Figure 2-3(a), by merging nodes in such a way as to maintain the graph structure. Let T_R be the root of the graph G, and T_1 and T_2 be the subtrees whose roots are the left and right children of T_R, respectively. We fold T_2 into T_1 by traversing down each subtree simultaneously, and moving any equations within the current node of T_2 into the equivalent node of T_1. The root node T_R is also merged into the root node of T_1, otherwise T_R would contain only a single child. This method maintains the adjacency of vertices in T_2 within T_1, as long as each subtree is symmetrical. Non-symmetrical structures can still be folded imperfectly by merging the vertices in T_2 that have no corresponding vertex in T_1 such that a minimum of additional edge length is required.

2.3 PHASE 2: MAPPING VIRTUAL PES TO PHYSICAL REGIONS

Once a structured graph of virtual PEs has been created, each virtual PE must be mapped to a physical location on the target platform. This mapping must consider both the average and maximum distances between PEs to reduce congestion and critical paths introduced via inter-PE communication channels. The simple solution to this problem is

to let a commercial synthesis tool flatten the design hierarchy, and run heuristic algorithms to select an appropriate placement. However, a circuit that contains hundreds of PEs is sufficiently complex such that modern tools cannot find good solutions without having additional constraints specified. Our approach defines a two-dimensional grid of physical PE regions on a target FPGA platform. Each physical PE region in the grid contains just enough resources to implement a single PE. Physical PE regions are defined at specific locations to create a two-dimensional grid that can be addressed using a XY Cartesian coordinate system. Whether or not the physical PE region actually contains a physical PE depends on the subsequent mapping. Virtual PEs can be mapped to physical PE regions on the grid using either structure-specific graph embedding techniques that place a guest graph into a host graph algorithmically, or by a generic simulated annealing approach with custom cost functions to reduce wire length.

2.3.1. FPGA PLATFORM TWO-DIMENSIONAL GRID

When performing place and route operations on large PE networks using commercial tools (Xilinx ISE 13.4) and a flattened netlist, we noticed that the critical path most often manifests between memories or logic components that belong to the same PE. Each PE in our design requires two memories (BRAMs), one multiplier (DSP), and approximately 250 lookup-tables (LUTs). We expected that communication channels between different PEs would be the primary cause of delay. Because of the complexity of large PE networks, the tools are not able to always place components of the same PE nearby each other. This problem can be addressed via the use of placement constraints during synthesis and place and route.

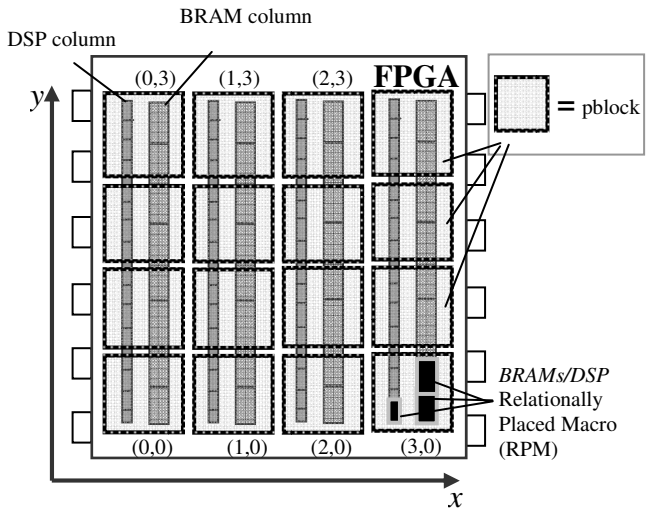

Figure 2-4: A 4x4 grid of physical PEs on a FPGA.

We first utilize Relationally Placed Macros (RPMs) to establish relative distances between PE memories. RPMs have been shown to provide faster circuit designs, even with modern tools [52]. On Xilinx FPGAs, a Cartesian coordinate system is used to specify the locations of components like DSPs and BRAMs (Figure 2-4). BRAM and DSP modules are physically located in homogeneous columns that stretch the height of the FPGA. We create an RPM for a PE using the Xilinx RLOC constraint by specifying that the offset between its instruction and data memories should be X=0, Y=1, and that the offset between the instruction memory and the DSP should be exactly X=-4, Y=0. The RPM thus ensures that PE memories are placed in neighboring BRAMs within the same BRAM column, and that the related DSP module is in the closest available location in a neighboring DSP column.

RPMs are useful for ensuring the close locality of BRAM and DSP modules that belong to the same PE, but we still must constrain each PE to specific physical PE

regions on the target platform. We utilize the Xilinx AREA_GROUP constraint during place and route to place PEs into physical PE regions. A selection of physical components of the FPGA (BRAM, DSPs, and slices) is first grouped into a *pblock*. We use the Xilinx PlanAhead tool to manually create pblocks in a grid structure. Each Pblock contains enough resources for a PE: two BRAMs, multiple DSPs, and more than 300 LUTs. The PEs in the design netlist can then be constrained via the AREA_GROUP constraint to a specific pblock region. The use of pblocks not only designates an exact location to place a PE, but also helps the place and route tools by requiring that the components in a PE hierarchy be placed within the pblock area. Since the area of the pblock is roughly what is required of a PE, the resulting PE implementation is densely packed and optimized. The use of placement constraints helps to shift the circuit critical path from internal PE connections to PE network communication channels.

We target a Xilinx XC6VSX475T. The Virtex6 platform contains approximately 297K LUTs, 2K DSP units, and 1K Block RAM (36KB each) memories. The grid size that can be constructed is 14x39, yielding a maximum of 504 PEs. For the vast majority of physical models, 500 PEs is sufficient for much faster than real-time simulation speeds. We note that our approach is not limited to one specific tool, platform or vendor; all FPGAs consist of a regular, reconfigurable fabric and most vendors allow blocks of resources to be grouped to create uniform structures. We consider only the specifically denoted FPGA and vendor (Xilinx) above to ease the discussion.

2.3.2. UTILIZING MODEL STRUCTURE

Physical models that exhibit common structures are able to take advantage of graph embedding techniques during physical placement. *Graph embedding* is the process of mapping a guest graph of architecture *g* onto a host graph of a different architecture *h*. Graph embedding has studied for at least 30 years by mathematical theorists, and many optimal solutions have been found for the embedding of structures like trees and meshes onto grids and hypercubes [11][35][58]. The typical metric that graph embedding algorithms are evaluated by is *maximum dilation*, or the maximum number of nodes that a wire may need to pass through to be completed. Since in physical model-solving PE networks the communication channels are point-to-point between PEs, the dilation is always exactly one. We thus alter the metric's definition slightly to be the maximum wire length between any two PEs. A second important metric is the *average dilation*, or the average wire length of all communication channels in the circuit.

By taking advantage of the research on graph embedding techniques to map virtual PEs to physical PEs on the target platform, the resulting physical placement can achieve smaller maximum and average dilation in the circuit. Smaller maximum dilation implies a reduction in the critical path, since once a PE has been constrained using RPMs and pblocks the longest wires for any complex network is typically connected between different PEs (as opposed to internal PE connections). Lower average dilation means that less routing resources will be required, which typically results in faster circuits [62]. In the next sections, we first define the graph embedding problem. We then show how to

utilize a graph embedding technique called H-tree construction to embed a binary tree structured physical model into a 2D grid of PEs.

2.3.3. GRAPH EMBEDDING

The graph embedding problem relates to the general mapping problem [6], where computational tasks must be placed onto a host architecture such that communication between PEs is minimized. Let $G_T = (V_T,E_T)$ be the guest graph, where G_T is the structured virtual PE graph (see section 4). Let $G_H = (V_H,E_H)$, where G_H is a graph that represents the physical PE layout. V_H is a set of all the physical PE regions, and E_H is initially empty because no connections exist until virtual PEs are placed. An embedding of G_T onto G_H is a result of applying an injective mapping function $\psi_V : V_T \rightarrow V_H$ to every vertex in G_T. Once the vertex mapping has been completed and a placement is created, then an additional mapping $\psi_E : E_T \rightarrow E_H$ can be inferred automatically by creating an edge $e = (u,v) \in E_H$ for every edge $p = (l,k) \in E_T$ where $\psi_V^{-1}(l) = u$ and $\psi_V^{-1}(k) = v$.

The quality of the graph embedding is denoted by the average and maximum dilation of the result of applying ψ_V and ψ_E. Since dilation in the context of PE networks on FPGAs with point-to-point communication is wire length, we use a basic Euclidean distance measure $D = sqrt((y2-y1)^2 + (x2-x1)^2)$. While possible to measure dilation using specific FPGA routing architecture characteristics [19], at a macro level the simple distance between physical grid locations will suffice.

Embedding binary trees onto two-dimensional grids is a solved problem [11][32][58]. Embedding a binary tree onto square grids has an $O(sqrt(n))$ maximum

dilation, where n is the number of generations of the tree. We utilize the H-tree construction technique (popular in VLSI) for the layout of tree architectures onto optimally sized square hosts [57]. H-tree construction creates an H-fractal tree where each subsequent branch of the tree alternates between horizontal and vertical tracks and wire length is halved. The graph is split recursively into four subtrees until leaf nodes can be placed. Where each split occurs, a track is used to host the root of the split and its two children, which are the roots of the actual 4 subtrees. Additional horizontal tracks are added for the three relevant parent nodes of each split subtree. Following the second split, leafs can be placed nearby their parents.

For square grids, the H-fractal tree method produces optimal results (in terms of dilation); however, for rectangular-shaped grids such as the 14x39 PE grid available on our target FPGA, H-tree construction requires some modifications. For example, the number of vertical tracks required for a 7-generation tree using the H-tree method is 31, or more than twice the number of available columns in the FPGA PE grid. We leverage the fact that our FPGA can route wires between PEs diagonally, as opposed to the strict row-column ordering of previous H-tree considerations [32]. Also, since the width of the target is the limiting factor to the number of possible recursive splits, it is not possible to maintain the ideal H-fractal shape in a rectangular grid. We therefore define a base case for the bottom k-generations of a tree that can no longer maintain H-fractal shape, such that an optimal placement of lower generations and leaf nodes can be completed. To embed the tree, we first perform placement via recursive splits down to the leaves of the

tree, than perform compaction and reordering of rows to further minimize maximum wire length.

Figure 2-6 illustrates the process. The binary tree is split into four subtrees and assigned to a quadrant of the grid. The blue lines mark connections between physical PE regions that contain a mapped virtual PE, which are marked with blue dots. The graph embedding follows the H-tree fractal shape design until the grid becomes too narrow to maintain the shape when placing the final two generations of the tree. At that point, a base case known placement is utilized to place the remaining virtual PEs into physical PE regions with minimal wire lengths. Rows four and ten contain no mapped virtual PEs, which unnecessarily inflates the maximum wire length. A simple greedy algorithm can be used to compact the graph embedding by moving the row with the longest wire until no improvement can be made.

2.3.4. EXAMPLE: BINARY TREE EMBEDDING ONTO 2D GRID

Embedded binary trees onto two-dimensional grids is a thoroughly researched area [11][32][58]. It has been proven that the graph embedding of a binary tree onto optimally-sized square grids have an *O(sqrt(n))* maximum dilation, where *n* is the number of generations of the tree. We utilize the H-tree construction technique that is used in VLSI for the layout of tree architectures onto optimally sized square hosts [57][66]. H-tree construction creates an H-fractal tree shaped liked that of Figure 2-5, where each subsequent branch of the tree alternates between horizontal and vertical tracks and wire length is halved. This process is done by splitting the graph recursively

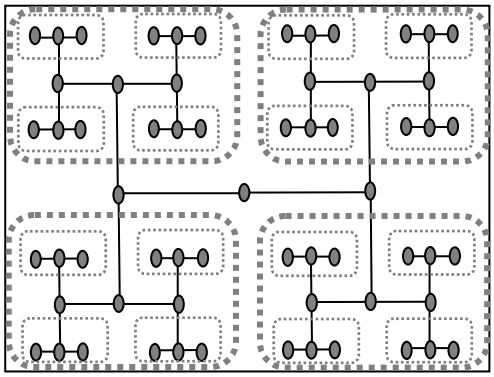

Figure 2-5: Six level binary-tree placed on a square 2-dimensional mesh. Dashed boxes indicate recursive splits into subtrees.

into four subtrees until leaf nodes can be placed. Where each split occurs, a track is used to host the root of the split and its two children, which are the roots of the actual 4 subtrees. In Figure 2-5, the tree is labeled by breadth-first ordering, such that the root is '0', the left child is '1', the right child is '2', and so on. Leaf nodes are not labeled for figure clarity. The thick dashed boxes represent the subtrees of the first recursive split; the row of vertices '0', '1', and '2' have a horizontal track allocated to them. The thin dashed boxes represent the subtrees created by a second recursive split of each of the first four subtrees. Additional horizontal tracks are added for the three relevant parent nodes of each split subtree. Following the second split, leaf nodes can be placed nearby their parents.

For optimally-sized square grids, the method demonstrated in Figure 2-5 produces optimal results (in terms of dilation). However, for rectangular-shaped grids such as the 14x39 PE grid available on our target FPGA, H-tree construction can not be immediately applied without some modifications. For example, the number of vertical tracks required for a 7-generation tree using the H-tree method is 31, or more than twice the number of

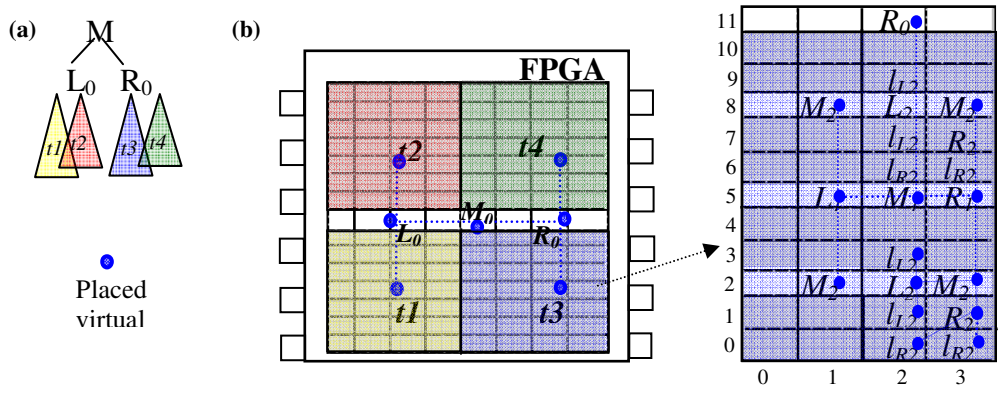

Figure 2-6: Embedding 7-level binary tree into a 2D grid: (a) Initial split; (b) two more recursive splits. For clarity not all branches shown.

available columns in the FPGA PE grid. We can take advantage of the fact that our FPGA can route wires between PEs diagonally, as opposed to the strict row-column ordering of previous H-tree considerations [32]. Also, since the width of the target is the limiting factor to the number of possible recursive splits, it's not possible to maintain the nice H-fractal shape of the graph embedding in a rectangular grid. We therefore define a base case for the bottom k-generations of a tree that can no longer maintain H-fractal shape, such that an optimal placement of lower generations and leaf nodes can be completed.

To embed the tree, we first perform placement via recursive splits down to the leaves of the tree, than perform compaction and reordering of rows to further minimize maximum wire length.

1. Separate the grid into 4 quadrants to host the initial split of the tree.
2. Place the root node M_0 and its children L_0, R_0 in the center row of the grid.

M_0 is placed in a column in the center of the grid. L_0 and R_0 are placed in a middle column of the neighboring upper and lower quadrants

3. Place each child of L_0 and R_0 onto the same vertical track as its parent, and onto the center row of a quadrant (Figure 2-6(a)).
4. Recursively split each subtree by placing the children of the subtree's root on the same row, and allocating additional rows to host new subtrees (Figure 2-6(b)).
5. At generation N-1, utilize a known placement to place the final levels (non-fractal shape).

The process described in the steps above can be seen in Figure 2-6. The binary tree is split into four subtrees and assigned to a quadrant of the grid. The blue lines mark connections between physical PE regions that contain a mapped virtual PE, which are marked with blue dots. The graph embedding follows the H-tree fractal shape design until the grid becomes too narrow to maintain the shape when placing the final two generations of the tree. At that point, a base case known placement is utilized to place the remaining virtual PEs into physical PE regions with minimal wire lengths. Note that rows four and ten contain no mapped virtual PEs, which unnecessarily inflates the maximum wire length. A simple greedy algorithm can be used to compact the graph embedding by moving the row with the longest wire until no improvement can be made.

Chapter 3. PLACEMENT OF NON-STRUCTURED APPLICATIONS

Simulated annealing is a general method that can map any structured virtual PE graph onto physical PEs. This approach is useful when a physical model has no obvious structure for which a graph embedding algorithm could be used, such as unbalanced or asymmetrical trees [16]. Simulated annealing also yields useful comparisons to embedding by providing reasonable PE placements.

The simulated annealing approach utilizes methods previously described for timing-driven placement in the VPR tool [35]. We define a cost function that considers FPGA architectural features, wiring cost, and timing cost (critical path length); it is shown experimentally that our cost function correlates linearly with resulting circuit frequency. We also present a neighbor function that swaps virtual PEs based on the relative placement of connected virtual PEs. Our neighbor function provides faster convergence and results in lower cost placements than random swaps.

3.1 COST FUNCTION

The cost function is calculated each iteration of the simulated annealing algorithm to determine the effect of a perturbation on the current solution state. The *Wiring_Cost* term determines the cost associated with routing all of the inter-PE nets in the design. This term is a summation of all the wire lengths in the design that are routed between physical PE regions:

$$Wiring\_Cost = \sum_{n=0}^{N_{NETS}} D(n_{SNK}, n_{SRC}) \quad (1)$$

$D(n_{SNK}, n_{SRC})$ is the distance of the net from source to sink, defined below. The VPlace algorithm of VPR uses a similar equation to calculate routing cost, although using a Manhattan distance measure and a factor to compensate for the extra resources required by highly connected nets. Minimizing the wiring cost during simulated annealing produces a placement with less congestion, resulting in faster circuit implementations [62].

The Timing_Cost term considers the impact of the longest wires in the design, which are most likely to form a critical path in the circuit. Similar to the T-VPlace algorithm, wires are assigned a weight depending on their length [35]. Wires closer to the maximum wire length of the design are weighted heavily, while shorter wires have less impact. Note that our current implementation assumes a delay that corresponds linearly to the distance between physical PE regions; future work could achieve more accurate timing cost estimates by modeling the delay between physical PE regions in the target FPGA. Each wire is assigned a weight according to the below equation.

$$\text{Weight(n)} = 1 - \frac{\text{Wmax - D}(n_{SNK}, n_{SRC})}{\text{Wmax}} \quad (2)$$

Wmax is the maximum PE-to-PE wire length in the design. The Timing_Cost for each net is calculated as below.

$$\text{Timing\_Cost(n)} = \text{D}(n_{SNK}, n_{SRC}) \cdot \text{Weight(n)}^{\text{W\_exp}} \quad (3)$$

W_exp is a user-defined exponent that causes higher-weighted wires to have more impact on timing cost. In T-VPlace this exponent is called the *criticality exponent*.

FPGAs commonly contain architectural features which prevent placement of logic in specific areas. For example, Figure 3-4 shows a Virtex6 FPGA that has a large gap in the middle for monitoring/programming components. Wires routed across such gaps incur an additional timing cost penalty through exponentiation of the distance by a user-defined constant *Arch_exp*. The exponentiation adds high penalties to wires that are both long and cross a gap. Whether or not a wire crosses a gap is determined by projecting a vector *v* from source to sink, and recording intersecting physical PE regions in a set *route*. Physical PE regions that contain gaps or features restricting logic are annotated and recorded in a set *ignored*. Distance is then calculated based on whether or not *route* and *ignored* are disjoint.

$$\mathrm{D(n)} = \begin{cases} dist(n) & route \cap ignored = \emptyset \\ dist(n)^{Arch\_\exp} & \text{otherwise} \end{cases} \tag{4}$$

Above, *dist(n)* is the distance measure, e.g., Euclidean or Manhattan distance. The total timing cost for the PE network placement is equal to the summation of the timing cost of every net.

Furthermore, the cost function incorporates previously described auto-normalization techniques to ensure that the cost of a single perturbation is related to relative changes in both the wiring and timing costs.

$$\Delta Cost = \lambda \cdot \frac{\Delta\mathrm{Timing\_Cost}}{\mathrm{Previous\_Timing\_Cost}} + (1-\lambda)\frac{\Delta\mathrm{Wiring\_Cost}}{\mathrm{Previous\_Wiring\_Cost}} \tag{5}$$

λ controls how much weight to give each cost term after each iteration. For all experiments in this paper we use $\lambda = 0.5$.

Figure 3-2(a) shows a linear regression representing how the cost function relates to the resulting circuit frequency of a PE network once placed and routed.

3.2 NEIGHBOR FUNCTION

In simulated annealing, a *neighbor function* is a local perturbation of a solution, which may or may not improve its overall quality. An initial solution, which is likely to be far from optimal, can be computed randomly or using an efficient polynomial-time heuristic. A simple neighbor function in the context of our PE placement problem is to randomly select two PEs within the network and swap their locations; we use this neighbor function as a baseline.

The neighbor function presented here attempts to cluster connected PEs together, with the goal of reducing wire lengths in the process. A random physical PE region P_1 that contains a mapped virtual PEV is selected first. Each connection $e = (P_1,P_p)$ in V is evaluated, where P_p is the physical PE region of the virtual PE connected to V. A Euclidean vector is constructed from P_1 to P_p. An average of all the vectors originating from P_1 identifies a physical PE region that minimizes the average wire length of all connections to the PE if the virtual PE were placed there.

If a virtual PE does not already occupy the target physical PE region, then P_1 can be placed immediately on the target. Otherwise, the algorithm examines each of the target PE's neighbors. If any neighbor is unoccupied, then P_1 is moved there. If all neighbors

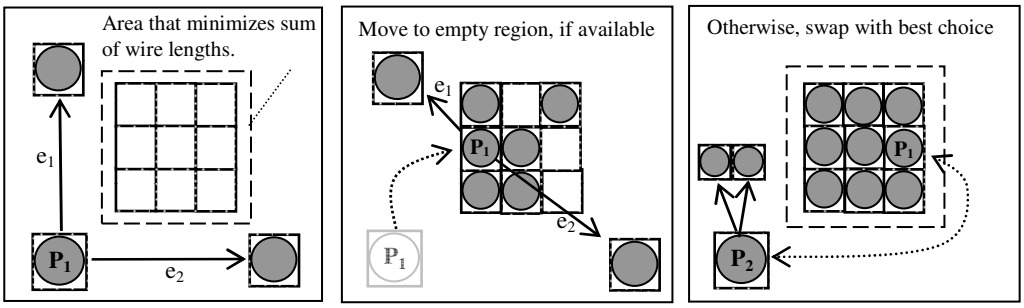

Figure 3-1: Neighbor function minimizes wire length of PE1. (a) Area to move PE1. (b) PE1 moved to empty region if available, (c) otherwise, PE1 swapped with PE2.

are occupied, the algorithm selects the physical PE region whose average connection vector endpoint is closest to P_1 for swapping.

Figure 3-1 provides an example. P_1 is randomly selected. An average of the two connections e_1 and e_2 yields an area of the platform where P_1 should be placed to minimize wire lengths. If there is an empty physical PE region than P_1 is moved there. Otherwise, each physical PE region in the area is evaluated and the best candidate is swapped with P_1. The best candidate is determined by computing the potential cost reduction

3.3 SIMULATED ANNEALING ALGORITHM STRATEGIES

The cooling schedule used during simulated annealing can cause dramatic differences in the obtained solution [45]. We experimented with linear [$T(t) = T_0 - \eta t$], geometric [$T(t) = T_0 / t$], and exponential [$T(t) = T_0\alpha^t$] cooling schedules. We found that both linear and geometric schedules produce a configuration with a similar cost for a given physical model, while the exponential schedule ($\alpha = 0.99$) yields a configuration that is highly dependent on the initial random placement and does not generally produce a

good result. This is due to the quickly decaying nature of the exponential function, which makes it difficult to escape local minima in the solution space. All experiments in this paper utilize a geometric cooling schedule.

Figure 3-2(c) shows the effect that modifying the number of solution perturbations performed per iteration has on the average final cost of 5 different models (Weibel10, neuron1d, neuron2d, asymmetrical tree, and random). More perturbations/iteration implies a longer runtime, but with more swaps occurring at a higher temperature. Using 16 perturbations/iteration gives a 19% decrease in the final cost, on average, over 1 perturbation/iteration. The runtime of simulated annealing for a large 500 PE network using 32 perturbations/iteration is less than 10 minutes, while the runtime when using 1 perturbation/iteration is approximately 2 minutes.

The initial temperature is determined automatically for each run by performing trial annealing runs and searching for a temperature that results in a given acceptance ratio [25]. We use an acceptance ratio of 0.9 in this paper for all simulated annealing runs. Additionally, a restart functionality resets the current configuration if no perturbation produces a higher-quality configuration after 250 consecutive iterations, hence the oscillations on the right-hand side of Figure 3-2(b) for the random neighbor function. The reset functionality helps to ensure that the annealing process does not get stuck in a local minima after having accepted a worse solution. When a reset occurs, the best configuration seen thus far is reloaded, but the anneal schedule continues without being reset. Instead of continuing along a worse path indefinitely, the algorithm can reset to a better known configuration and continue. The number of iterations allowed before a reset

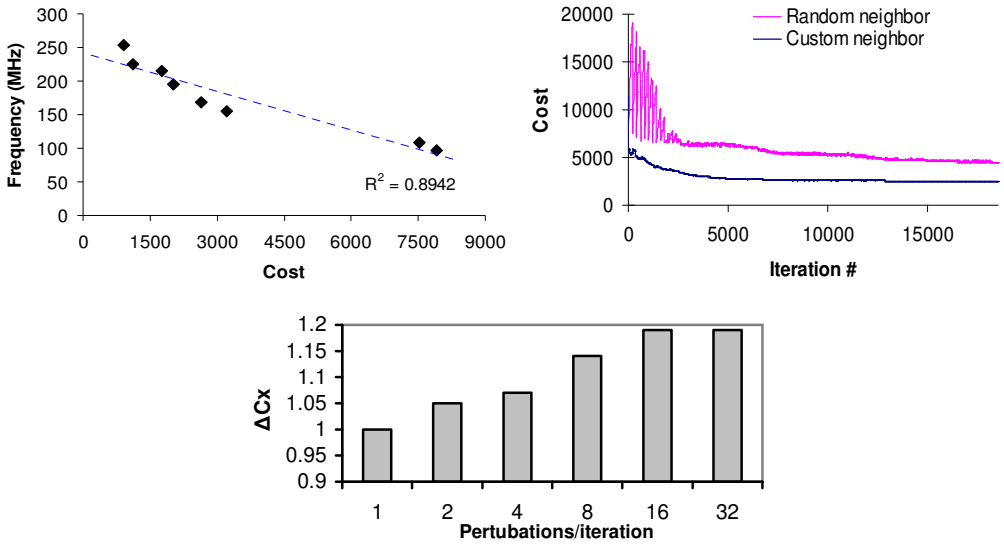

Figure 3-2: Analyzing the annealing results. (a) Cost function correlates with frequency, (b) comparing random to custom neighbor, and (c) average effect of perturbations per iteration; ΔCx is the improvement over 1 perturbation / iteration.

is configurable - setting the number too low may ruin the hill-climbing capabilities of the algorithm, but setting too high may result in longer runtimes. The annealing process ends if 1000 consecutive iterations do not improve the configuration.

3.4 PLACEMENT RESULTS

Figure 3-3 shows the resulting circuit frequencies of implementing PE networks on an FPGA with the four different techniques. The first and second columns use the equation partitioning performed by the PE network compiler. The third and fourth columns use a partitioning based on the structure of the model, and the corresponding graphs were folded to meet target platform constraints.

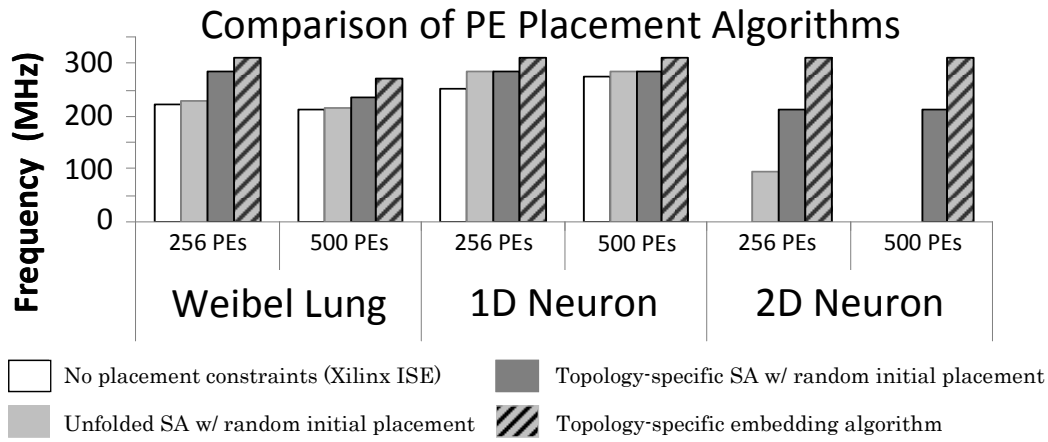

Figure 3-3: Freqs. of PE networks. Missing entries did not finish place-and-route.

The first columns do not use physical placement constraints. These data points represent the ability of Xilinx ISE to place and route the PE network compiler-generated circuit without any guidance. The second columns use the same RTL as the first, but with placement constraints generated via simulated annealing that map virtual PEs to physical PE regions. The difference between the two columns in each case show that applying placement constraints through an automated simulated annealing process can provide some marginal improvement, and is even able to route a network that Xilinx was unable to do without constraints.

The third and fourth columns use a partitioning of equations based on the model's structure. The third column uses simulated annealing and the fourth column uses a topology-specific graph embedding algorithm to generate the placement. The graph embedding approach is almost always able to produce a circuit that tops 300 MHz. The ceiling for the circuit frequency in a PE network is approximately 310 MHz for the selected platform. We determined the ceiling by placing and routing a circuit with a

single PE and evaluating the critical path of the internal datapath. It is not possible for a network of PEs to operate faster than the ceiling, and any decrease in performance can be attributed to critical paths introduced by inter-PE connections. The graph embedding approach is typically able to minimize the critical path length and provide placements that allow the circuit to approach the ceiling. The only embedding example that could not reach the ceiling of 310 MHz is the 10-generation Weibel lung model using 500 PEs. Because the two-dimensional grid of the physical PE regions is narrow, an optimal embedding of the tree cannot occur. Wire lengths between successive generations are longer, resulting in longer critical path delays.

Missing columns indicate that the Xilinx tools were not able to place and route the design due to high congestion. The compiler that partitioned the equations and created the communication network could not sufficiently reduce the data dependencies between PEs for these large physical models, resulting in an overwhelming number of wires in the network. Note that these designs are routable if we use either a topology-specific simulated annealing or graph embedding approach.

Figure 3-4 depicts the placement of the first few generations of a nine generation Weibel model on 256 PEs, as captured by the Xilinx PlanAhead tool. An overlay of nodes and connections shows where virtual PEs have been mapped onto the FPGA.

Figure 3-4(a) shows how Xilinx ISE implements the PE network in the absence of additional constraints that map to specific physical regions. Due to the complexity of the circuit, the resources of a single PE can be spread over a wide area, thus we have marked only the approximate central location of the first four generations of the left subtree of the

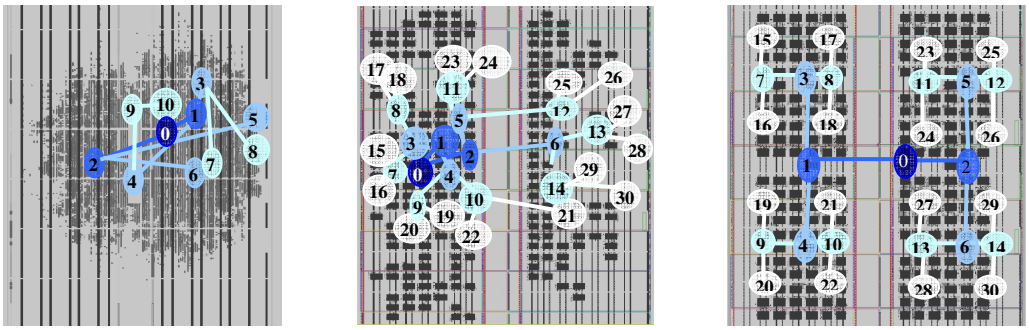

Figure 3-4: Different placements of 256 PE Weibel lung model: (a) unconstrained placement; (b) simulated annealing; (c) graph embedding.

graph. Note that if we do not specify placement constraints, the tool places PEs at sub-optimal locations, yielding potentially long wire distances between PEs. For example, the wires between node two and its children five and six span more than halfway across the entire design.

Figure 3-4(b) depicts the placement produced by the simulated annealing algorithm. Each black block indicates a virtual PE that has been mapped to a physical PE location. An empty space is a physical PE region onto which no virtual PE was mapped. As a consequence of simulated annealing, nodes that share connections tend to be grouped together, while the overall tree tends to expand outward from the center of the grid toward leaf nodes grouped on the perimeter.

Figure 3-4(c) shows the tree embedded in the host grid using a topology-specific algorithm. The center of many common (Xilinx) FPGAs contains immutable logic, and minimization of the routing across the center is desired. This embedding requires a single wire across the gap, at the second generation of the tree.

We also measured the static and dynamic power of each case using the Xilinx XPower Analyzer. The unconstrained placement uses approximately 20% less absolute dynamic power on average than both the simulated annealing and embedding constrained placement approaches. The Xilinx ISE options for power reduction during circuit implementation were disabled for every reported experiment, thus timing is the driving optimization goal.

3.5 USING GRAPH DRAWING ALGORITHMS TO GENERATE INITIAL PLACEMENT

Thus far, the placement techniques have been limited to structured physical models with topologies for which embedding algorithms are known. For example, the H-Tree embedding for binary trees is not immediately translatable to assymmetric trees, which are representative of lung blood circulatory models [16][28]. Furthermore, some physical models are fully irregular. For example, a spiking neural network model, which simulates a human brain cortical network, consists of interconnected groups of neurons with statistically generated connectivity [42]. Such models result in random, commonly non-planar graph structures that make embedding difficult, if not impossible. Furthermore, if the structure is not clear, then a structured virtual PE graph can not be created. We refer to models that are not easily embedded into the 2-dimensional FPGA host grid as *non-structured models*.

Simulated annealing can generate placement constraints for non-structured models, thereby improving circuitry frequency compared to placement using commercial tools with unconstrained placement. In this section, we describe improvements to the simulated

annealing algorithm that provide better solutions with increased clock frequency for unstructured models.

As described previously, our simulated annealing placer uses a randomly generated initial placement as an initial solution. To improve the overall quality of results, we introduce a heuristic to generate a lower cost initial solution compared to a random placement. One of the benefits of simulated annealing algorithms is that good solutions can be found even from random, high entropy initial configurations. However, by starting with a good placement that is assumed to be close to an approximately global optimal solution, the annealing process can focus on performing target-specific optimizations and find a good solution in less time. Decades of research into graph drawing techniques has produced algorithms that attempt to minimize edge distances, edge intersections, and the area of drawing size [15]. Such techiques can be leveraged to generate an initial layout of network PEs on the FPGA's two-dimensional grid that has substantially less cost compared to a random initial layout. Previous research has shown that a better-than-random starting solution can yield better final solutions, on average [25].

We extended our previously introduced PE network compiler to use graph drawing algorithms to generate an initial placement. The approach uses the graph drawing and visualization tools Graphviz and/or Tulip to generate a layout of an existing, possibly folded, virtual PE network. Graphviz and Tulip are freely available open-source projects. Given a DOT-formatted input file that describes the nodes (PEs) and edges (wires) of the graph, the output of a graph drawing tool is the virtual PE network with X,Y coordinates assigned to each node. The placed graph is then transposed onto the FPGA 2-dimensional

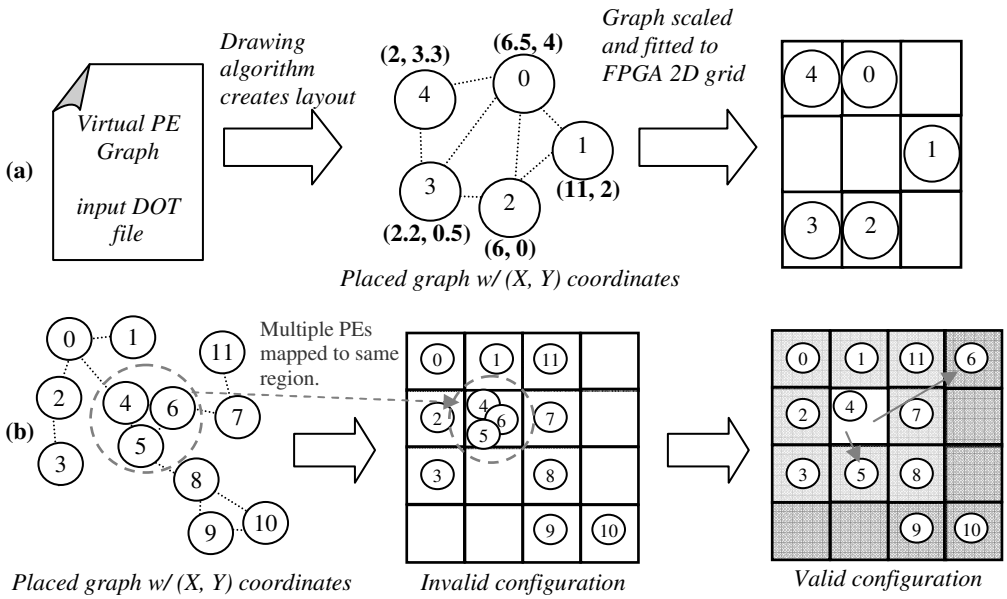

Figure 3-5: Finding initial layout using graph-drawing algorithms. (a) Finding initial layout, and (b) creating a valid configuration.

grid by normalizing the aspect ratio of the graph with that of the grid. Graph nodes are placed onto physical PE regions based on their normalized X,Y coordinates from the placed graph output of Graphviz. Figure 3-5(a) illustrates the process of mapping Graphviz output to the FPGA.

The result of the initial mapping may be an invalid configuration, which may occur if more than one virtual PE is mapped to the same physical PE region. The normalization process shrinks a graph layout down to fit the FPGA ; sometimes, several virtual PEs may overlap. In such scenarios, a process, shown in Figure 3-5(b), can legalize the configuration. Our solution evaluates each physical PE region in turn to determine if it contains multiple virtual PEs. If so, then all but one must be relocated to empty physical PE regions. Ideally they will be relocated to nearby empty regions to maintain the

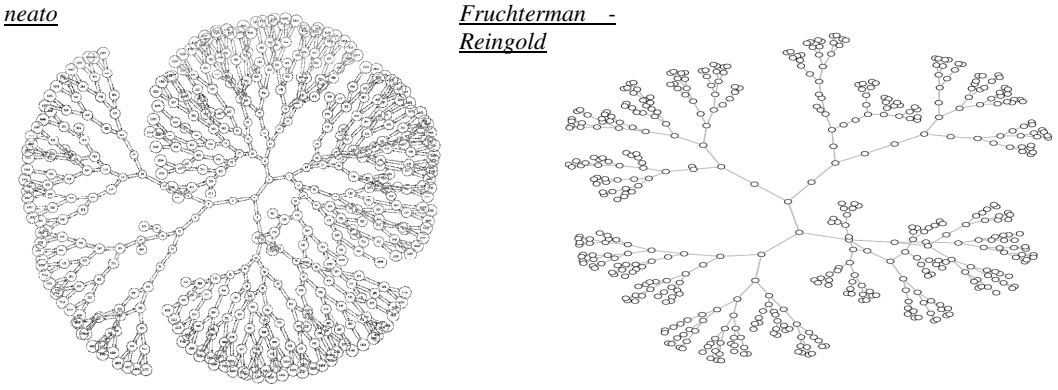

Figure 3-6: Using different graph drawing algorithms to the same graph.

placement generated by the drawing algorithm as much as possible. Our approach is to search an expanding radius of neighbor regions until an empty region is found. Neighbors with a distance of 1 are searched first (shaded lightly in Figure 3-5(b)), then a distance of 2 (shaded dark), etc. Thus, overlapping virtual PEs are moved a minimum distance from their original location, and an empty region is guaranteed to be found, provided that *num_PEs* <= *num_regions*. Other approaches besides the presented greedy algorithm are possible.

3.5.1. GRAPH DRAWING ALGORITHMS

Many graph drawing algorithms exist, and each may produce a different layout of the same graph. Various categories of graph drawing algorithms exist, including force-directed, orthogonal or planar, layered, and *tree* layout strategies.

As shown in Figure 3-6, different graph drawing algorithms give different layouts. Force-based algorithms, like neato and Frutcherman-Reingold, give reasonable layouts for almost any model. Such algorithms utilize their own heuristics to converge on a good

solution that may not be optimal, but will at least be within some local minima with reduced edge lengths and better layout than a random approach.

In contrast to force-based drawing algorithms, other algorithms are specific to certain structures or graph-types. Heirarchical and tree based algorithms perform best if the model has a specific structure that matches the algorithm. Often such algorithms work *only* on graphs of a specific structure, e.g., a tree drawing algorithm may fail when trying to draw a graph with cycles.

3.5.2. PLACEMENT USING GRAPH DRAWING

Table 3-1 shows initial and final costs of 5 different graph-drawing algorithms applied to structured and non-structured models. The cost is calculated by the equation presented in Section 0, which considers the wire length total, critical path, and the architectural features of the target platform (Xilinx Virtex 6). We implement and compare the force-based algorithms neato, fdp, and Frutcherman-Reingold (FR), the heirarchical algorithm Sugiyama (Sg), and the radial layout algorithm circo. Each reported result is the average of 5 simulated annealing runs for each configuration. We establish initial temperatures using an acceptance ratio of 0.9, although a lower acceptance ratio might better preserve the initial placement seed yielded by the graph drawing algorithm.

Run times of the simulated annealing algorithm vary with network size and connectivity. The minimum run time was 36 seconds for 256 PE binary tree model; the maximum was about 6 minutes for the 2D neuron model. The run time of the Xilinx tools is not affected by the placement constraints, requiring 1-3 hours depending on the model.

In contrast, implementing an embedding algorithm can take hours to days, depending on the complexity of the structure. The simulated annealing approach thus provides a faster and more generalized method for creating faster circuits automatically without requiring structure-specific implementations.

Every drawing algorithm, except for the heirarchical algorithm Sugiyama (Sg), produces a layout with lower cost than a random mapping. Sugiyama creates very wide layouts with large aspect ratios with most nodes at the bottom, which makes for poor starting points to the annealing process.

The force-based algorithms neato and fdp generate the best initial layouts, reducing the initial cost compared to random by an average of 4.31X and 4.22X. In two cases circo produces the best initial layout. Circo produces layouts similar to H-tree embedding for tree-like acyclic graphs. The PE network compiler may generate networks that recapture some of the tree structure of the original model, thus the initial layout resembles an H-tree.

The higher-quality initial layouts that are generated by drawing algorithms result in a lower final costs on average for the neato, fdp, and circo algorithms. The fdp and neato algorithms produce final costs that are 17% and 14% lower, on average.

The placement constraints generated by the best-performing drawing algorithm for each model were used during an implementation of the circuit; the results are reported in Figure 3-7. The first three columns do not use a graph folding technique to capture the model-structure. Instead, a PE network compiler produced the architecture. In contrast, the columns labeled "Topology-specific" had equations partitioned according to the

Structure (#PEs)	Initial cost						Final cost after simulated annealing					
Drawing algorithm	rand	neato	fdp	FR	Sg	circo	rand	neato	fdp	FR	Sg	circo
	Structured graphs											
bitree(255)	*11500*	*1433*	***1128***	*1389*	*20256*	*4847*	*1028*	*944*	***920***	*1052*	*9028*	*990*
bitree(500)	*19496*	*9005*	***8073***	*10282*	*11782*	*9006*	*1917*	*1702*	***1606***	2848	*2455*	*1746*
linear(256)	*10335*	***899***	*1045*	*1280*	*1785*	*2529*	*861*	***704***	*759*	*1019*	*905*	*837*
linear(500)	*19366*	*6047*	*8278*	*8415*	***4784***	*8157*	*1600*	*1355*	*1366*	*2651*	*1582*	***1319***
mesh(256)	*29832*	***8386***	*8466*	*9082*	*19496*	*18204*	*8778*	*7728*	***7715***	*8329*	*9996*	*8629*
mesh(500)	*96K*	*73K*	*69K*	*67K*	*N/A*	***65K***	*63K*	*61K*	***58K***	*60K*	*N/A*	*63K*
	Unstructured graphs											
a_tree(500)	*19806*	*6836*	*9242*	*7833*	*13079*	***5931***	*2697*	*1981*	*2423*	*2353*	*2664*	***1875***
noStrc(300)	*9440*	***2329***	*2379*	*2490*	*3854*	*5249*	*2290*	*1950*	***1905***	*2060*	*2153*	*2117*
noStrc(500)	*13135*	*6454*	*6124*	*7574*	*8007*	***5355***	*3346*	***2713***	*2865*	*2923*	*3500*	*2894*
Avg. impr. over random (X)	--	***4.31***	*4.22*	*3.70*	*2.40*	*2.41*	--	***1.17***	*1.14*	*0.95*	*0.85*	*1.12*

Table 3-1: Initial and final cost of graph drawing algorithms for structured and unstructured models. 'N/A' entries could not complete the drawing algorithm.

model structure, which typically reduces the number of wires in the circuit and results in less congestion during place-and-route.

The embedding approach achieves close to optimal implementations, because the embedding placement migrates the critical path from intra-network PE-to-PE communication channels to intra-PE logic. The three rightmost models, which are non-structured models (asymetrical tree counts as non-structured becaus there is no simple embedding), lack folded or embedded results because such models can not use those approaches. The largest model, neuron2d(500), could not not be implemented by any unfolded approach due to high number of wires in the design. The neuron2d(256) results could only be routed if placement constraints were used. In some cases where the

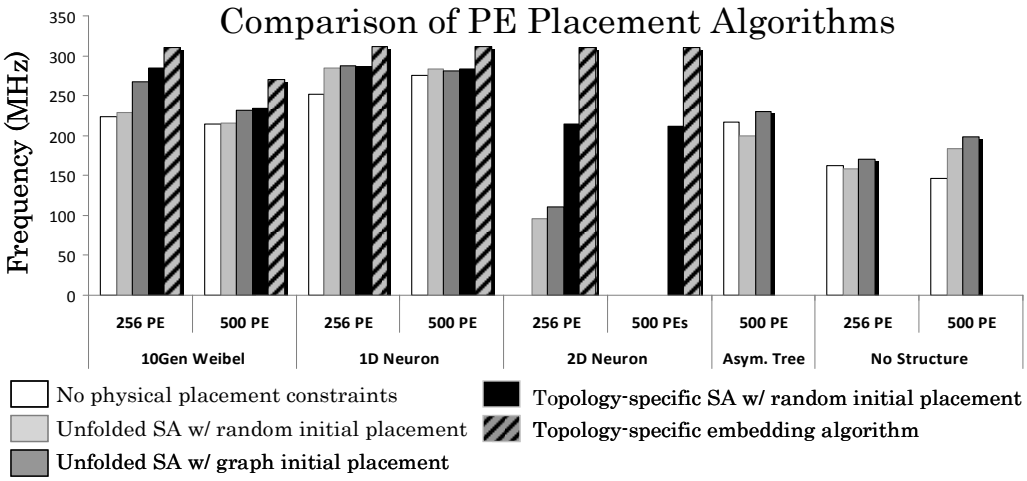

Figure 3-7: Freq. of circuits for various models and methods. Missing results could not be routed by Xilinx or are not applicable.

connectivity of the network is low, Xilinx is able to perform place and route effectively without placement constraints. The described approaches have the most benefit for larger, more connected structures.

The third columns in Figure 3-7 shows the frequency of the circuit when using the graph drawing-algorithm approach on an unfolded graph. The second columns show a random initial-layout approach. Comparing these two results for each model yields the improvement in circuit frequency due to better initial placement using graph drawing algorithms, which is an average of 9% higher frequency. The leftmost columns show no placement constraints. Comparing "No physical placements constraints" with "Unfolded SA w/ graph initial placement" yields a 13% average improvement in circuit frequency due to placement constraints and initial graph-drawing layouts, even when not considering model-specific structure.

Chapter 4. REDUCING NETWORK CONGESTION

The PE network compiler can generate implementable designs for models consisting of up to approximately a few thousand equations. However, compiling very large models consisting of tens of thousands of equations will result in designs that saturate the available FPGA routing resources. The architecture of the generated network is point-to-point, meaning that every data dependency between PEs requires a communication channel between those PEs. If a network contains a few thousand such channels, then the routing resources of the FPGA will be saturated and the network may be too congested to complete place-and-route. This paper describes a new *reduction phase* of the synthesis problem, wherein following partitioning of equations to PEs, the reduction phase reduces the network communication architecture to fit the routing resource constraints of a target FPGA. The reduction phase of the network is combined with new time-multiplexing of channels to trade off final circuit frequency for latency.

For example, Figure 4-1 shows a set of model equations, partitioned into 8 PEs. Each of the 8 PEs computes equations that have a data dependency on variables partitioned to other neighboring PEs. The partitioning and dependencies yield an *architecture graph*, which specifies point-to-point connections between processing elements. Each connection *a*, *b*, *c,* etc. in Figure 4-1 represents a communication bus when the network is implemented on an FPGA. A new reduction phase generates a *reduced architecture graph*, wherein the total number of edges of the graph have been reduced to meet routing constraints of the target FPGA, but the reachability of the graph remains unchanged such

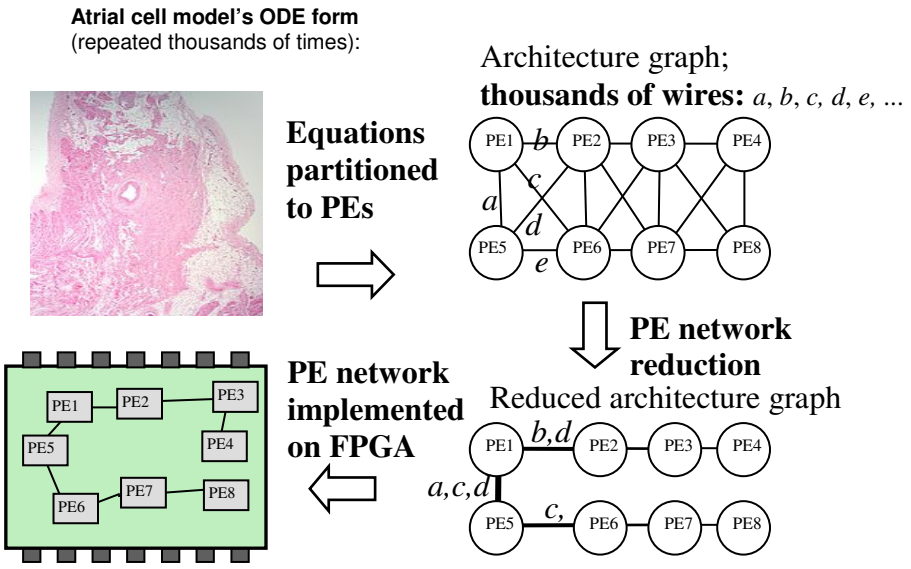

Figure 4-1: Reducing network connections leads to a faster, more routable design.

that intermediate PEs may serve as routing switches to store and forward data. The communication tasks are then scheduled using time-multiplexing over shared channels available in the reduced architecture graph.

4.1 IMPACT OF MODEL SIZE ON SYNCMC CIRCUITS

Using the above-described approach, compiling large models into PE networks may produce designs that saturate the available FPGA resources. The partitioner component of the compiler is constrained by the available LUTs, BRAMs, and DSPs available on a target FPGA. However, the partitioner has no such fixed resource constraints on the number of inter-PE connections allowed. The partioner provides a parameter that drives the partitioning heuristics to favor either size or performance of a network. Favoring size produces designs with fewer inter-PE connections and more computation cycles per iteration; favoring performance produces designs with more inter-PE connections and

	ρ	#PEs	#wires	Comp. cycles	Comm. cycles	Total cycles	Freq (MHz)	KIters / sec	ODEs/ sec x 10^9
Weibel 10 (2048 ODEs)	0.1	420	849	167	11	178	207	1.16	4.76
	1	441	905	151	11	162	195	1.20	4.92
	10	480	1261	144	13	157	173	1.10	4.51
Weibel 11 (4096 ODEs)	0.1	500	1130	277	13	290	183	0.63	5.16
	1	500	1268	260	12	272	171	0.63	5.14
	10	500	**2564**	252	17	269	**--**	**0**	**0**
Weibel 12 (8192 ODEs)	0.1	500	1298	496	14	510	158	0.310	5.07
	1	500	**2177**	483	18	501	**--**	**0**	**0**
	10	500	**5170**	481	31	512	**--**	**0**	**0**
Weibel 13 (16384 ODES)	0.1	500	**2576**	950	21	971	**--**	**0**	**0**
	1	500	**3519**	946	29	975	**--**	**0**	**0**
	10	500	**12985**	946	65	1011	**--**	**0**	**0**

Table 4-1: The clock frequency drops and designs eventually become unroutable as model complexity increases.

fewer computation cycles per iteration. For very large models, setting the performance/size parameter to favor size can not compensate for the high amounts of data dependencies, and synthesis tools may not be able to achieving acceptable clock frequencies for the compiled PE network. Note that exploring the configuration space of PE networks is an interesting problem itself, which we previously researched [40].

Table 4-1 shows the throughput of a number of PE networks emulating various models. The results for partitioner size/performance ratio ρ is shown for 0.1 (favor performance), 1 (balanced), and 10 (favor size). *Comp. cycles* describes the number cycles in the computation phase of each model iteration; similarly, *comm. cycles* describes the number of communication cycles per iteration. *Freq* gives the final circuit clock frequency. *KIters/sec* describes how many thousands of model iterations can be executed per second. Finally, *ODEs/sec* describes how many ODEs are solved per

second, which is a function of the total number of cycles, number of model ODEs, and final circuit clock frequency. Entries with 0 frequency could not be placed and routed using Xilinx ISE 14.2, due to designs that were too congested for a Virtex6-475T architecture. From the table, we can observe that designs with more than 1500-2000 wires can not complete place-and-route.

Clearly there is a need to reduce the required routing resources for very large models, since peer-to-peer network architectures do not scale well on FPGAs. The number of communication cycles compared to computation cycles per iteration is relatively small, usually less than 5%-10% of total cycles. Introducing time-multiplexing may allow for large models to still complete place-and-route, but with added communication cycles per iteration.

4.2 INTRODUCING TIME-MULTIPLEXING COMMUNICATION INTO SYNCHMCS

This section describes a basic (naive) approach to performing an additional graph reduction phase following PE allocation and partitioning of equations to PEs. Figure 4-2 shows an updated compilation flow, where the new reduction phase takes as input the architecture graph, and produces a reduced architecture graph.

Reduction of the graph is an optimization problem with the following definition: Given an architecture graph $G = (V, E)$, generate a graph $H = (V, E')$ such that the reachability of $G = H$, and the metrics $Cost_{SIZE}$ and $Cost_{PERF}$ (introduced later) are minimized. The reachability relationship requires that G and H have the same transitive

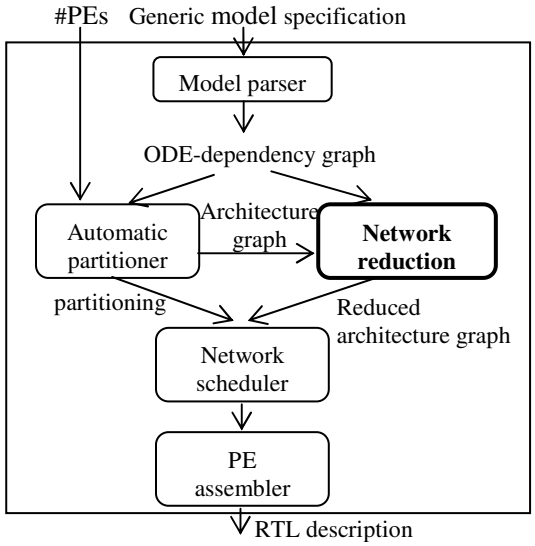

Figure 4-2: PE network compiler flow with additional network reduction phase.

closure, i.e., that for all edges $e = (u, v) \in E$ there exists a path from u to v. $Cost_{SIZE}$ and $Cost_{PERF}$ are metrics that give an estimate of the network performance in terms of architectural (circuit frequency) and performance (iteration cycle time) cost, respectively.

Figure 4-3 gives example representations of G and a reduction H. Both G and H use the same vertex set, but the edge set is different. H is considered a reduction of G because H has fewer wires (3 in H compared to 4 in G), and the same transitive closure. The two communication tasks b and c in H must be routed through intermediate nodes to reach their destination.

4.2.1. NETWORK SIZE COST METRIC

We first establish a metric that is a measure of the "implementability" of the network on a target FPGA device, i.e., the estimated clock frequency of the placed-and-routed network. One primary factor of network size is the total number of wires, *num_wires*.

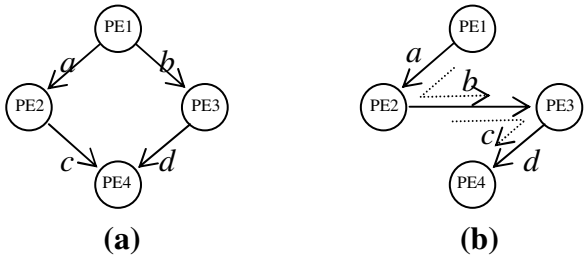

Figure 4-3: Reducing an architecture graph *G* to reduced architecture graph *H*. (a) *G* with 4 PEs and 4 wires. (b) *H* with fewer wires than G, but requiring multiplexing *b*,*c*.

However, the number of wires alone is not a good indicator of clock frequency. Networks that are very dense, having a high ratio of edges to nodes, may not yield high clock frequencies due to congestion during place-and route.

Table 4-2 shows a variety of networks and corresponding clock frequencies for increasing levels of network density. Such networks might be present as subcomponents of a larger design containing hundreds of PEs. Even for networks consisting of only 20 PEs, substantial timing penalties begin to accrue as the density increases.

Consider the graph in Figure 4-4(a), which plots final circuit frequency after place-

# PEs	# Wires	Density (#wires/#PEs)	Freq (MHz)
5	20	4.0	299
10	90	9.0	284
15	210	14.0	265
20	200	10.0	270
20	300	15.0	256
20	380	19.0	188
25	592	23.7	0
30	847	28.2	0

Table 4-2: Number of PEs and wires alone do not predict clock frequency.

and-route against the number of wires in the network for a variety of different models. In fact, if a network is particularly dense, than circuit performance can drop dramatically as shown (circled points). There are cases in which particularly dense designs consisting of 500-1000 wires can not complete place-and-route, while other designs containing 1000-1500 wires achieve 150-200 MHz clock frequencies.

To accurately estimate clock frequency the *densest-at-least-k-subgraph* problem, denoted *d(k)*, is solved for a given reduced architecture graph *H*. *d(k)* is the density of the most dense subgraph in *H* with at least *k* vertices. *k* is a parameter chosen by considering the size and architecture of the target FPGA. Choosing higher values of *k* filters out dense subgraphs with small numbers of PEs, which is useful because smaller subgraphs have minimal impact on frequency and may be ignored. Note that the densest-at-least-k-subgraph problem is NP-complete; we compute an approximation whose algorithm requires *O(m)* time for the unweighted ODE dependency graphs [4].

A function D(k) is defined that represents the overall density of a given network.

$$D(k) = \begin{cases} 1, & d(k) < d_{\min} \\ C_{FPGA} * \exp\left(\frac{d(k)}{d_{\min}}\right), & otherwise \end{cases} \quad (1)$$

d_{min} is a parameter that sets a limit on the minimum allowable density before circuit performance is impacted. If the densest-at-least-k subgraph found by *d(k)* is less than d_{min}, than *D(k)* evaluates to 1. However, if $d(k) \geq d_{min}$ then the cost is scaled by an exponential factor $d(k)/ d_{min}$. C_{FPGA} is a user-selected parameter chosen based on the size

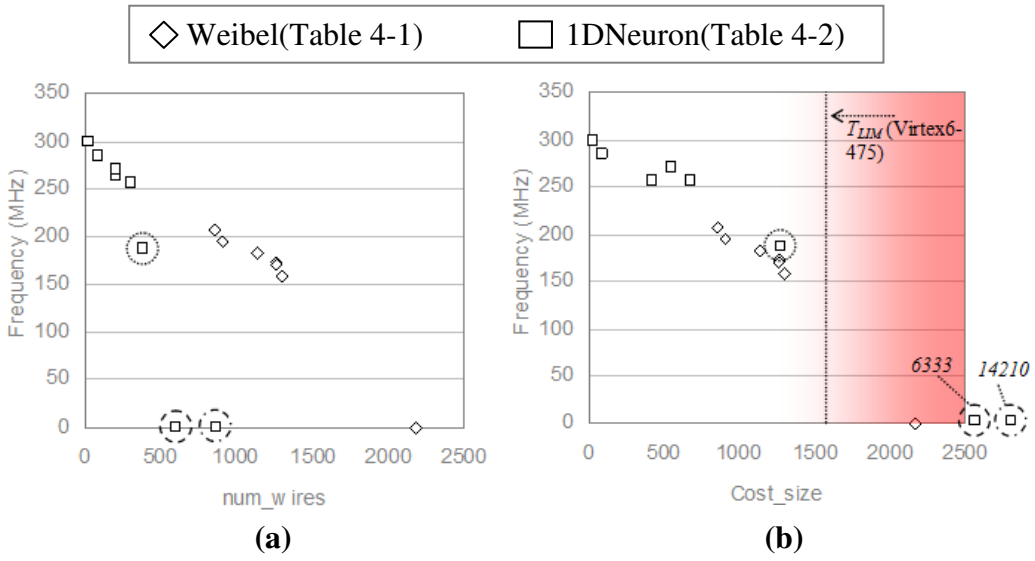

Figure 4-4: Wires alone do not predict frequency well; (a) note circled points with low wire count but low frequency. (b) The proposed cost function is a better predictor.

and architecture of the target FPGA. We use C_{FPGA} = 0.5 for a Xilinx Vertex6-475T FPGA; when targeting smaller devices C_{FPGA} should be increased, since a smaller device can less effectively implement a dense design compared to a larger device, and vice-versa. We use K_{FPGA} = 15 and d_{min} = 10 throughout this paper

Using the above density function $D(k)$, the cost metric $\text{Cost}_{\text{SIZE}}$ can be defined.

$$Cost_{SIZE} = num\_wires * D(K_{FPGA}) \qquad (2)$$

A high value of $Cost_{SIZE}$ corresponds to a more complex architecture and a high amount of congestion. A low value corresponds to an architecture easily placed on the target FPGA. Figure 4-4(b) shows the correlation between $Cost_{SIZE}$ and final circuit frequency using the results from Table 4-1, Table 4-2, and additional PE networks synthesized implementing matrix multiplication applications consisting of 100-500 PEs.

There is a clear limit to the complexity of a design that can be supported on any given platform, denoted as T_{LIM}. For a Xilinx Vertex6-475T FPGA, T_{LIM} = 1500. T_{LIM} may be reached either through sheer network size and the number of wires, or if a network contains dense subgraphs of nontrivial size. Ideally T_{LIM} is set to a value that represents the maximum size cost that can still be implemented, albeit at a low final circuit frequency. Any value greater than T_{LIM} would likely result in a failure during place-and-route.

4.2.2. NETWORK PERFORMANCE COST METRIC

The $Cost_{PERF}$ metric measures the performance of an architecture in terms of the number of communication cycles required per model iteration. Time-multiplexing multiple communications impacts the peak throughput of the network, since intermediate PEs must spend additional cycles to store and forward each intermediate communication. Thus, $Cost_{PERF}$ is the maximum number of communication cycles c_i required in any PE in the network.

$$Cost_{PERF} = \max(c_1, c_2, ..., c_n) \quad (3)$$

As an example, consider the number of communication cycles required for each graph in Figure 4-3. The communication of the original graph *G* can be optimally scheduled in 2 cycles. In the first cycle PE1 outputs *a*, PE2 stores *a* and outputs *c*, and PE4 stores *c*. In the second cycle PE1 outputs *b*, PE3 stores *b* and outputs *d*, and PE4 stores *d*. Thus, *G* has a $Cost_{PERF}$ value of 2. *H* is a reduction of *G*, having one fewer wire but requiring time-multiplexing of the middle physical channel. *H* can be scheduled in 3

cycles. In the first cycle, PE1 outputs *a*, PE2 stores *a* and outputs *c*, PE3 stores *c* and outputs *d*, and PE4 stores *d*. In the second cycle PE1 outputs *b*, PE2 stores *b*, PE3 outputs *c*, and PE4 stores *c*. In the third cycle PE2 outputs *b* and PE3 stores *b*. Thus, *H* has a $\text{Cost}_{\text{PERF}}$ value of 3.

4.2.3. COST FUNCTION

The goal of the optimization is to minimize $Cost_{SIZE} + Cost_{PERF}$. Since $Cost_{PERF}$ is small compared to $Cost_{SIZE}$, each term is adjusted to fit on a scale from 1-100 so that changes in each metric are relative to the actual impact on the final design throughput.

$Cost_{SIZE}$ is scaled according to the maximum allowable value for a given target T_{LIM}. For example, if a network has a value of $Cost_{SIZE}$ = 500, then the normalized value $NCost_{SIZE}$ is 500/1500 * 100 = 33.3 when targeting a Virtex6-475T. If $Cost_{SIZE} > T_{LIM}$, then $NCost_{SIZE}$ is set to the maximum 100 value.

$$NCost_{SIZE} = \min\left(\frac{\text{Cost}_{\text{SIZE}}}{T_{LIM}},1\right)*100 \qquad (4)$$

$Cost_{PERF}$ is scaled according to the overhead introduced by time-multiplexing. The normalized value $NCost_{PERF}$ should be 1 when there is no overhead, i.e. there has been no reduction of the graph. On the other hand, $NCost_{PERF}$ should be 100 when high amounts of overhead will severely limit the peak total throughput of the network. We select $NCost_{PERF}$ to be 100 when the number of cycles per iteration C_I for the reduced graph reaches 10x the required C_I for a non-reduced graph. Note that C_I includes both computation and communication cycles required per iteration so that schedules

dominated by computation cycles are not penalized by marginal increases in communication cycles.

$$NCost_{PERF} = \min\left(\frac{\text{Cost}_{\text{PERF}}}{10 * C_I}, 1\right) * 100 \quad (5)$$

Finally, a weighting parameter λ allows a designer to favor either size or performance. Setting λ to 0 only considers the communication overhead, setting to 1 considers only the size and complexity of the network, and fractional values in the range [0,1] balance the costs accordingly.

$$Cost = \lambda * NCost_{SIZE} + (\lambda - 1) NCost_{PERF} \quad (6)$$

4.2.4. REDUCING SYNCHMC WIRES

Now that a cost function has been established and the quality of reduced architecture graphs can be evaluated, we present a greedy approach for generating reductions of PE networks architectures. Starting with the architecture graph and partitioning generated by the PE network compiler, the architecture graph is iteratively reduced while maintaining the graph reachability. We investigated graph theoretic techniques referred to as *graph sparsification* [53] as a method for reducing PE network density. However, while sparsification techniques for undirected graphs are well known, similar techniques for directed graphs remain an open problem. As such, we have developed our own method of

eliminating edges that maintains important PE network properties like reachability by finding partial transitive reductions, as described in Section 4.2.5.

The *transitive reduction* of a directed acyclic graph *G* is the graph *H* with the fewest edges that has the same reachability as *G* [1]. The architecture graph generated by the PE network compiler and the transitive reduction of that graph represent the two boundaries of the solution space. The architecture graph represents a high size cost and low performance cost, since all communication is point-to-point. The transitive reduction represents a low size cost and high performance cost, due to time-multiplexing over the smallest possible number of channels. Our approach begins with the architecture graph as the initial solution, iteratively computing partial transitive reductions of the graph where each iteration is allowed to remove more wires. The size and performance cost of the resulting network after each iteration is evaluated, and the algorithm terminates when the cost is no longer decreasing.

Figure 4-5 gives an overview of the greedy algorithm, which consists of four main components within a do/while loop:

1. Compute a reduction of the original architecture graph *G*, given a maximum component size parameter that increases at each iteration (explained in detail in Section 4.2.5).

2. Schedule the communication tasks of *G* on reduced architecture *H*, yielding the maximum number of communication cycles required for any node in the network.

3. Compute the densest-at-least-K-subgraph of *H*.

4. Compute the total cost. Terminate the search if no improvement seen.

```
GREEDYSEARCH(G):
  Comp_MAX ← 1
  Cost ← ∞
  Last_cost ← cost
  do
    H ← NetworkReduce(G, Comp_MAX)
    comm_cycles ← Schedule(G, H)   // Schedule communication tasks of G on reduced architecture H.
    density ← Ds(H, K_FPGA)  // Find densest-at-least-K_FPGA-subgraph of H

    Cost_SIZE ← min((H.edges * density) / T_LIM, 1) * 100
    Cost_PERF ← min(comm_cycles / (C_I*10), 1) * 100

    Last_cost ← Cost
    Cost ← λ*Cost_SIZE + (1- λ)*Cost_PERF

    Comp_MAX += Δ // Increase max component size reduces wires in H next iteration
  while Last_cost > Cost

  Return H
```

Figure 4-5: Greedy algorithm for finding a reduction of graph G

Note that scheduling the time-multiplexing is a critical process that can have large impact on network performance - Chapter 5 discusses scheduling in more detail.

4.2.5. FINDING REDUCTIONS

A transitive reduction of a directed graph *G* containing cycles can be found by first computing the *condensation* of *G*. A condensation is created by contracting the strongly connected components of the graph into a single vertex, and then adding edges between the condensation vertices if an edge exists between the strongly connected components in the original graph. Once the condensation of *G* has been found, then a Hamiltonian cycle (possibly consisting of both original and newly added edges) in each strongly connected component is identified and wires not in this cycle can be removed. Thus, the resulting

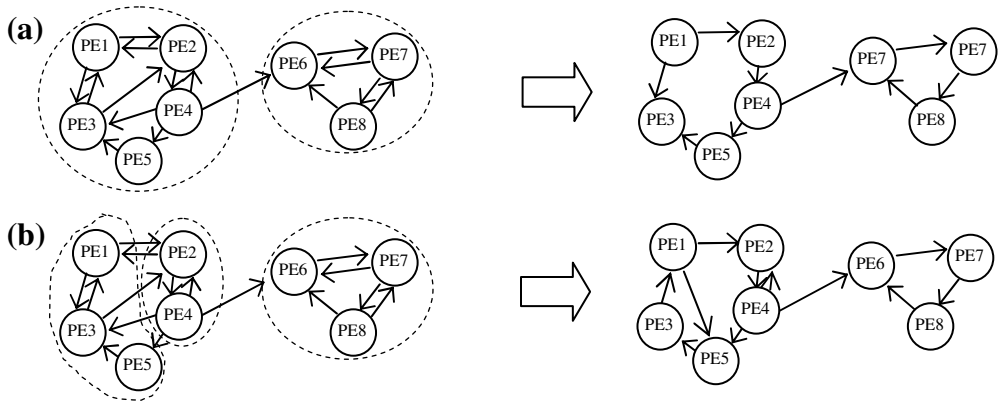

Figure 4-6: Reducing network wires. (a) Computing *H* via condensation of *G*; strongly connected components circled. (b) Limiting component size to 3 gives fewer wires but requires more cycles.

graph has a minimal number of wires in each strongly connected component, and no more than δ edges exists between any two components in a given direction. δ is a user-selected parameter than defines a minimum graph cut-size.

Figure 4-6(a) shows an example of finding the transitive reduction by first finding the condensation. Two strongly connected components exist in the graph, each node in a component being able to reach any other node in the same component. The condensation replaces the edges in each component with a Hamiltonian cycle; reachability in the component remains unchanged, but now the maximum latency for a communication is the size of the component minus one. For example, for PE2 to communicate with PE1, 4 hops are required to PE4, PE5, PE3, and finally PE1.

Of course, computing the transitive reduction yields a graph with the minimum number of wires. The solution space between the original graph and the transitive reduction is of the most interest. Searching this space is enabled by introducing a parameter $Comp_{MAX}$ that sets the maximum size of a strongly connected component. Components larger than $Comp_{MAX}$ are partitioned into separate components.

Figure 4-6(b) shows the effect of setting $Comp_{MAX}$ to 3 for the given graph. The left component, originally consisting of 5 nodes, is split into two separate components. The cycles are constructed by linking a series of nodes together, and condensation edges are placed between components. Setting $Comp_{MAX}$ to higher values results in graphs with fewer edges, while setting $Comp_{MAX}$ to low values yields graphs with more edges; setting $Comp_{MAX}$ to 1 gives the original graph with no changes.

Note that finding Hamiltonian cycles in a graph is NP-Complete. However for this application, it is not necessary to select the cycle within each component using existing edges. That is, every edge in a component can be deleted, and a cycle can be formed simply by creating new edges to link each node together, as shown. Thus, with the caveat that arbitrary edges can be placed to form a cycle, the problem becomes more tractable and expensive heuristics to find existing cycles are avoided. The order of the nodes chosen currently is random, but some improvement may be possible by keeping as many original edges of the graph as possible, reducing the overall amount of hops required during an update phase.

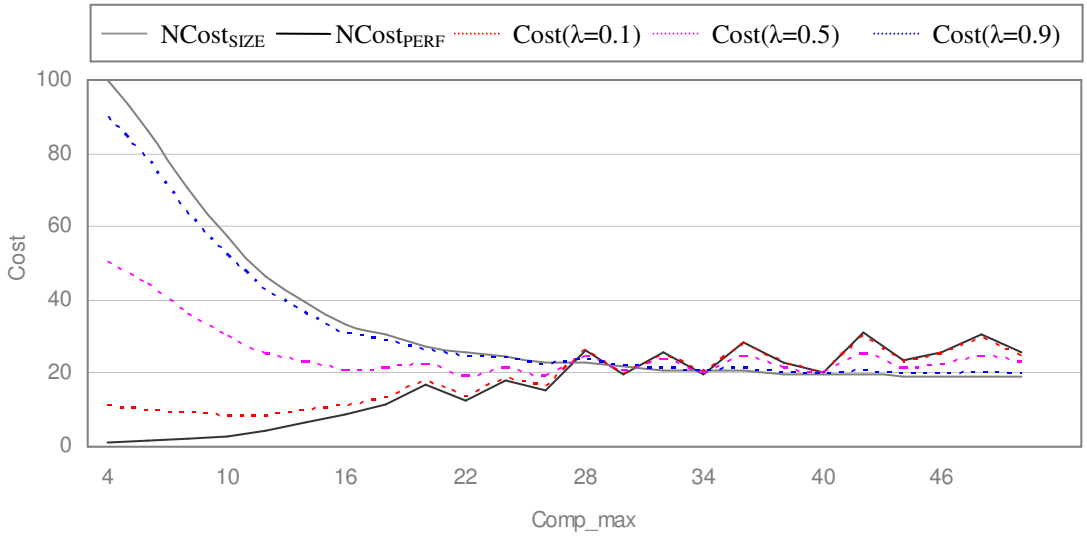

Figure 4-7: Cost of a PE network design as a function of $Comp_{MAX}$.

λ	$NCost_{SIZE}$	$NCost_{PERF}$	Freq (MHz)	ODEs / sec x 10^9
0.1	57.1	2.8	130	1.62
0.2	39.4	6.3	148	1.61
0.3	33.1	8.4	200	1.92
0.4	33.1	8.5	200	1.91
0.5	33.1	8.4	200	1.92
0.6	30.1	11.4	215	2.06
0.7	25.9	12.6	209	1.64
0.8	25.9	12.7	209	1.63
0.9	23.1	15.17	213	1.66

Table 4-3: Effect of λ on network throughput.

To avoid bottlenecks, maintaining a minimum cut-size between any two components is important to overall network performance. A parameter α can be set that limits the maximum number of edges between components. We use α = 2 throughout this work. Once condensation edges have been identified, a cycle is built in each component in *H* by linking nodes together. On an average case, the distance of a hop is approximately $Comp_{MAX}$ / 2, since a communication may be the next node in the cycle in the best case, or may be the last reachable node in another component in the worst case.

Figure 4-7 shows the effect of $Comp_{MAX}$ on $Cost_{SIZE}$ and $Cost_{PERF}$ for various settings of the weighting coefficient λ.

Table 4-3 shows the throughput of a PE network implementing an 11-generation Weibel lung model with 256 PEs and ρ = 1. Best network throughput is achieved when λ is around 0.5 - 0.6, such that the size and performance weightings are balanced.

4.3 COMPARING SYNCHMCS WITH TIME-MULTIPLEXING SYNCHMCS

The key benefit of introducing the reduction phase and time-multiplexing communication is that models of arbitrary size and complexity can complete place-and-route on target FPGAs. Table 4-4 gives place-and-route results for four Weibel lung models of varying complexity, along with the corresponding peak possible computation rates for each implementation. Without a reduction phase, the more complicated models quickly begin to produce designs that are too congested for the target Virtex-6 FPGA, and can not be implemented. Using a reduction phase allows even the largest of models to be implemented successfully.

Smaller models that can fit on the target device likely will have better performance if a reduction phase is not used. Introducing time-multiplexing communication may lengthen the schedule such that the improved frequency of the reduced design does not compensate for the extra cycles. In Table 4-4 the throughput of routable designs is reduced approximately 50% in each case. However, if the synthesized model is only a

		Without reduction phase (ρ=1.0)				With reduction phase (λ=0.5, ρ=1.0)			
Model	# PE	# wires	# cycles	Freq (MHz)	ODEs/sec x10^9	# wires	# cycles	Freq (MHz)	ODEs/sec x10^9
Weibel10 (2048 ODEs)	250	784	265	208	3.21*	431	444	214	1.97*
	500	1006	154	223	5.92*	817	344	190	2.27*
Weibel11 (4096 ODEs)	250	1262	495	--	**0**	382	927	214	**1.89**
	500	1325	271	162	4.89*	770	569	192	2.76*
Weibel12 (8192 ODEs)	250	3468	982	--	**0**	382	2750	174	**1.03**
	500	1651	502	--	**0**	836	1338	174	**2.47**
Weibel13 (16384 ODEs)	250	7209	1969	--	**0**	522	5378	140	**0.85**
	500	3553	975	--	**0**	656	2442	154	**2.06**
Atrial30x30x30 (27,000 ODEs)	500	5984	1500	--	**0**	842	6935	103	**0.60**
Neuron100x100 (30,000 ODEs)	500	13559	1171	--	**0**	1052	5175	137	**0.80**

Table 4-4: Reduction phase enables synthesis of previously unsynthesizable designs.

*A reduction phase reduces throughput of a design because of time-multiplexed communication, but can increase clock frequency. Smaller designs can achieve better performance by not using a reduction phase.

part of a design destined for the target FPGA, i.e. if including soft-core control processor or I/O logic, then the increased clock frequency may be desired to meet timing requirements of other components. In such cases, a designer likely would set a minimum requirement on network performance, and then reduce the network while the performance requirement is met, such that clock frequency is maximized.

Note that $\lambda = 0.5$ is used as the weighting coefficient for each model. For smaller models like Weibel10, a lower setting of λ would likely produce designs with higher peak throughput, since removing wires is not as important as reducing the schedule length.

The targeted platform is a Xilinx Virtex6-475T FPGA; the circuits are implemented using Xilinx ISE 14.2. ISE is run in Performance Evaluation mode, which does non-timing driven implementation. Performance evaluation mode is used to approximate

circuit performance; generally about 10-15% performance increases in clock frequency could be achieved by re-running place-and-route using the performance evaluation result as a timing constraint [62], though such benefits would apply to both the reduced and non-reduced designs.

4.4 COMPARING TIME-MULTIPLEXING SYNCHMCS WITH GENERAL PURPOSE CPUS

Targeting FPGA platforms is worthwhile only if the speedup over more traditional platforms is substantial. Previous work showed general PE networks could achieve 15x speedup in performance over a 3 GHz Intel CPU [22]. Custom PE networks, wherein the processing elements are custom pipelined data paths instead of general-purpose-processors, were shown to achieve 4-9x performance increases over the general PE approach, 9x over high-level synthesis, 26x over GPU, more than 100X over a 6-core DSP, and 24x over a 4-core CPU.

The past results investigated models whose size was amenable to fitting on the target FPGA. In this work, we evaluate very large models that must introduce a time-multiplexing scheme so that place-and-route of the network is possible. Figure 4-8 compares the performance of the three previously described large models, consisting of 16,000, 27,000, and 30,000 variables respectively. The graph shows the time to execute a single iteration of the model - lower time indicates faster execution speeds.

CPU results are given for 1 and 4 threads. The CPU used was a quad core Intel i7-3770, clocked at 3.4 GHz. The code was compiled using GCC 3.4.4 and full

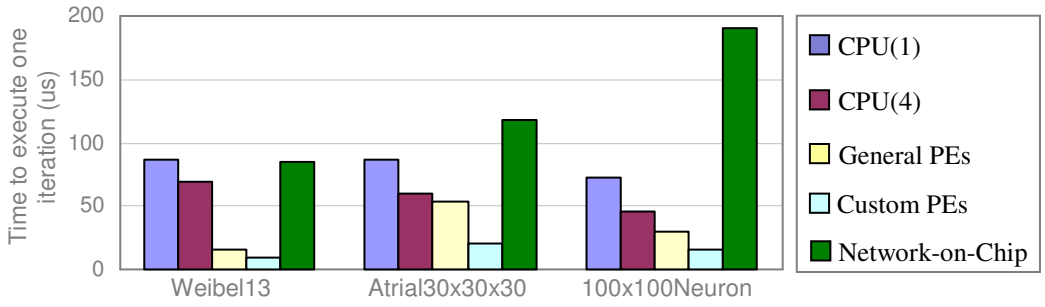

Figure 4-8: Time to execute one iteration of a model. General PEs are 3x faster than CPU(1), and custom PEs are 6x faster than CPU(1).

optimizations (-O3). We wrote custom kernels to solve each model, and parallelized the programs by partitioning the lung branches / neurons / atrial cells among the available threads. Each model is executed one second of real time, and then hardware performance counters are queried to obtain performance results. Adding additional threads to the CPU implementation did not result in additional performance gain, since thread management overhead requires synchronization after each iteration and limits possible performance gains.

On average for the given three large models, general and custom PE networks achieve 3.2x and 6.2x better performance than the single-core CPU implementation, respectively. Although custom PE networks previously were shown to achieve over a order of magnitude speedup over even general PEs, here their benefits are less pronounced (but still 2x faster than general PEs). The schedule of large models is dominated by the communication update phase, instead of the compute phase, thus the faster-computing custom PE networks have diminishing returns.

4.5 COMPARING WITH NETWORK-ON-CHIP

Network-on-Chip (NoC) architectures have been shown to be an effective and scalable platform for implementing multi-core system-on-chip designs [30]. In this section we compare time-multiplexing PE networks to the Configurable NEtwork Creation Tool (CONNECT) [47]. CONNECT is a highly parameterized NoC RTL generator that specifically targets FPGA architectures. The generated NoC can be customized to fit a given application by specifying the number of endpoints, network topology, pipelining, packet size, and so on. Below, we describe the parameters used to generate NoC architectures for comparison to PE networks.

- *Topology* - Structure of the network. We evaluate a subset of the possible topologies: double-ring, fat tree, and 2D mesh.
- *Number of endpoints* - How many PEs can be connected to the NoC. This parameter is constrained by available FPGA resources; we evaluate NoCs that can support 16-64 PEs.
- *Number of VCs* - How many virtual channels: We use 2 VCs.
- *Flow Control Type* - How the PEs interface with the NoC. We use CONNECT's peek flow-control in lieu of credit-based control flow.
- *Flit Data Width* - Size of a flit. Each packet consists of a single flit of size 42, not including header information. 32 bits is data, and 10 bits is used to identify the variable.
- *Flit Buffer Width* - Depth of buffers in flits. We use 64-flit size buffers.

- *Allocator* - Router allocation algorithm: We use separable input-first round-robin.
- *Pipeline Router* - The router core is pipelined to increase clock frequency.

The traffic pattern of synchronized PE networks is different from typical applications that target NoC architectures. Typically network traffic is uniformly distributed in terms of flits/cycle, with minor variations depending on the application. However, due to the 2-phase update/compute synchronization, PE networks have spikes of high loads during an update phase, followed by no load during the compute phase. Our NoC architectures must therefore be able to operate under high load without becoming saturated. To account for the load spikes, the CONNECT *Flit Buffer Width* parameter is set to its maximum value of 64, and packet injection during an update phase is limited to avoid saturation.

To test the performance of a generated NoC we first use the PE network compiler to partition an application into a set of PEs, and then schedule the global communication between PEs. The global communication tasks are translated into packets to be routed by

			Weibel13		*Atrial30x30x30*		*2DNeuron100x100*	
Topology	# PEs	Freq (MHz)	# cycles	uSec / iter	# cycles	uSec / iter	# cycles	uSec / iter
DoubleRing	32	182	19789	109	31613	174	49938	274
DoubleRing	64	135	14788	110	23112	172	39783	296
Mesh	32	107	20117	189	26388	247	19310	369
Mesh	64	126	10734	85	14840	118	24129	191
FatTree	32	80	19275	242	24767	311	42619	535

Table 4-5: NoC performance is limited by the number of available endpoints. The 2D mesh topology with 64 endpoints yields the best performance.

a NoC. A NoC architecture is simulated on a cycle-accurate simulator (Xilinx ISim). Each PE attempts to inject a packet into the NoC on each cycle, until all the packets have been routed. To avoid saturating the network, we found that idle cycles needed to be introduced between packet injections. The number of idle cycles depends on the number of packets and network architecture, and ranges from 3 to 17 cycles. The NoC architectures are synthesized to obtain a clock frequency, and the time to compute an iteration of model is found as the product of the period and total schedule length. The schedule length is the number of computation cycles and number of cycles to route all packets.

Chapter 5. SCHEDULING TIME-MULTIPLEXED SYNCHMCS

Time-multiplexing introduces new challenges to scheduling the update phase of a PE network. Every communication task of the original PE network architecture graph must be scheduled on the newly generated reduced architecture graph, using intermediate PEs to store and forward non-direct messages. We investigate three different approaches to scheduling: a shortest-path greedy scheduler, a heuristic based on path lengths, and an integer linear program.

The scheduling problem has the following definitions and constraints:

1. *Operations per cycle* — Each PE can execute at most one storing action per cycle, and at most one forwarding action per cycle.
2. *Intermediate local storage* — A PE can store intermediate values in local memory in cases where a higher priority message must be handled, such as forwarding a message with a long path length.
3. *Objective* — The optimal schedule minimizes the maximum number of communication cycles required for any given PE (including idle cycles).

Related work by Kapre investigated scheduling of packet-switched and time-multiplexed overlay networks on FPGAs [29]. The work utilizes a greedy algorithm for

```
GREEDYSCHEDULE(H, T):
  Insts[PE][cycle] ← { }
  foreach T_{a,b} in T
    P_{a,b} ← a = v_1, v_2, ..., v_k = b  // Shortest path from a to b
    cycle ← 0
    foreach edge (v_i, v_{i+1}) in P_{a,b}
      while (!AvailableForward(v_i, cycle) || !AvailableStore(v_{i+1}, cycle))
        cycle ← cycle + 1
      Insts [v_i][cycle] ← Forward(T_{a,b})
      Insts [v_{i+1}][cycle] ← Store(T_{a,b})
      cycle ← cycle + 1
```

Figure 5-1: Greedy scheduling algorithm

scheduling, although the authors note that a PathFinder implementation is also available. Although PathFinder is designed to be used as a routing algorithm, we hope to investigate its use as a heuristic for scheduling PE networks in future work.

5.1 GREEDY SCHEDULER

A simple approach is a greedy scheduler that loops over every message and routes that message on the shortest path from source to sink. Let T be the set of communication tasks that must be routed. Let $T_{a,b}$ be a communication task with source a and sink b. The shortest path from a to b is denoted $P_{a,b}$, and consists of an ordered set of vertices $a = v_1$, v_2, ..., $v_k = b$. Figure 5-1 gives the algorithm psuedocode. *Insts* is a data structure that stores the communication instructions for each PE at every cycle. In the inner loop, for any i, we find the first cycle for which message $T_{a,b}$ can be forwarded from v_i to v_{i+1}, that is the first cycle when v_i is not busy forwarding some other message and v_{i+1} is not busy storing some other message.

The Search() function can be any appropriate graph path-finding algorithm; we use A* search. Even for large graphs, the algorithm generally requires less than one second to

complete scheduling. However, the simplicity of the approach comes at the cost of less-than optimal schedules. Because the shortest path is always taken, alternative routes that are longer but less congested are not used. While the greedy algorithm can produce near-optimal schedules for simple and sparse networks, dense networks with high amounts of traffic yield poor scheduling results. As an example, a lung model with 2000 equations implemented on a 75-PE network is scheduled in 1550 cycles, which is about 3x more cycles than the optimal schedule.

5.2 MATCH-SCHEDULE

A better scheduler utilizes alternative paths from a to b when scheduling $T_{a,b}$, instead of always taking the shortest path. Given a reduced architecture graph H and set of tasks T, the presented algorithm iteratively schedules each cycle. The scheduling of each cycle is framed as an optimization problem, where a subset of active tasks can be scheduled (given the constraint of at most one store/forward operation per PE), and each task can choose from many possible paths through the network. The objective is to maximize the number of tasks scheduled in the cycle, while considering the relative importance of each task, quality of each path for every task, and PE congestion.

To schedule each cycle $l = 1,2,3,...$ a weighted bipartite graph J_l is first constructed. For reference, see Figure 5-2 (the bipartite graph of each cycle being bounded by the dashed vertical lines). The *current* vertices on the left-hand side of J_l are PEs that contain an active task to be scheduled in the current cycle l (set A_l). The *next* vertices on the right-

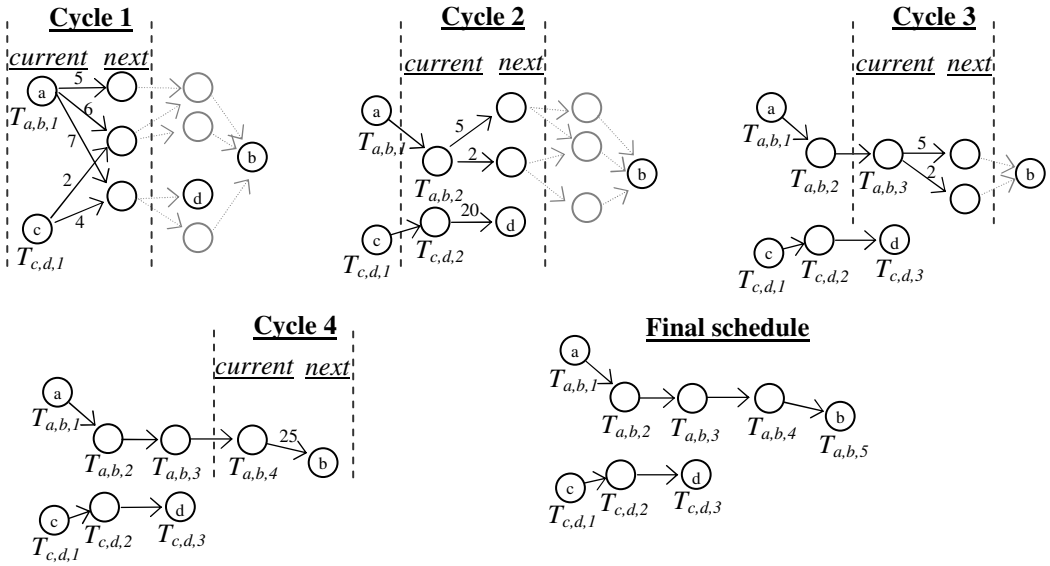

Figure 5-2: Scheduling a tasks $T_{a,b}$ and $T_{c,d}$ using Match-Schedule. Each cycle is scheduled via a maximum-weighted matching operation.

hand side of J_l are PEs for which a task $T_{a,b}$ currently in A_l can take to reach sink b (set B_l).

Let $T_{a,b,l} = x$ be the current PE of task $T_{a,b}$ at cycle l; let y be a PE in B_l. An edge (x, y, $T_{a,b}$) from x to y with label $T_{a,b}$ exists in set E_l if sink b can be reached from x through y. Multiple edges can exist from x to y for different tasks. The bipartite graph J_l is thus defined by the vertex and edge sets (A_l, B_l, E_l). To illustrate, cycle 1 of Figure 5-2 shows a task $T_{a,b}$ originating from PE a that can choose one of three neighbors of a on a path to sink b, and a task $T_{c,d}$ that can choose from two neighbors of c on a path to sink d. A_1 consists of PEs a and c, B_1 consists of three neighbors of a on a path to b and two neighbors of c on a path to d. E_1 consists of five edges between A_1 and B_1.

Weights are assigned to each edge in J_l according to a weighting function (described in the following section). The weighting function considers path length, source

congestion, and expected path delay. Once weights have been assigned to each edge, the maximum-weighted matching of J_l is computed. The maximum-weighted matching operation finds a set of edges of maximum total weight in which no two edges share an endpoint. Thus, the constraint of at most one store or forward operation per PE per cycle is met. In Figure 5-2 two paths $T_{a,b}$ and $T_{c,d}$ are scheduled; in cycle 1 the edge weights 6 and 4 yield the maximum matching weight of 10. The figure shows only an example with two tasks, for clarity. (Also, the weight values are selected by hand, not according to the formulas below.) In reality, in the applications we consider, the algorithm may schedule thousands of tasks in each cycle, and there is high contention between tasks for next hops.

The maximum matching operation is implemented by describing the problem as an integer linear program. Our implementation calls IBM's CPLEX 12.6 linear program solver to compute the optimal solution. The linear program is quite simple, requiring only constraints that limit the number of edges connected to any endpoint, while maximizing edge weights [13].

Figure 5-3 gives an overview of the Match-Schedule pseudocode. For any two PEs $x = T_{a,b,l}$ and y, the FindPath() function determines whether y is a neighbor of x, and whether y is on a path to b in H (in which case edge (x, y, $T_{a,b}$) will be created). FindPath() can be implemented via a BFS search originating from each y where edge (x, y) exists in H. Possible optimizations include only adding edges of paths that are not longer than the average distance of all paths from x to b, or using a backwards BFS originating from b to instead of multiple BFSs from each y.

```
MATCHSCHEDULE(H, T):
  l ← 0  // current cycle
  while not all tasks T_{a,b} scheduled
    A_l ← set of all PEs with pending tasks in cycle l
    B_l ← all PEs
    foreach x in A_l
      foreach task T_{a,b}: T_{a,b,l} = x
        foreach y in B_l
          if path x→y→...→b exists in FindPath(x, y, b, H)
            add_edge((x, y, T_{a,b}), E_l)  // Add edge (x, y) with label T_{a,b} to set E_l

    J_l ← (A_l, B_l, E_l)          // Create bipartite graph

    foreach edge e = (x, y, T_{a,b}) in E_l
      e.weight = α*U(T_{a,b}) + β*E(x, y, T_{a,b}) + γ*C(x)

    foreach edge e = (x, y, T_{a,b}) in MaxMatch(J_l)
      x.ScheduleForward(T_{a,b})
      y.ScheduleStore(T_{a,b})

    l ← l + 1
```

Figure 5-3: Match-Schedule pseudocode.

The weighted bipartite graph for each cycle l i*s* constructed by finding A_l and B_l, then adding edges where paths for each task exist from current PE $T_{a,b,l} = x \in A_l$ to b through $y \in B_l$. Edges are weighted according to the weighting function described below. Then the maximum matching is found and the result is used to schedule forward operations for PEs in A_l, and store operations for PEs in B_l in the current cycle.

The weighting of each edge $(x, y, T_{a,b})$ considers three factors: *urgency*, *estimated delay*, and *source congestion*. *Urgency* represents the global importance of task $T_{a,b}$ compared to all other tasks. Tasks whose minimum distance to the sink b is large have higher urgency and should be scheduled first. The urgency of edge $(x, y, T_{a,b})$ depends only on x and b. Every edge has *estimated delay* that represents a bound on the delay required to reach sink b via y. Paths with smaller estimated delay are favored. Finally, for

all x in A_l, the *source congestion* represents the number of messages currently waiting in x to be forwarded. PEs with higher amounts of congestion should be favored so that no PE becomes a bottleneck. All three factors are represented by values between 0 and 1, which are then weighted to obtain the final weight of edge (x, y, $T_{a,b}$).

Urgency can be calculated as the shortest path length from x to sink b in H. Normalizing task urgency to the maximum urgency of all tasks yields a value in the interval [0,1]. Let $D_{a,b}$ represent the shortest distance from $T_{a,b,l} = x$ to b (through any hop y). Then the urgency of $T_{a,b}$ is calculated as:

$$U\left(T_{a,b}\right) = \frac{D_{a,b}}{\max_{u,v}\left(D_{u,v}\right)} \tag{7}$$

The estimated delay of edge (x, y, $T_{a,b}$) is determined based on the path length from y to b. Let $D_{a,b,y}$ represent the minimum distance in H from $T_{a,b,l} = x$ to b through hop y. We need to convert $D_{a,b,y}$ into a value in [0,1] that is decreasing with respect to $D_{a,b,y}$, so that shorter paths are given higher priority. To this end, we first replace $D_{a,b,y}$ by $D'_{a,b,y} = D_{a,b,y} - \min_h(D_{a,b,h})$, where the minimum is over all possible hops h in B_l, and then the priority value representing estimated delay is computed as

$$E\left(x, y, T_{a,b}\right) = \delta^{-D'_{a,b,y}} \tag{8}$$

The value of δ controls the strength of weighting paths of different lengths. For example, if a task's shortest path has length 2 and the longest path has length 10, then a δ value of 2.0 would give the shortest path E = 1, and the longest path E = ~0.004. If δ is closer to 1, for example 1.1, then for the same length paths the shortest path has E = 1,

and the longest path has E = ~.47. We have found experimentally that a δ value of 1.5 works well.

The source congestion of a PE x is the number of active tasks waiting to be forwarded in x, that is the number of tasks $T_{a,b}$ for which $T_{a,b,l} = x$. The corresponding priority value $C(x)$ is computed by normalizing the congestion of x to the maximum congestion for all PEs in A_l, to give a value in the interval [0,1].

A coefficient for each of the above factors allows tuning the relative importance of each term. The final weight is calculated as:

$$W(x, y, T_{a,b}) = \alpha * U(T_{a,b}) + \beta * E(x, y, T_{a,b}) + \gamma * C(x) \qquad (11)$$

Experimentally we have found that the parameter values for α, β, γ of 0.75, 1.0, and 0.25 respectively give good results, although the network architecture and traffic characteristics can influence the ideal parameter settings.

5.3 INTEGER LINEAR PROGRAM SCHEDULER

We have developed a system of integer linear program (ILP) constraints that can generate an optimal schedule of a PE network, given that the reduced architecture and the paths for each task are provided as input. The ILP can be solved optimally for small networks using CPLEX. However, ILPs for medium to large-sized networks, i.e. networks with more than approximately one hundred PEs, more than a few hundred paths, or requiring more than a few hundred cycles to schedule, can not be solved because they are too large (having more than a few millions of variables and constraints).

We found that such ILPs could not be solved on an Intel i7-based desktop PC, as the available 8 GB of RAM was insufficient. Thus, while the ILP approach can not scale to producing optimal solutions for networks that are the most interesting (large and dense), we can use the ILP approach as a baseline indicator of the greedy scheduler's and Match-Schedule's performance on smaller networks.

Our ILP uses 0-1 variables $X_{P,a,b,l}$, where P is a path, (a, b) is an edge in path P, and l is a cycle. The variable $X_{P,a,b,l}$ indicates whether edge (a, b) in path P is used in cycle l. For each cycle l, we also have variable y_l that indicates whether cycle l is used in any path.

Using these variables, a natural objective function would be to minimize $\sum_l y_l$, the quantity that represents the number of cycles used in the schedule. We found, however, that with this objective function, ILPs for networks requiring a hundred or more cycles to schedule could not be solved in a reasonable amount of time (say, within 24 hours). As such, the above objective is useful only for very small networks. Instead, we use an alternative approach, where we set a maximum number of cycles L and check whether or not *any* solution with at most L cycles can be found that satisfies the constraints of the ILP (given below). A binary search approach can then be used to find the optimal schedule length, decreasing the number of cycles L when the ILP is feasible, and increasing L when the ILP infeasible. In this case, we do not need any objective function in our ILP.

We now list the constraints of the ILP. The variable l representing a cycle number is in the range [0, L]. The first constraint enforces that y_l is 1 if cycle l is used. To this end, for all $l = 0,...,L$, for all paths P, and for all edges $(a,b) \in P$, we have inequalities:

$$y_l \geq X_{P,a,b,l} \tag{12}$$

The system must be constrained such that each hop of a path occurs in the correct order in the schedule. For example, if a path is $a \rightarrow b \rightarrow c$, then $a \rightarrow b$ must be used before $b \rightarrow c$. To ensure this, for all $l = 0,...,L-1$, for all paths P, and for all pairs of consecutive edges (a,b), $(b,c) \in P$, we have inequalities:

$$\sum_{i=0}^{l} X_{P,a,b,i} \geq \sum_{i=0}^{l+1} X_{P,b,c,i} \tag{13}$$

Two or more paths cannot use the same edge on the same cycle. Thus, for all $l = 0,...,L$, and for all pairs of tasks P and Q, and for all edges $(a,b) \in P \cap Q$, we include inequalities:

$$X_{P,a,b,l} + X_{Q,a,b,l} \leq 1 \tag{14}$$

For every cycle, each PE can only store or forward at most one communication task, that is only one incoming edge and one outgoing edge for any PE may be used. Thus, for all $l = 0,...,L$, for all paths P, and for all PEs v, we have:

$$\sum_{P,a:(a,v)\in P} X_{P,a,v,l} \leq 1 \tag{15}$$

$$\sum_{P,b:(v,b)\in P} X_{P,v,b,l} \leq 1 \tag{16}$$

Finally, to drive the system to produce a nonzero solution, each edge in every path must actually be taken at some cycle *l*. Thus, for all paths *P*, and for all edges $(a,b) \in P$, we have constraints:

$$\sum_{i=0}^{L} X_{P,a,b,i} = 1 \tag{17}$$

Alternatively, to lower the number of constraints, such a constraint may be given only for the final edge in every path.

When compiling a PE network, OPL code describing the above linear system program is automatically generated. This code is passed to IBM CPLEX Optimizer 12.6 and solved to generate a schedule. The maximum number of cycles *L* has a drastic impact on program size -- we have found the simplest approach is to allocate as many cycles as the greedy scheduler requires initially. If performance is too slow then *L* is reduced until a solution can be found or the program is deemed to be infeasible by CPLEX.

5.4 SCHEDULING EVALUATION

Figure 5-4 shows the number of communication cycles required for a variety of physical models, for each of the greedy, Match-Schedule, and ILP scheduling

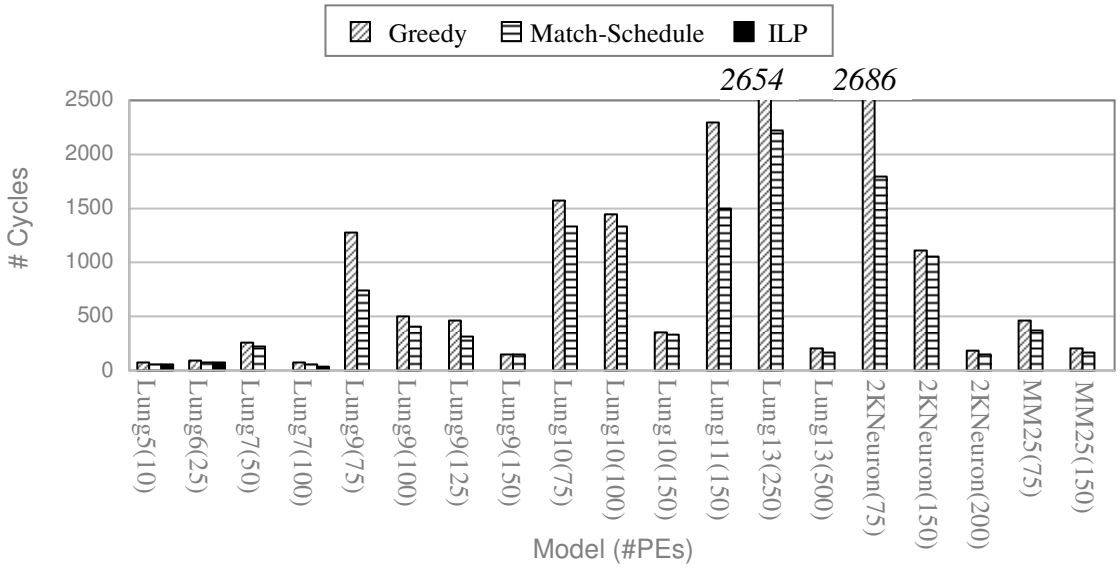

Figure 5-4: Number of cycles required by each scheduling approach.

approaches. The models were selected to represent a wide range of possible complexity and network densities. The lung models are a binary tree-structured lung model, MM25 is a 25x25 matrix multiplication application, and 2KNeuron simulates a 2000 element neural network. On average, Match-Schedule generates a schedule with 25% fewer cycles than the greedy algorithm for the given examples with high network density.

When considering only designs whose ILP solution could be computed, Match-Schedule requires 9% more cycles than the ILP solution on average. Note that the ILP approach can only schedule the first few models, which have a low number of variables and require less than a hundred cycles. The ILP solutions use the paths generated from Match-Schedule as input - there may be alternative paths that could yield a better schedule. Thus the ILP solutions should not be considered globally optimal for the given reduced architecture graph.

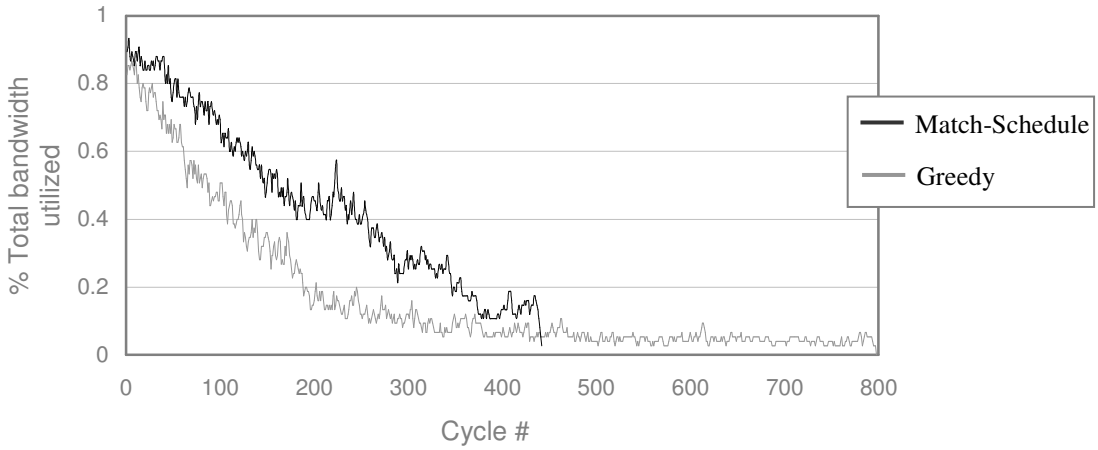

Figure 5-5: Match-schedule better utilizes bandwidth early in a schedule.

Figure 5-5 helps to illustrate why Match-Schedule performs better than greedy by illustrating the percentage of total possible bandwidth being utilized each cycle. Allowing alternative paths allows more of the existing channels to be utilized per cycle, thus allowing progress to be made despite high amounts of congestion early in schedule.

Chapter 6. EXPLORING SYNCMC CONFIG. SPACE WHEN EMULATING PHYSICAL MODELS

Physical models capture environmental phenomena such as biochemical reactions, a beating heart, or neuron synapses, using mathematical equations. synchMCs can execute orders of magnitude faster on FPGAs (Field-Programmable Gate Arrays) compared to desktop PCs. Different models of the same physical phenomenon may vary, with more accurate ("upgraded") models being more accurate but using more FPGA area and having slower performance. We propose that design space exploration considering upgradable models can dramatically increase the useful design space. We present an analysis of the solution space for utilizing networks of processing-elements (PEs) on FPGAs to emulate physical models, implement a web-based frontend to a compiler and cycle-accurate simulator of PE networks to estimate solution metrics, and utilize design-of-experiments (DOE) statistical methods to identify Pareto points. By considering upgradeable models during the design space exploration of a human lung physical model, the solution space of possible speedup, area, and accuracy by 6X, 7.3X, and 1.5X, respectively, compared to evaluating a single model.

Previous work has applied traditional design space exploration, partitioning equations among different types and numbers of processing elements to achieve area and performance tradeoffs. However, physical systems provide a rather unique additional solution option. The same physical system can be modeled with different equations. Each

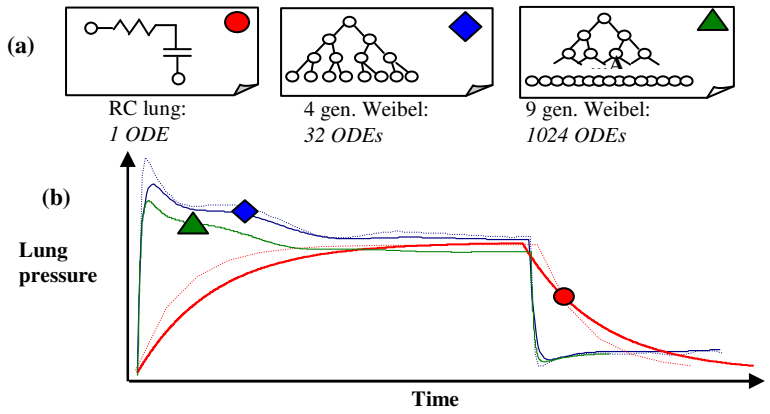

Figure 6-1: (a)A set of upgradeable models, and (b)relative accuracy of each model. Dashed lines show variations in accuracy from different solvers / step sizes.

model may have tradeoffs in terms of the number of equations, ease of computation, and accuracy. We denote sets of models that are functionally similar as *upgradeable* models, since a designer may 'upgrade' to a more accurate model at the expense of area and performance. For example, Figure 6-1(a) shows three models that capture the behavior of the same physical system of lung airway mechanics. A simple RC model can coarsely capture the behavior using a single ordinary differential equation (ODE). For higher accuracy, a binary-tree shaped Weibel model [61] with variable levels of complexity can be used at the expense of higher computational costs. Accuracy also depends on the step size and type of equation solver. A smaller step size yields higher accuracy but slower performance. Likewise, more accurate solvers yield slower performance. Figure 6-1(b) illustrates the accuracy of each model, where dashed lines represent some deviations in accuracy due to different step solvers or step sizes.

Upgradeable models substantially increase the solution space that must be explored, not only via expanding area and performance ranges, but also by adding the design metric of accuracy. Figure 6-1 shows various tradeoffs in terms of speedup, area, and accuracy for a set of upgradeable models.

Upgradable models introduce numerous additional parameters that influence and tremendously increase the design space. The influence on design metrics of those parameters can be complex and interdependent. To deal with these new parameters, we apply the statistical method known as design-of-experiments (DOE), which efficiently determines the impacts and dependencies of parameters, to enable efficient search of the design space. We introduce an approach that expands design space exploration to consider upgradeable models. To search the large design space, we utilize DOE statistical method to generate the Pareto points of the design. By generating the Pareto points, the design space can be pruned to enable a feasible exploration of solutions. We present a web-based tool that uses a processing-element (PE) network compiler and cycle-accurate simulator to automatically generate a PE network from an MML-language [38] based input model specification and evaluate the relevant size, performance, and accuracy metrics of PE network implementations. The web-based tool also supports automatic exploration of the design space using DOE techniques to aid in finding and appropriate model to use from an upgradeable set of models and an appropriate underlying PE network implementation that meets given constraints.

6.1 UPGRADEABLE MODELS

Upgradeable models in the context of physical systems emulation refers to having multiple underlying sets of equations that are each able to emulate the same physical model, with tradeoffs among accuracy versus area and performance. In this section, we define how to determine if relative models are part of the same upgradeable set, and discuss size-scalable upgradeable models.

6.1.1. FUNCTIONAL EQUIVALENCE OF MODELS

We consider different models to be a part of the same upgradeable set if they meet the following requirements:

1. The models contain the base input/output interface required to support the physical system behavior.

2. The models are functionally similar, i.e. they produce similar output for all possible inputs.

The first requirement ensures that all models can operate on the same inputs and can provide the same outputs. Physical model emulations are usually a part of a larger design, often for testing purposes, thus ensuring that all models in an upgradeable set have a similar interface ensures smooth transitions and reduces the potential to introduce new errors. Some small differences in interfaces may be acceptable, as long as a correct transformation is available. For example, a lung model may require either an air flow input or an air pressure input. Flow can be easily converted into air pressure, and vice versa, thus we may still consider the models to be functionally equivalent. Models may

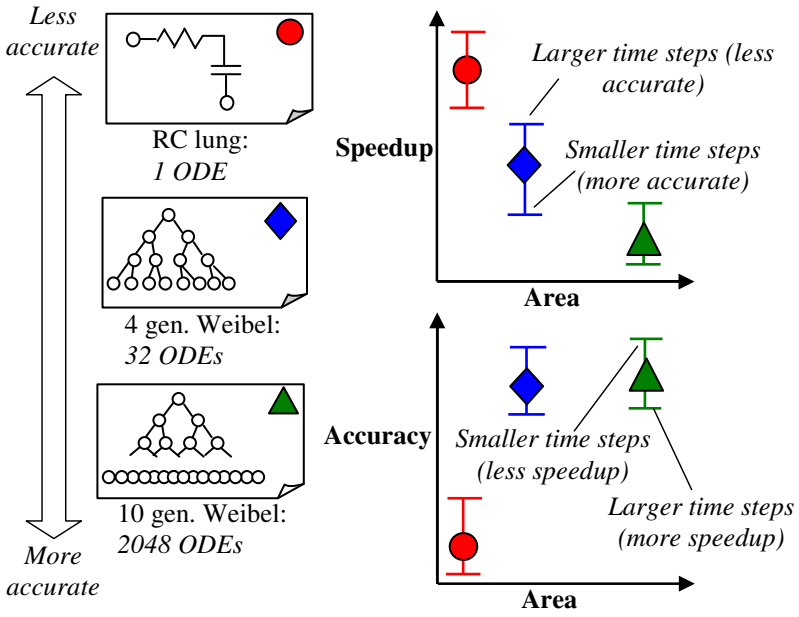

Figure 6-2: Effect of different models on area/speedup/accuracy metrics.

also provide a supplemental input/output interface, in addition to the required base interface. For example, the Weibel lung model of Figure 6-2(a) provides output pressures at each of the leaf nodes, whereas the RC lung model provides only a single output pressure node below the capacitor. The supplemental interface is not required, but may improve model accuracy or provide additional information about the internal model state to the designer.

The second requirement requires that all models in an upgradable set produce similar outputs for given inputs. This requirement ensures that the physical model being emulated is similar in functionality, despite any differences in the underlying equations that are computed. Similarity can be determined by both qualitative and quantitative methods. Figure 6-2(b) shows the output of three various lung models; the models are

considered interchangeable because they all produce an output of the same physical system, yet they have different quantitative and qualitative measures of accuracy. A designer can either determine that models are close enough to be functionally interchangeable, or a distance measure could be automatically calculated.

6.1.2. SCALING MODEL SIZE

Many physical moderns have a common, repeating pattern or structure. The previously introduced Weibel model has a binary tree structure, which resembles the 23 bifurcating branches of a human lung. Neuron or cell models can consist of hundreds or thousands of individual elements that are connected to neighboring elements in mesh or grid structures. Previous work has shown that physical model structure can even be utilized to aid placement of PE networks on FPGA fabrics [40]. Physical models with regular structures can be considered upgradeable if the physical model can be scaled in size by adding new elements into the structure.

Scaling a physical model may or may not affect the accuracy of the model, but certainly impacts the resulting area and performance of the implementation. The Weibel model can be scaled in size to have more or less tree generations - having more generations implies a higher level of accuracy because the number of branches is closer to actual lung physiology. However, doubling the number of cells in a cell tissue model doesn't necessarily imply that the equations of each individual cell are more accurate. Even so, a designer may want to know how many cells can be included, given some area or performance constraints.

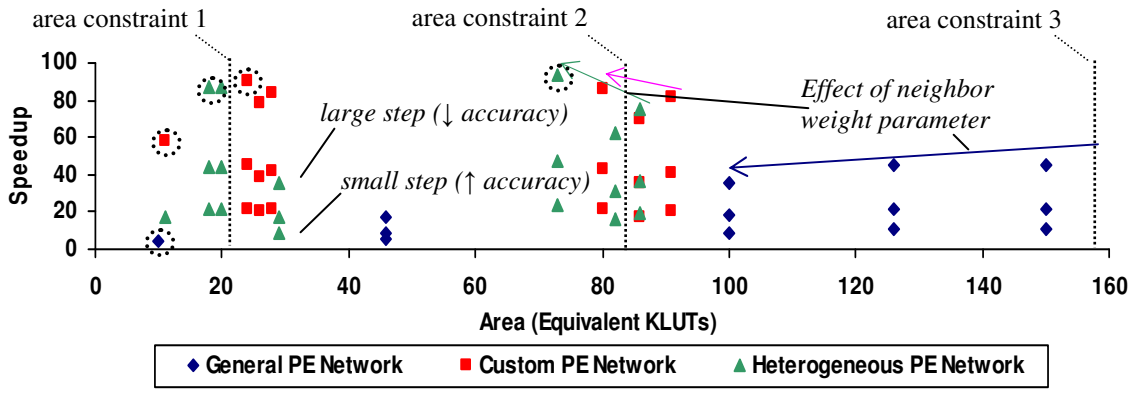

Figure 6-3: Possible configurations for a 300 ODE model. Three dashed lines indicate area constraints. Each solution shown with step sizes 1e-2, 0.5e-2, and 0.25e-2 ms with Euler solver. Dashed circles show possible area/performance Pareto points.

6.2 SYNCHMC PARAMETERS AND METRICS

For a given physical model of sufficient size and complexity, the solution space for a PE network that emulates the model is extremely large. This is due mostly to the parameters available during PE network synthesis, and partly to the non-deterministic heuristics used during equation partitioning. The considered key parameters are the model specification itself, PE network type, equation partitioning neighbor function weightings, the given resource constraints, step size, and solver type. The key solution metrics are FPGA area (LUTs, memory, and DSP usage), performance (speedup over real-time), and accuracy (closeness to exact solution).

Figure 6-3 depicts a chart of possible PE network solutions for a neuron model with 300 equations. Three sections are shown which depict solutions yielded by using different area constraints. For each area constraint, the PE network type, neighbor weight,

and step size parameters are varied. Area/speedup metric Pareto points are circled; the accuracy metric is not measured explicitly in the figure, though points towards the bottom typically use smaller time steps and thus would be more accurate.

The model itself is an important parameter. The PE network solutions that can be generated depend highly on how many equations are in the model, the complexity of each equation, and the data dependencies between the equations. The user also must specify a coefficient that quantitatively captures the quality of the model compared to the others in the set. For example, the 9 generation Weibel lung model may be considered to be the most accurate, and have a coefficient of 1. The RC model may be considered to have a coefficient of 0.4, because it only coarsely captures realistic behavior. The coefficients are used when comparing relative accuracies of different models.

PE type is a critical parameter that determines the type of PE network that is generated to emulate the model. There are three options: homogeneous general PE network, homogeneous custom PE network, and heterogeneous PE network. A general PE is a flexible, programmable, ALU-based processor that can solve any equation. A custom PE uses a pipelined datapath to solve a single specific equation much faster than a general PE, but may use more FPGA resources and incur higher routing congestion cost. Heterogeneous PE networks combine general and custom PEs to create a network with balanced performance and area metrics.

Neighbor weight refers to an option within the PE network compiler that controls whether the equation partitioning favors size or performance. This option is a sliding scale that can be set from 1 (favor size) to 10 (favor performance).

Area constraints detail the available LUTs, DSPs, and BRAMs on the target platform. The PE compiler will not allocate more than the available resources. The area constraint may have a very large area or performance impact on sufficiently complex models. An area constraint which is too small for the given model will not allow enough PEs to be allocated and reduce the possible speedup. Small models will typically not be affected highly by the area constraint parameter.

Solver type selects the iterative step solver to use. The currently supported solvers are Euler and Runge-Kutta4. Euler is the most simple solver, but can be inaccurate or diverge with medium to large time steps. The Runge-Kutta solver type is much more accurate, but may require up to 4X more computation time than the Euler solver.

Step size determines the amount of time between iterative solutions of the model equations. Decreasing the step size requires more computations per second, which reduces the performance of the model, but allows the solvers to be more accurate.

6.3 DOE-BASED EXPLORATION OF SYNCHMCS

To explore the space of PE network solutions for a set of upgradeable physical models, we have developed a visual web-based frontend coupled with a design-of-experiments statistical approach to identifying Pareto points that span an upgradeable set of physical models. In the following section we describe briefly what DOE is, and how our tool utilizes DOE.

Design-of-experiments is a statistical technique that identifies a minimal set of experiments that provide maximal cover of the possible solution space. Originally, DOE

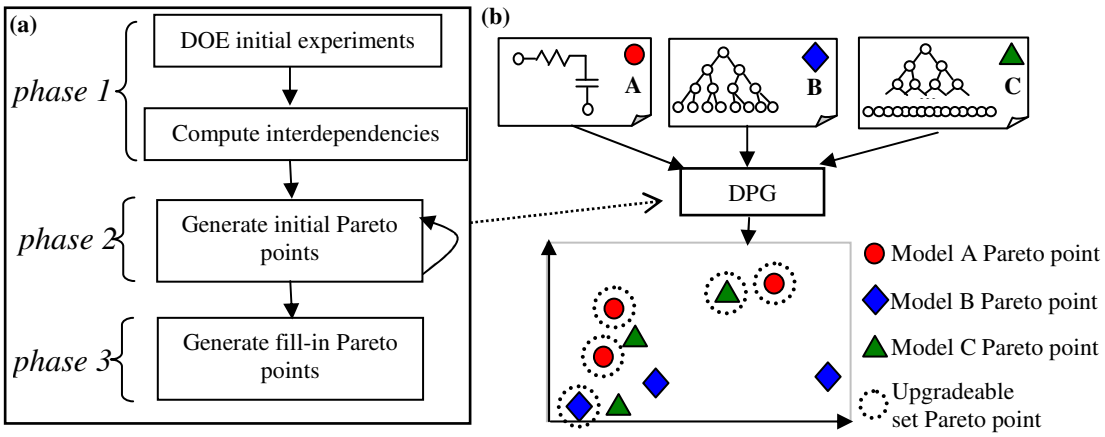

Figure 6-4: (a) The DPG algorithm flow. (b) Finding Pareto points for an upgradeable set via DPG.

was developed for use in agriculture, but has since been developed into a powerful statistical technique used in many fields. DOE automatically identifies each parameter's magnitude of influence on the solution, since the differences between physical models (complexity, connectivity of equations, etc.) can impact how much a specific parameter like PE network type or neighbor weight matters. For example, a model with few equations will not be sensitive to area constraints.

6.3.1. THE DPG ALGORITHM

The DOE-based Pareto-point Generation (DPG) algorithm [51] can be used to apply DOE to and identify PE network solution Pareto points. By applying DPG to the upgradeable models, such that the models themselves are a parameter to the algorithm, the Pareto points that span across the set can be easily located. A basic flow chart of DPG is given in Figure 6-4. DPG consists of three phases: running initial experiments to identify parameter interdependencies, generating initial Pareto points, and filling in gaps

in the Pareto curve. DOE uses either two or three-level parameters. Since PE networks have some continuous parameters, such as step size, we always select the minimum and maximum and midpoint values for continuous parameters to ensure we cover the space well enough. Phase three fills of DPG fills in the gaps of the solution space left by this discretization.

Phase 1 of DPG runs an initial Plackett-Burman [48] set of experiments to automatically generate a *weighted parameter interdependency graph*. This graph details the relationship between parameters for each metric in a single description. DPG generates the graph by first estimating the solution metrics for every pair of parameters in the system, and then running the experiment. The amount error between the estimated and actual value suggests the amount of interdependency between the parameters. Figure 6-5 shows the interdependency graph for the speedup metric for a RC lung, 6 gen. Weibel lung, and 9 gen. Weibel set of upgradeable models. Each node represents a parameter: *A* is the model from the set, *B* is the area constraint, *C* is the step size, *D* is the PE network type, *E* is the solver, and *F* is the neighbor weights. Higher edge weights represent higher levels of interdependency; for example, the 0.99 weight between *D* and *F* indicates that the effects of the PE type and neighbor weight options on the solution depend on one another, which is observable in Figure 6-3 by examining the changes in speedup due to different neighbor weights.

Phase 2 of DPG generates initial Pareto points from the parameter interdependency graph. The algorithm starts by evaluating the edge with the highest error weighting, and exhaustively searching the possible ranges of the two associated parameters. DOE uses

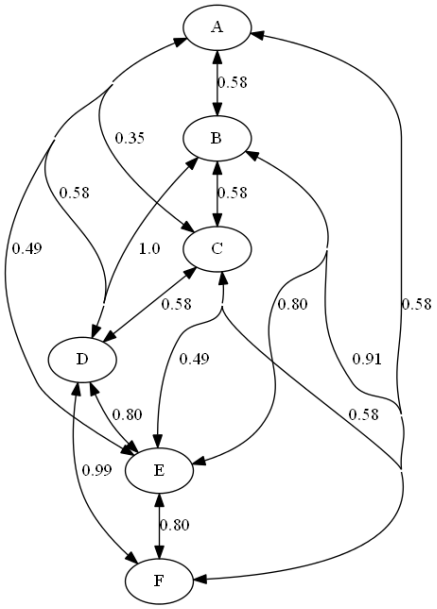

Figure 6-5: Weighted parameter interdependency graph generated by DPG.

either two or three level parameter values, so there is a maximum of nine possible configurations to run. The solutions of the search are pruned to only the local Pareto points, and the two parameter nodes of the graph are merged. This continues until only one node remains which contains a set of Pareto points for the entire design.

Phase 3 of DPG identifies regions which were not explored, due to the reduction of continuous parameters into a discrete three-level parameter. Parameters which are constant on either side of the region are locked, and a local search within the region takes place. New Pareto points are added to the set identified in phase two.

6.3.2. TOOL

A tool to explore PE network solution space consists of a web page frontend and server DPG backend, implemented in ASP.NET 4.0. A PE network compiler and

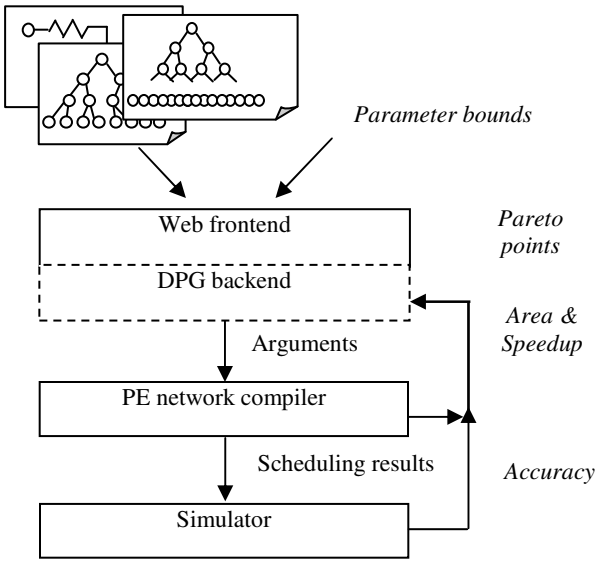

Figure 6-6: Architecture of the PE network exploration tool.

simulator are implemented as .NET WCF web services. Figure 6-6 shows the architecture of the tool. The set of upgradeable models and parameter bounds are entered by the user. Parameter bounds include minimum and maximum area constraints, step sizes, etc. The DPG backend selects a set of experiments to run, and iteratively runs the compiler and simulator to generate area, speedup, and accuracy metrics. Pareto points are selected from the results by DPG and plotted visually on the web page.

The PE network compiler accepts a set of arguments generated by the DPG algorithm to partition the equations of a specific model across a set of PEs. Once the equations have been partitioned, a scheduler generates a schedule for each PE in the network. The partitioning and schedule information is enough to generate the area and speedup metrics. Area is reported back by the compiler in terms of the number of LUTs, DSPs, and BRAM components used. The compiler automatically calculates the number

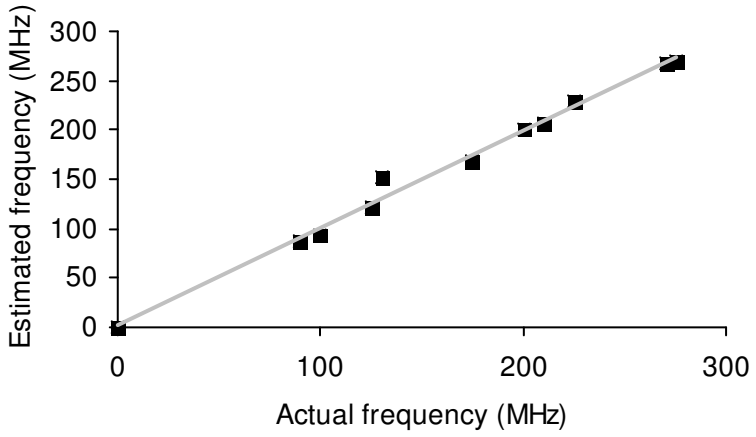

Figure 6-7: Regression model for estimating circuit frequency.

of these components based on the type and frequency of each PE type. General PEs use only one DSP and one BRAM each, while custom PEs may use arbitrary numbers of DSPs and typically a single BRAM. The final area metric is *equivalent LUTs*, which is a useful method for comparing resource usage for designs with various usage of logic cells and hard macros like DSPs. For a Xilinx Virtex6-240T, we use the following equation to calculate equivalent LUTs, where L_{EQ} is the number of equivalent LUTs, L is the number of LUTs, K_{DSP} is the equivalent LUTs per DSP (250), D is the number of DSPs, K_{BRAM} is the equivalent LUTs per BRAM (360), and B is the number of BRAMs:

$$L_{EQ} = L + K_{DSP} D + K_{BRAM} B$$

To calculate the speedup metric, the frequency of the resulting circuit must be estimated. The maximum frequency for a single PE is approximately 300 MHz when targeting a Virtex6, thus the maximum frequency that a larger PE network could achieve is also 300 MHz. As the number of PEs and connections between PEs grows larger, the

place-and-route tools (Xilinx ISE 14.2) can not maintain the same timing due to congestion. We have created a regression model to estimate the frequency based on FPGA resource usage and the number of connections in the design:

$$Freq = K_0 - K_1 W - K_2 R_{DSP} + K_3 R_{BRAM} - K_4 R_{LUT}$$

Freq is the estimated frequency of the design, K_0, K_1, K_2, and K_3 are regression coefficients based on experimental data from PE networks targeting a Virtex6. *W* is the number of wires in the design (PE-to-PE connections), and R_{DSP}, R_{BRAM}, and R_{LUT} are resource usage ratios. This model is able to estimate frequencies to within 5% of their actual values, as shown in Figure 6-7.

Once the frequency has been estimated, total speedup can be calculated:

$$Speedup = \frac{1}{\frac{1}{Freq} * C * S}$$

C is the number of cycles required to compute one iteration of the model. *S* is the step size parameter given to the compiler. The factor 1/*Freq**C yields the amount of time to compute one iteration; multiplying by *S* then gives the simulated time per second, which is translated to a factor of real-time by dividing 1 second with the result.

Accuracy is determined by simulating the PE network. A cycle-accurate simulator executes each instruction of each PE for a short interval of time. The simulation is performed twice: once using the given solver and step size parameters, and once using a 'golden' set of parameters that consists of the most accurate configuration. For the golden parameters, we use an RK4 solver and a 0.01 ms step size.

Parameter	Low	Mid	High
LUTs	*10K*	*50K*	*150K*
DSPs	*20*	*200*	*716*
BRAMs	*20*	*200*	*417*
PE Type	*General*	*Custom*	*Hybrid*
Neighbor weight	*1*	*5*	*10*
Solver type	*Euler*	-	*RK4*
Step size (ms)	*0.01*	*0.1*	*1.0*

Table 6-1: Enumerated input parameters and bounds for DPG.

After the simulations are complete, the time-series traces of each variable are retrieved and compared. The simulator finds the variable in the user-defined simulation that is of a maximal distance from the golden standard simulation trace and returns the error. The error is then multiplied by the coefficient that describes the model's relative accuracy in the upgradeable set, as described in section 0.

6.4 CASE STUDY

The following details a case study that performs exploration of a set of upgradeable RC and Weibel lung models. We target a Xilinx Virtex6-240T FPGA, which consists of 150K LUTs, 716 DSPs, and 417 BRAMs. Table 6-1 enumerates the parameters and bounds that are input into the DPG algorithm for each set of models, since DOE uses 2 or 3-level parameters.

The set of upgradeable lung models includes the RC model and 6-generation and 9-generation Weibel models previously described. We use coefficients of 0.4, 0.9, and 1.0 to describe relative model accuracy, respectively. Figure 6-8 shows three plots comparing the area, speedup, and accuracy metrics of the 57 Pareto points found by the DPG

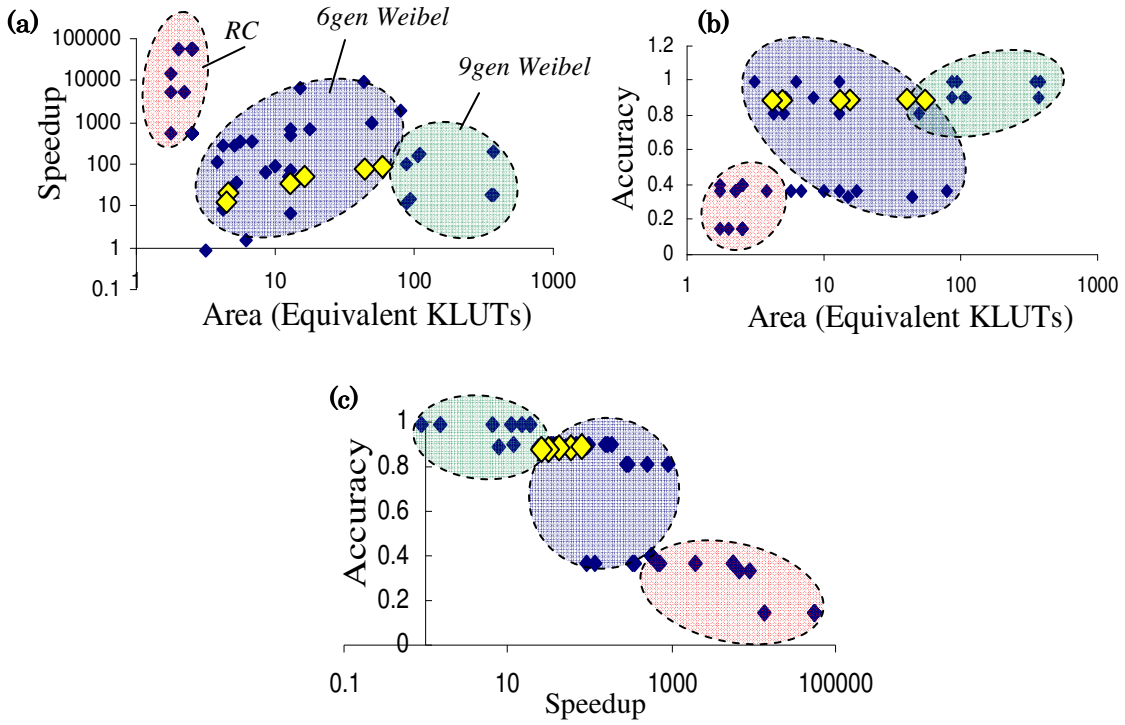

Figure 6-8: 3-dimensional Pareto point plots projected onto 2-dimensional space. (a) Speedup v. area, (b) accuracy v. area, and (c) accuracy v. speedup. Yellow points show where accuracy parameters are constant, emphasizing area/speedup tradeoff only.

algorithm. We consider the total design space size to be over 7,200 configurations, if the area constraint is discretized into just ten levels. Thus, the DPG algorithm prunes more than 99% of the design space, making exploration of the solutions more feasible. Filled circles indicate where groupings of Pareto points originate from the same model. For example, the left-most plot showing speedup vs. area has a group of Pareto points in the top-left corner that all are related to the RC model. Since the RC model is relatively simple, it has high speedup and low area requirements; however, the other plots that include accuracy indicate the low fidelity of the solution.

The lighter points of Figure 6-8 illustrate Pareto points that correspond to configurations using an RK4 solver, 0.01 ms step size, and the 9-generation Weibel model. The points represent the 'normal' design space exploration of a PE network that has configurable type, number of PEs, etc. Considering only the middle case of the 6-generation Weibel model yields a solution space with speedups between 8X and 9200X, area between 4KLUTs and 55 KLUTs, and accuracy between 0.33 and 0.89. By considering the RC and 9-generation Weibel models during exploration, the solution space expands to speedups between 0.86X and 55000X, area between 1.7KLUTs and 376KLUTs, and accuracy between 0.15 and 0.99. Overall, the solution space that can be considered has increased in size by 6X in terms of speedup, 7.3X in terms of area, and 1.5X in terms of accuracy.

REFERENCES

[1] Emmanuel Agullo, Jim Demmel, Jack Dongarra, Bilel Hadri, Jakub Kurzak, Julien Langou, Hatem Ltaief, Piotr Luszczek, and Stanimire Tomov. 2009. Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. IOP J. Physics: Conference Series, 180, 1 (2009). DOI:http://dx.doi.org/10.1145/1188913.1188915

[2] Alfred V. Aho, Michael R. Garey, and Jeffrey D. Ullman. 1972. The transitive reduction of a directed graph. *SIAM J. Comput.* 1, 2 (1972), 131-137. DOI:http://dx.doi.org/10.1137/0201008

[3] Aleliunas, R., and Rosenberg, A.L. 1982. On Embedding Rectangular Grids in Square Grids. Computers, IEEE Transactions on , vol.C-31, no.9, pp.907-913, Sept. 1982.

[4] Reid Andersen and Kumar Chellapilla. 2009. Finding dense subgraphs with size bounds. In *Algorithms and Models for the Web-Graph*, Springer Berlin Heidelberg, 25-37. DOI:http://dx.doi.org/10.1007/978-3-540-95995-3_3

[5] Banerjee, P., Sur-Kolay, S., Bishnu, A., Das, S., Nandy, and S.C., Bhattacharjee, S. 2009. FPGA placement using space-filling curves: Theory meets practice. ACM Trans. Embed. Comput. Syst. vol. 9, no. 2, Oct. 2009.

[6] Berman, F., and Snyder, L. 1987. On mapping parallel algorithms into parallel architectures, Journal of Parallel and Distributed Computing, vol. 4, no.5, Oct. 1987, pp 439-458.

[7] Bhatelé, A., and Kalé, L.V. 2008. Benefits of Topology Aware Mapping for Mesh Interconnects. Parallel Processing Letters, vol.18, no.4, pp.549-566, 2008.

[8] Bokhari, S.H. 1981. On the Mapping Problem. Computers, IEEE Transactions on , vol.C-30, no.3, pp. 207-214, March 1981.

[9] Silas Boyd-Wickizer, Haibo Chen, Rong Chen, Yandong Mao, M. Frans Kaashoek, Robert Morris, Aleksey Pesterev et al. 2008. Corey: An Operating System for Many Cores. In *Proceedings of the 8th USENIX conference on Operating Systems Design and Implementation* (OSDI '08). USENIX Association, Berkeley, CA, 43-57.

[10] Shekhar Borkar. 2007. Thousand core chips: a technology perspective. In *Proceedings of the 44th annual Design Automation Conference* (DAC '07). ACM, New York, NY, 746-749. DOI:http://dx.doi.org/10.1145/1278480.1278667

[11] Chen, W.K., and Stallmann, M. 1995. On embedding binary trees into hypercubes. J. Parallel Distrib. Comput. 24, 2 (February 1995), 132-138.

[12] Michael deLorimier, Nachiket Kapre, Nikil Mehta, Dominic Rizzo, Ian Eslick, Raphael Rubin, Tomas E. Uribe, Thomas F. Jr. Knight, and Andre DeHon. 2006. GraphStep: A System Architecture for Sparse-Graph Algorithms. *In Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '06)*. IEEE Computer Society, Washington, DC, USA, 143-151. DOI: http://dx.doi.org/10.1109/FCCM.2006.45

[13] Jack Edmonds. 1965. Maximum matching and a polyhedron with 0, l-vertices. *J. Res. Nat. Bur. Standards B*, 69, (1965), 125-130.

[14] Ellis, J.A. 1991. Embedding Rectangular Grids Into Square Grids. IEEE Transactions on Computers, pp. 46-52, Jan. 1991.

[15] Thomas M. J. Fruchterman and Edward M. Reingold. 1991. Graph drawing by force-directed placement. Softw. Pract. Exper. 21, 11 (November 1991), 1129-1164. DOI=10.1002/spe.4380211102 http://dx.doi.org/10.1002/spe.4380211102

[16] Gabryś, E., Rybaczuk, M.,and Kędzia, A. 2005. Fractal models of circulatory system. Symmetrical and asymmetrical approach comparison, Chaos, Solitons Fractals, vol. 24, no. 3, May 2005, pp 707-715.

[17] Gholkar, A., Isaacs, A., and Arya, H. 2004. Hardware-In-Loop Simulator for Mini Aerial Vehicle, Sixth Real- Time Linux Workshop, NTU, Singapore, Nov. 2004.

[18] Maya Gokhale, Jan Stone, Jeff Arnold, and Mirek Kalinowski. 2000. Stream-oriented FPGA computing in the Streams-C high level language. *In IEEE Symposium on Field-Programmable Custom Computing Machines* (FCCM '00). IEEE, 49-56. DOI:http://dx.doi.org/10.1109/FPGA.2000.903392

[19] Gopalakrishnan, P., Li, X., and Pileggi, L. 2006. Architecture-aware FPGA placement using metric embedding. In Proceedings of the 43rd annual Design Automation Conference (DAC '06). ACM, New York, NY, USA, pp. 460-465.

[20] Huang, C., Vahid, F., and Givargis, T. 2011. A Custom FPGA Processor for Physical Model Ordinary Differential Equation Solving. Embedded Systems Letters, IEEE , vol.3, no.4, pp.113-116, Dec. 2011.

[21] Huang, C., Miller, B., Vahid, F., and Givargis, T. 2012. Synthesis of custom networks of heterogeneous processing elements for complex physical system emulation. In Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis (CODES+ISSS '12). ACM, New York, NY, USA, pp. 215-224.

[22] Chen Huang, Frank Vahid, and Tony Givargis. 2013. Automatic synthesis of physical system differential equation models to a custom network of general processing elements on FPGAs. *ACM Trans. Embed. Comput. Syst.* 13, 2, Article 23 (September 2013), 27 pages. DOI:http://doi.acm.org/10.1145/2514641.2514650

[23] Michael Hucka, A. B. B. J. Finney, Benjamin J. Bornstein, Sarah M. Keating, Bruce E. Shapiro, Joanne Matthews, Ben L. Kovitz et al. 2004. Evolving a lingua franca and associated software infrastructure for computational systems biology: the Systems Biology Markup Language (SBML) project. *Systems biology* 1, 1 (2004), 41-53. DOI:http://dx.doi.org/10.1049/sb:20045008

[24] Jiang, Z., Pajic, M., and Mangharam, R. 2011. Model-Based Closed-Loop Testing of Implantable Pacemakers. *Cyber-Physical Systems* (ICCPS), 2011 *IEEE/ACM International Conference on*, pp.131-140, April 2011.

[25] David S. Johnson, Cecilia R. Aragon, Lyle A. McGeoch, and Catherine Schevon. 1989. Optimization by simulated annealing: an experimental evaluation. Part I, graph partitioning. Oper. Res. 37, 6 (October 1989), 865-892. DOI: http://dx.doi.org/10.1287/opre.37.6.865

[26] Gilles Kahn. 1974. The semantics of a simple language for parallel programming. In *Information Processing: Proceedings of the IFIP Congress.* 74, (1974).

[27] H. T. Kung. 2003. Systolic array. In Encyclopedia of Computer Science (4th ed.), Anthony Ralston, Edwin D. Reilly, and David Hemmendinger (Eds.). John Wiley and Sons Ltd., Chichester, UK 1741-1743.

[28] Keith Horsfield, Wendy Kemp, and Sally Phillips. 1982. An asymmetrical model of the airways of the dog lung. *J. Applied Physiology*, 52, 1, (1982) 21-26.

[29] Nachiket Kapre, Nikil Mehta, Michael deLorimier, Raphael Rubin, Henry Barnor, Michael J. Wilson, Michael Wrighton, and Andre DeHon. 2006. Packet Switched vs. Time Multiplexed FPGA Overlay Networks. In *Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines* (FCCM '06). IEEE Computer Society, Washington, DC, USA, 205-216. DOI:http://dx.doi.org/10.1109/FCCM.2006.55

[30] Shashi Kumar, Axel Jantsch, J-P. Soininen, Martti Forsell, Mikael Millberg, Johny Oberg, Kari Tiensyrja, and Ahmed Hemani. 2002. A Network on Chip Architecture

and Design Methodology. In *Proceedings of the IEEE Computer Society Annual Symposium on VLSI*. IEEE Computer Society, Washington, DC, USA, 105-112. DOI:http://dx.doi.org/10.1109/ISVLSI.2002.1016885

[31] Edward A. Lee and David G. Messerschmitt. 1987. Synchronous data flow. In *Proceedings of the IEEE*. IEEE Computer Society, Washington, DC, USA, 75, 9 (1987), 1235-1245. DOI:http://dx.doi.org/10.1109/PROC.1987.13876

[32] Lee, S.K., and Choi, H.A. 1996. Embedding of complete binary trees into meshes with row-column routing. Parallel and Distributed Systems, IEEE Transactions on , vol.7, no.5, pp.493-497, May 1996.

[33] Rose Liu, Kevin Klues, Sarah Bird, Steven Hofmeyr, Krste Asanović, and John Kubiatowicz. 2009. Tessellation: space-time partitioning in a manycore client OS. In *Proceedings of the First USENIX conference on Hot topics in parallelism* (HotPar'09). USENIX Association, Berkeley, CA, USA, 10-10.

[34] Catherine M. Lloyd, Matt DB Halstead, and Poul F. Nielsen. 2004. CellML: its future, present and past. *Progress in biophysics and molecular biology*, 85, 2 (2004), 433-450.DOI: http://dx.doi.org/10.1016/j.pbiomolbio.2004.01.004

[35] Alexander Marquardt, Vaughn Betz, and Jonathan Rose. 2000. Timing-driven placement for FPGAs. In Proceedings of the 2000 ACM/SIGDA eighth international symposium on Field programmable gate arrays (FPGA '00). ACM, New York, NY, USA, 203-213. DOI=10.1145/329166.329208 http://doi.acm.org/10.1145/329166.329208

[36] Mathworks. 2011. Matlab and Simulink. Retrieved April 22, 2014 from http://www.mathworks.com

[37] Matic, S. 1990. Emulation of hypercube architecture on nearest-neighbor mesh-connected processing elements. Computers, IEEE Transactions on , vol.39, no.5, pp.698-700, May 1990.

[38] Miller, J. A., Nair, R. S., Zhang, Z., Zhao, H. "JSIM: A JAVA-based simulation and animation environment". Simulation Symposium, 1997. Proceedings. 30th Annual, pp. 31-42. IEEE.

[39] Miller, B., Vahid, F., and Givargis, T. 2012. Digital mockups for the testing of a medical ventilator. In Proceedings of the 2nd ACM SIGHIT International Health Informatics Symposium (IHI '12). ACM, New York, NY, USA, pp. 859-862.

[40] Bailey Miller, Frank Vahid, and Tony Givargis. 2013. Exploration with upgradeable models using statistical methods for physical model emulation. In *Proceedings of the*

50th Annual Design Automation Conference (DAC '13). ACM, New York, NY, USA, , Article 154 , 6 pages. DOI:http://doi.acm.org/10.1145/2463209.2488925

[41] Motuk, E., Woods, R., and Bilbao, S. 2005. Implementation of finite difference schemes for the wave equation on FPGA. ICASSP.

[42] Jayram Moorkanikara Nageswaran, Nikil Dutt, Jeffrey L. Krichmar, Alex Nicolau, and Alexander V. Veidenbaum. 2009. A configurable simulation environment for the efficient simulation of large-scale spiking neural networks on graphics processors. *Neural Netw.* 22, 5 (July 2009), 791-800. DOI:http://dx.doi.org/10.1016/j.neunet.2009.06.028

[43] National Instruments. 2001. NI FPGA. Retrieved April 22, 2014 from http://www.ni.com/fpga.

[44] Edmund B. Nightingale, Orion Hodson, Ross McIlroy, Chris Hawblitzel, and Galen Hunt. 2009. Helios: heterogeneous multiprocessing with satellite kernels. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles* (SOSP '09). ACM, New York, NY, USA, 221-234. DOI:http://doi.acm.org/10.1145/1629575.1629597

[45] Nourani, Y., and Andresen, B. 1998. A comparison of simulated annealing cooling strategies. Journal of Physics A: Mathematical and General, vol. 31, no. 41, 1998.

[46] Yasunori Osana, Tomonori Fukushima, and Hideharu Amano. 2004. ReCSiP: a reconfigurable cell simulation platform: accelerating biological applications with FPGA. In *Proceedings of the 2004 Asia and South Pacific Design Automation Conference* (ASP-DAC '04). IEEE Press, Piscataway, NJ, USA, 731-733. DOI:http://doi.ieeecomputersociety.org/10.1109/ASPDAC.2004.172

[47] Michael K. Papamichael and James C. Hoe. 2012. CONNECT: re-examining conventional wisdom for designing nocs in the context of FPGAs. In *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays* (FPGA '12). ACM, New York, NY, USA, 37-46. DOI:http://doi.acm.org/10.1145/2145694.2145703

[48] Petersen, R. "Design and Analysis of Experiments". Mercel Dekker Inc. New York, New York, 1985.

[49] de Pimentel, J.C.G., and Tirat-Gefen, Y.G. 2006. Hardware Acceleration for Real Time Simulation of Physiological Systems. Engineering in Medicine and Biology Society, 2006. EMBS '06. 28th Annual International Conference of the IEEE , vol., no., pp. 218-223, Aug. 2006.

[50] Adrian Schüpbach, Simon Peter, Andrew Baumann, Timothy Roscoe, Paul Barham, Tim Harris, and Rebecca Isaacs. 2008. Embracing diversity in the Barrelfish manycore operating system. In *Proceedings of the Workshop on Managed Many-Core Systems*, p. 27.

[51] Sheldon, D., Vahid, F. "Making good points: application-specific pareto-point generation for design space exploration using statistical methods". *International Symposium on Field Programmable Gate Arrays*, 2009, pp. 123-132. ACM.

[52] Singh, S. 2011. The RLOC is dead - long live the RLOC. In Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays (FPGA '11). ACM, New York, NY, USA, pp. 185-188.

[53] Daniel A. Spielman and Shang-Hua Teng. 2004. Nearly-linear time algorithms for graph partitioning, graph sparsification, and solving linear systems. In *Proceedings of the thirty-sixth annual ACM symposium on Theory of computing* (STOC '04). ACM, New York, NY, USA, 81-90. DOI:http://doi.acm.org/10.1145/1007352.1007372

[54] Terman, D., Ahn, S., Wang, X., and Just, W. 2008. Reducing neuronal networks to discrete dynamics, Physica D: Nonlinear Phenomena, vol. 237, no. 3, March 2008.

[55] Tagkopoulos, I., Zukowski, C., Cavelier, G., and Anastassiou, D. 2003. A custom FPGA for the simulation of gene regulatory networks. In Proceedings of the 13th ACM Great Lakes symposium on VLSI (GLSVLSI '03). ACM, New York, NY, USA, pp. 132-135.

[56] Truong, D.N., Cheng, W.H., Mohsenin, T., Zhiyi Yu, Jacobson, A.T., Landge, G., Meeuwsen, M.J., Watnik, C., Tran, A.T., Zhibin Xiao, Work, E.W., Webb, J.W., Mejia, P.V., Baas, B.M. 2009. A 167-Processor Computational Platform in 65 nm CMOS, *Solid-State Circuits, IEEE Journal of*, 44 (4), 1130-1144, April 2009 DOI:http://doi.acm.org/10.1109/JSSC.2009.2013772

[57] Ullma, J.D. 1984. Computational Aspects of VLSI. W. H. Freeman & Co., New York, NY, USA.

[58] Ullman, S., and Narahari, B. 1990. Mapping binary precedence trees to hypercubes and meshes. Parallel and Distributed Processing, 1990. *Proceedings of the Second IEEE Symposium on* , pp. 838-841, Dec. 1990.

[59] van Meurs, WL. 2011. Modeling and Simulation in Biomedical Engineering: Applications in Cardiorespiratory Physiology. McGraw-Hill Professional.

[60] Wagner, A.S. 1991. Embedding all binary trees in the hypercube. *Parallel and Distributed Processing*, Proceedings of the Third IEEE Symposium on , pp. 104-111, Dec 1991.

[61] Weibel, E.R. 1963. Morphometry of the Human Lung. Berlin, Germany: Springer-Verlag 1963.

[62] Xilinx, 2010. Inc. Virtex-6 FPGA Routing Optimization Design Techniques. http://www.xilinx.com/support/documentation/white_papers/wp311.pdf

[63] Yoshimi, Masato, Yasunori Osana, Tomonori Fukushima, and Hideharu Amano. 2004. Stochastic simulation for biochemical reactions on FPGA. In *Field Programmable Logic and Application*, pp. 105-114. Springer Berlin Heidelberg, 2004. DOI:http://dx.doi.org/10.1007/978-3-540-30117-2_13

[64] Zhiyi Yu, Michael Meeuwsen, Ryan Apperson, Omar Sattari, Michael Lai, Jeremy Webb, Eric Work, Tinoosh Mohsenin, Mandeep Singh, and Bevan Baas. 2006. An asynchronous array of simple processors for DSP applications. *In IEEE International Solid-State Circuits Conference* (ISSCC'06), 49, 428-429.

[65] Zhang, H., Holden, A.V., and Boyett, M.R. 2001. Gradient model versus mosaic model of the sinoatrial node. *Circulation*, 103, 584-588.

[66] Zienicke, P. 1990. Embeddings of Treelike Graphs into 2-Dimensional Meshes. *In Proceedings of the 16th International Workshop on Graph-Theoretic Concepts in Computer Science* (WG '90). London, UK, pp. 182-192.