GRAph Parallel Actor Language — A Programming Language for Parallel Graph Algorithms

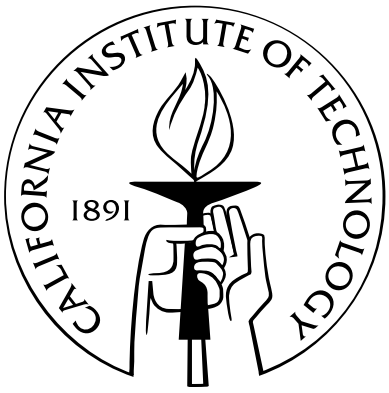
Thesis by

Michael deLorimier

In Partial Fulfillment of the Requirements

for the Degree of

Doctor of Philosophy



California Institute of Technology

Pasadena, California

2013

(Defended June 12, 2012)

Abstract

We introduce a domain-specific language, GRAph PArallel Actor Language, that enables parallel graph algorithms to be written in a natural, high-level form. GRAPAL is based on our GraphStep compute model, which enables a wide range of parallel graph algorithms that are high-level, deterministic, free from race conditions, and free from deadlock. Programs written in GRAPAL are easy for a compiler and runtime to map to efficient parallel field programmable gate array (FPGA) implementations. We show that the GRAPAL compiler can verify that the structure of operations conforms to the GraphStep model. We allocate many small processing elements in each FPGA that take advantage of the high on-chip memory bandwidth (5x the sequential processor) and process one graph edge per clock cycle per processing element. We show how to automatically choose parameters for the logic architecture so the high-level GRAPAL programming model is independent of the target FPGA architecture. We compare our GRAPAL applications mapped to a platform with four 65 nm Virtex-5 SX95T FPGAs to sequential programs run on a single 65 nm Xeon 5160. Our implementation achieves a total mean speedup of 8x with a maximum speedup of 28x. The speedup per chip is 2x with a maximum of 7x. The ratio of energy used by our GRAPAL implementation over the sequential implementation has a mean of $1/10$ with a minimum of $1/80$.

Contents

Chapter 1

Introduction

My Thesis: GRAPAL is a DSL for parallel graph algorithms that enables them to be written in a natural, high-level form. Computations restricted to GRAPAL's domain are easy to map to efficient parallel implementations with large speedups over sequential alternatives.

Parallel execution is necessary to efficiently utilize modern chips with billions of transistors. There exists a need for programming models that enable high-level, correct and efficient parallel programs. To describe a parallel program on a low level, concurrent events must be carefully coordinated to avoid concurrency bugs, such as race conditions, deadlock and livelock. Further, to realize the potential for high performance, the program must distribute and schedule operations and data across processors. A good parallel programming model helps the programmer capture algorithms in a natural way, avoids concurrency bugs, enables reasonably efficient compilation and execution, and abstracts above a particular machine or architecture. Three desirable qualities of a programming model are that it is general and captures a wide range of computations, high level so concurrency bugs are rare or impossible, and easy to map to an efficient implementation. However, there is a tradeoff between a programming model being general, high-level and efficient.

The GRAPAL programming language is specialized to parallel graph algorithms so it can capture them on a high level and translate and optimize them to an efficient low-level form. This domain-specific language (DSL) approach, which GRAPAL takes, is a common approach used to improve programmability and/or performance by trading off generality. GRAPAL is based on the GraphStep compute model [1, 2], in which operations are localized to graph nodes and edges and messages flow along edges to make the structure of

the computation match the structure of the graph. GRAPAL enables programs to be deterministic without race conditions or deadlock or livelock. Each run of a deterministic program gets the same result as other runs and is independent of platform details such as the number of processors. The structure of the computation is constrained by the GraphStep model, which allows the compiler and runtime to make specialized scheduling and implementation decisions to produce an efficient parallel implementation. For algorithms in GraphStep, memory bandwidth is critical, as well as network bandwidth and network latency. The GRAPAL compiler targets FPGAs, which have high on-chip memory bandwidth (Table 1.1) and allow logic to be customized to deliver high network bandwidth with low latency. In order to target FPGAs efficiently, the compiler needs the knowledge of the structure of the computation that is provided by restriction to the GraphStep model. GraphStep tells the compiler that operations are local to nodes and edges and communicate by sending messages along the graph structure, that the graph is static, and that parallel activity is sequenced into iterations. The domain that GRAPAL supports, as a DSL, is constrained on top of GraphStep to target FPGAs efficiently. Local operations are feed-forward to make FPGA logic simple and high-throughput. These simple primitive operations are composed by GraphStep into more complex looping operations suitable for graph algorithms.

We refer to the static directed multigraph used by a GraphStep algorithm as $G = (V, E)$. V is the set of nodes and E is the set of edges. (u, v) denotes a directed edge from u to v.

1.1 High-Level Parallel Language for Graph Algorithms

An execution of a parallel program is a set of events whose timing is a complex function of machine details which include operator latencies, communication latencies, memory and cache sizes, and throughput capacities. These timing details affect the ordering of low-level events, making it difficult or impossible to predict the relative ordering of events. When operations share state, the order of operations can affect the outcome due to write after read, read after write, or write after write dependencies. When working at a low level, the programmer must ensure the program is correct for any possible event ordering. Since it is very difficult to test all possible execution cases, race-condition bugs will be

exposed late, when an unlikely ordering occurs or when the program is run with a different number of processors. Even if all nondeterministic outcomes are correct, it is difficult to understand program behavior due to lack of repeatability. nondeterminism can raise a barrier to portability since the machine deployed and the test machine often expose different orderings.

Deadlock occurs when there is a cycle of N processes in which each process, P_i, is holding resource, R_i, and is waiting for $P_{(i+1) \bmod N}$ to release $R_{(i+1) \bmod N}$ before releasing R_i. When programming on a primitive level, with locks to coordinate sharing of resources, deadlock is a common concurrency bug. Good high-level models prevent deadlock or help the programmer avoid deadlock. Many data-parallel models restrict the set of possible concurrency patterns by excluding locks from the model, thereby excluding the possibility of deadlock. In transactional memory the runtime (with possible hardware support) detects deadlock then corrects it by rolling back and re-executing.

Even if it seems that deadlock is impossible from a high-level perspective, bufferlock, a type of deadlock, can occur in message passing programs due to low-level resource constraints. Bufferlock occurs when there is a cycle of hardware buffers where each buffer is full and is waiting for empty slots in the next buffer [3]. Since bufferlock depends on hardware resource availability, this is another factor that can limit portability across machines with varying memory or buffer sizes. For example, it may not work to execute a program on a machine with more total memory but less memory per processor than the machine it was tested on.

The GraphStep compute model is designed to capture parallel graph algorithms at a high level where they are deterministic, outcomes do not depend on event ordering, and deadlock is impossible. A GraphStep program works on a static directed multigraph in which state-holding nodes are connected with state-holding edges. First, GraphStep synchronizes parallel operations into iterations, so no two operations that read or write to the same state can occur in the same iteration. Casting parallel graph algorithms as iterative makes them simple to describe. In each iteration, or *graph-step*, active nodes start by performing an *update* operation that accesses local state only and sends messages on some of the node's successor edges. Each edge that receives a message can perform an

operation that accesses local edge state only and sends a single message to the edge's destination node. Destination nodes then accumulate incoming messages with a *reduce* operation and store the result for the update operation in the next graph-step. A global barrier-synchronization separates these *reduce* operations at the end of one graph-step from the *update* operations at the beginning of the next. Figure 1.2 shows the structure of these operations in a graph-step for the simple graph in Figure 1.1. Since node update and edge operations act on local state only and fire at most once per node per graph-step, there is no possibility for race conditions. With the assumption that the reduce operation is commutative and associative, the outcome of each graph-step is deterministic. All operations are atomic so the programmer does not have to reason about what happens in each operation, or whether to make a particular operation atomic. There are no dependency cycles, so deadlock on a high level is impossible. Since there is one message into and one message out of each edge, the compiler can calculate the required message buffer space, making bufferlock impossible.

1.2 Efficient Parallel Implementation

Achieving high resource utilization is usually more difficult when targeting parallel machines than when targeting sequential machines. If the programmer is targeting a range of parallel machine sizes, then program efficiency must be considered for each possible number of parallel processors. In a simple abstract sequential model, performance is only affected by total computation work. That is, if T is the time to complete a computation and W is the total work of the computation, then $T = W$. In general, parallel machines vary over more parameters than sequential machines. One of the simplest parallel abstract machine models is Parallel Random Access Machine (PRAM) [4], in which the only parameter that varies is processor count, P. In the PRAM model, runtime depends is the maximum amount of time used by any processor: $T = \max_{i=1}^{P} w_i$. Minimizing work, $W = \sum_{i=1}^{P} w_i$, still helps minimize T, but extra effort must be put into load balancing work across processors, so $T \approx W/P$. Other parallel machine models include parameters for network bandwidth, network latency, and network topology. The performance model

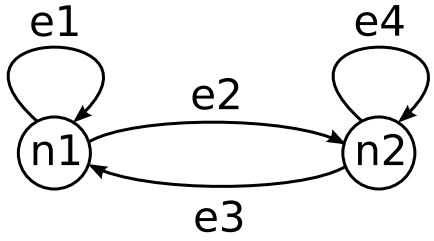


Figure 1.1: Simple graph

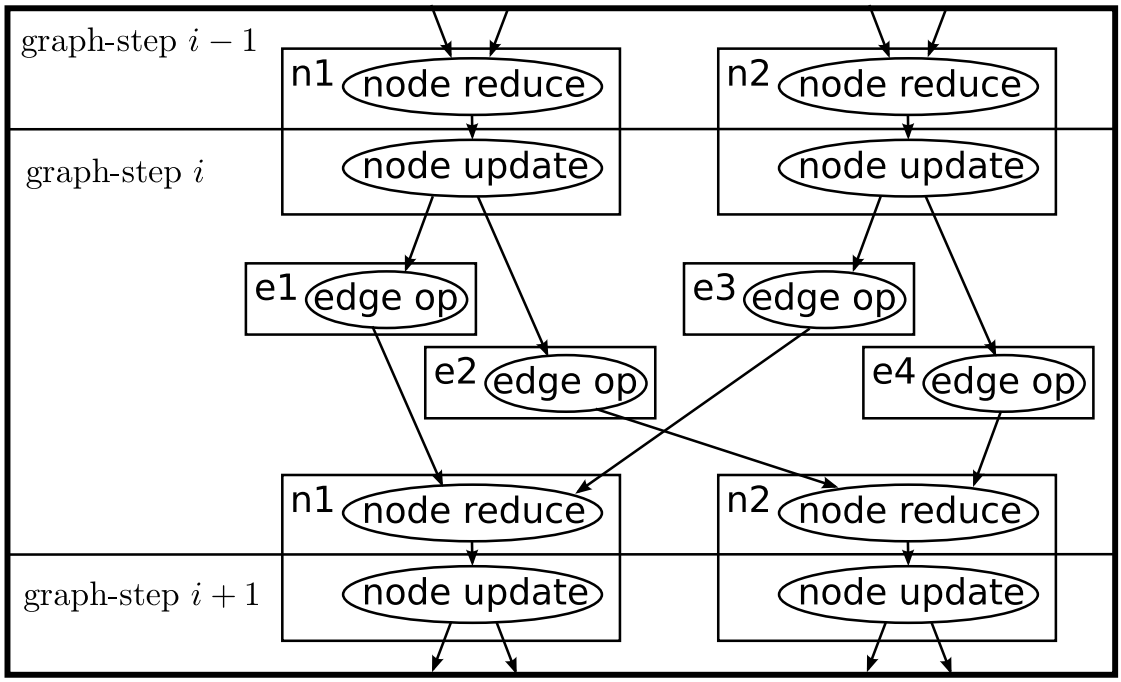


Figure 1.2: The computation structure of a graph-step on the graph in Figure 1.1 is shown here. In graph-step i, node update operations at nodes n1 and n2 send to edge operations at edges e1, e2, e3 and e4, which send to node reduce operations at nodes n1 and n2.

most relevant to GraphStep is the bulk-synchronous parallel model (BSP), [5] which has processor count, a single parameter to model network bandwidth, and a single parameter to model network latency. To optimize for BSP, computation is divided into supersteps, and time spent in a superstep, S, needs to be minimized. The time spent in a superstep performing local operations is $w = \max_{i=1}^{P} w_i$, where w_i is the time spent by processor i. The time spent communicating between processors is hg, where h is the number of messages and g is the scaling factor for network load. The inverse of network bandwidth is the primary contributor to g. Each superstep ends with a global barrier synchronization whose time is l. A superstep is the sum of computation time, communication time, and barrier

synchronization time:

$$S = w + hg + l$$

Now work in each superstep ($\sum_{i=1}^{P} w_i$) needs to be minimized and load balanced, and network traffic needs to be minimized.

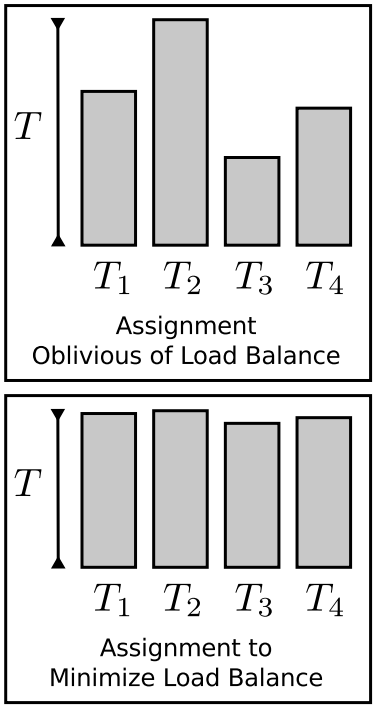


Figure 1.3: Work should be load balanced across the 4 processors to minimize runtime ($T = \max_{i=1}^{P} T_i$ in the PRAM model).

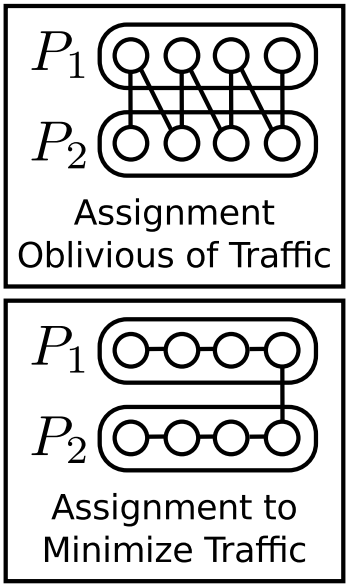


Figure 1.4: This linear topology of nodes or operators should be assigned to the 2 processors to minimize network traffic so h in the BSP model is 1 (bottom) not 7 (top).

An efficient execution of a parallel program requires a good assignment of operations to processors and data to memories. To minimize time spent according to the PRAM model, operations should be load-balanced across processors (Figure 1.3). In BSP, message passing between operations can be the bottleneck due to too little network bandwidth, g. To minimize message traffic, a local assignment should be performed so operations that communicate are assigned to the same processor (Figure 1.4). This assignment for locality should not sacrifice the load-balance, so both w and hg are minimized. Most modern parallel machines have non-uniform memory access (NUMA) [6] in which the distance between memory locations and processors matters. Typically, each processor has its own local mem-

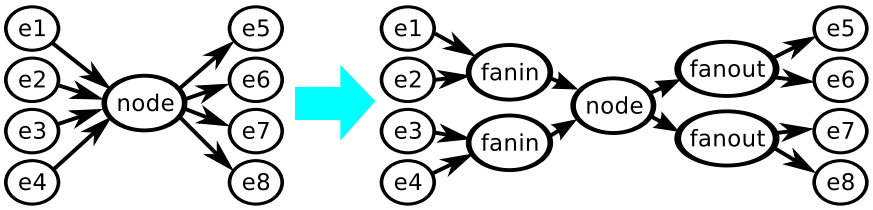


Figure 1.5: Decomposition transforms a node into smaller pieces that require less memory and less compute per piece. The commutativity and associativity of reduce operators allows them to be decomposed into fanin nodes. Fanout nodes simply copy messages.

ory which it can access with relatively low latency and high bandwidth compared to other processors' memories. In BSP each processor has its own memory and requires message passing communication for operations at one processor to access state stored at others. For graph algorithms to use BSP machines efficiently operations on nodes and edges should be located on the same processor as the node and edge objects. For graph algorithms to minimize messages between operations, neighboring nodes and edges should be located on the same processor whenever possible. Finally, load-balancing nodes and edges across processors helps load-balance operations on nodes and edges. In general, this assignment of operations and data to processors and memories may be done by part of the program, requiring manual effort by the programmer, by the compiler, or by the runtime or OS. For a good assignment either the programmer customizes the assignment for the program, or the programming model exposes the structure of the computation and graph to the compiler and runtime.

GraphStep abstracts out machine properties and assignment of operation and data to processors. The GRAPAL compiler and runtime has knowledge of the parameters of the machine being targeted so it can customize the computation to the machine. The GRAPAL runtime uses the knowledge of the structure of the graph to assign operations and data to processors and memories so machine resources can be utilized efficiently. The runtime tries to assign neighboring nodes and edges to processors so that communication work, hg, is minimized. Edge objects are placed with their successor nodes so an edge operation may get an inter-processor message on its input but never needs to send an inter-processor message on its output. Each operation on an inter-processor edge contributes to one inter-

processor message, so the runtime minimizes inter-processor edges. For GraphStep algorithms, the runtime knows that the time spent on an active node v with indegree $\Delta^-(v)$ and outdegree $\Delta^+(v)$ is proportional to $\max(\Delta^-(v), \Delta^+(v))$. Therefore, computation time in a BSP superstep at processor i, where $v(i)$ is the set of active nodes, is:

$$w_i = \sum_{v \in v(i)} \max(\Delta^-(v), \Delta^+(v))$$

Load balancing is minimizing $\max_{i=1}^{P} w_i$ so nodes are assigned to processors to balance the number of edges across processor. For many graphs, a few nodes are too large to be load balanced across many processors ($|E|/P < \max_{v \in V} \max(\Delta^-(v), \Delta^+(v))$). The runtime must break nodes with large indegree and outdegree into small pieces so the pieces can be load balanced (Figure 1.5). Node decomposition decreases the outdegree by allocating intermediate fanout nodes between the original node and its successor edges, and fanin nodes between the original node and predecessor edges. Fanout nodes simply copy messages and fanin nodes utilize the commutativity and associativity of reduce operations to break the reduce into pieces. Finally, restriction to a static graph simplifies assignment since it only needs to be performed once, when the graph is loaded.

1.3 Requirements for Efficient Parallel Hardware for Graph Algorithms

This section describes machine properties that are required for efficient execution of GraphStep algorithms.

1.3.1 Data-Transfer Bandwidths

Data-transfer bandwidths are a critical factor for high performance. In each iteration, or graph-step, the state of each active node and edge must be loaded from memory, making memory bandwidth critical. Each active edge which connects two nodes that are assigned to different processors requires an inter-processor message, making network bandwidth

	XC5VSX95T FPGA	3 GHz Xeon 5160 One Core	3 GHz Xeon 5160 Both Cores
On-chip Memory Bandwidth	4000 Gbit/s	380 Gbit/s	770 Gbit/s
On-chip Communication Bandwidth	7000 Gbit/s	N/A	770 Gbit/s

Table 1.1: Comparison of FPGA and Processor on-chip memory bandwidth and on-chip raw communication bandwidth. Both chips are of the same technology generation, with a feature size of 65 nm. The FPGA frequency is 450 MHz, which is the maximum supported by BlockRAMs in a Virtex-5 with speed grade -1 [7]. The processor bandwidth is the maximum available from the L1 cache [8]. All devices can read and write data concurrently at the quoted bandwidths. Communication bandwidth for the FPGA is the bandwidth of wires that cross two halves of the reconfigurable fabric [9, 10]. Communication bandwidth for the dual-core Xeon is the bandwidth between the two cores. Since cores communicate through caches, this bandwidth is the same as on-chip memory bandwidth.

critical (g in the BSP model). For high performance, GraphStep algorithms should be implemented on machines with high memory and network bandwidth. Table 1.1 shows on-chip memory and on-chip communication bandwidths for an Intel Xeon 5160 dual core and for a Virtex-5 FPGA. Both chips are of the same technology generation, with a feature size of 65 nm. Since the raw on-chip memory and network bandwidths of the FPGA are 5 times and 9 times higher, respectively, than the Xeon, our GRAPAL compiler should be able to exploit FPGAs to achieve high performance.

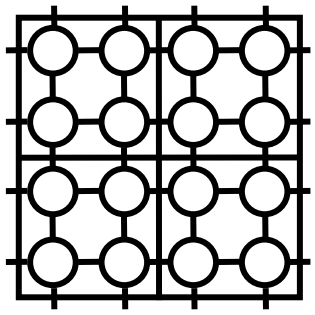


Figure 1.6: A regular mesh with nearest neighbor communication is partitioned into squares with 4 nodes per partition. Each of the 4 partitions shown here is identified with a processor.

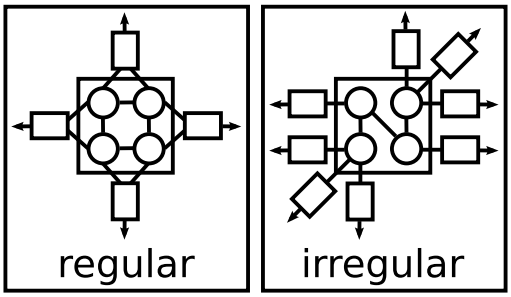


Figure 1.7: The regular mesh partition (left) has 4 neighboring partitions so its nodes can pack their 8 high-level, fine-grained messages into 4 low-level, coarse-grained messages. Nodes in the irregular graph partition (right) have 8 neighboring nodes in 8 different partitions, so each of the 8 low-level, messages carries only one high-level message.

1.3.2 Efficient Message Handling

Graph applications typically work on sparse, irregular graphs. These include semantic networks, the web, finite element analysis, circuit graphs, and social networks. A sparse graph has many fewer edges than a fully connected graph, and an irregular graph has no regular structure of connections between nodes. To efficiently support graphs with an irregular structure the machine should perform fine-grained message passing efficiently. For regular communication structures, small values can be packed into large coarse-grained messages. An example of a regular high-level structure is a 2-dimensional mesh with nodes that communicate with their four nearest neighbors. This mesh can be partitioned into rectangles so nodes in one partition only communicate with nodes in the four neighboring partitions (Figure 1.6). High-level messages between nodes are then packed into low-level, coarse-grained messages between neighboring partitions (Figure 1.7). When an irregular structure is partitioned, each partition is connected to many others so each connection does not have enough values to pack into a large coarse-grained message. High-level messages in GraphStep applications usually contain one scalar or a few scalars, so the target machine needs to handle fine-grained messages efficiently. Conventional parallel clusters typically only handle coarse-grained communication with high throughput. MPI implementations get two orders of magnitude less throughput for messages of a few bytes than for kilobyte messages [11]. Grouping fine-grained messages into coarse-grained messages has to be described on a low level and cannot be done efficiently in many cases. FPGA logic can process messages with no synchronization overhead by streaming messages into and out of a pipeline of node and edge operators. Section 5.2.1 explains how these pipelines handle a throughput of one fine-grained message per clock cycle.

1.3.3 Use of Specialized FPGA Logic

The structure of the computation in each graph-step is known by the compiler: Node update operators send to edges, edges operations fire and send to node reduce operators, then node reduce operators accumulate messages (Figure 1.2). By targeting FPGAs, the GRAPAL compiler specializes processors, or Processing Elements (PEs), for this structure. Compo-

nents of FPGA logic are often organized as spatial pipelines which stream data through registers and logic gates. We implement GraphStep operators as pipelines so each operator has a throughput of one operation per cycle. We group an edge operator, a node reduce operator, a node update operator, and a global reduce operator into each PE. This means that the only inter-PE messages are sent from a node update operation to an edge operation, which results in lower network traffic (h in BSP) than the case where each operator can send a message over the network. Since memory bandwidth is frequently a bottleneck, we allocate node and edge memories for the operators so they do not need to compete for shared state. Section 5.2.1 explains how PEs are specialized so each active edge located at a PE uses only one of the PE pipeline's slots in a graph-step. In terms of active edges per cycle, specialized PE logic gives a speedup of 30 times over a sequential processor that issues one instruction per cycle: Sequential code for a PE executes 30 instructions per active edge (Figure 5.8). Further, by using statically managed on-chip memory there are no stalls due to cache misses.

Like BSP, GraphStep performs a global barrier synchronization at the end of each graph-step. This synchronization detects when message and operation activity has quiesced, and allows the next graph-step after quiescence. We specialize the FPGA logic to minimize the time of the global barrier synchronization, l. By dedicating special logic to detect quiescence and by dedicating low-latency broadcast and reduce networks for the global synchronization signals we reduce l by 55%.

1.4 Challenges in Targeting FPGAs

High raw memory and network bandwidths and customizable logic give FPGAs a performance advantage. In order to use FPGAs, the wide gap between the high-level GraphStep programs and low-level FPGA logic must be bridged. In addition to capturing parallel graph algorithms on a high-level, GRAPAL constrains program to a domain of computations that are easy to compile to FPGA logic. Local operations on nodes and edges are feed forward so they don't have loops or recursion. This makes it simple to compile operators to streaming spatial pipelines that perform operations at a rate of one per clock

cycle. The graph is static so PE logic is not made complex by the need for allocation, deletion, and garbage collection functionality. GRAPAL's restriction that there is at most one operation per edge per graph-step allows the implementation to use the FPGA's small, distributed memories to perform message buffering without the possibility of bufferlock or cache misses.

Each FPGA model has a unique number of logic and memory resources, and a compiled FPGA program (bitstream) specifies how to use each logic and memory component. The logic architecture output by the GRAPAL compiler has components, such as PEs, whose resource usage is a function of the input application. To keep the programming model abstract above the target FPGA, the GRAPAL compiler customizes the output architecture to the amount of resources on the target FPGA platform. Unlike typical FPGA programming languages (e.g. Verilog), where the program is customized by the programmer for the target FPGA's resource count, GRAPAL supports automated scaling to large FPGAs. Chapter 8 describes how the compiler chooses values for logic architecture parameters for efficient use of FPGA resources.

1.5 Contributions

- **GraphStep compute model:** We introduce GraphStep as a minimal compute model that supports a wide range of high-level parallel graph algorithms with highly efficient implementations. We chose to make the model iterative so it is easy to reason about timing behavior. We chose to base communication on message passing to make execution efficient and so the programmer does not have to reason about shared state. GraphStep is interesting to us because we think it is the simplest model that is based on iteration and message passing and captures a wide range of parallel graph algorithms. Chapter 2 explains why GraphStep is a useful compute model, particularly how it is motivated by parallel graph algorithms and how it is different from other parallel models. This work explores the ramifications of using GraphStep: how easy it is to program, what kind of performance can be achieved, and what an efficient implementation looks like.

- **GRAPAL programming language:**

  We create a DSL that exposes GraphStep's graph concepts and operator concepts to the programmer. We identify constraints on GraphStep necessary for a mapping to simple, efficient, spatial FPGA logic, and include them in GRAPAL. We show how to statically check that a GRAPAL program conforms to GraphStep so that it has GraphSteps's safety properties.

- **Demonstration of graph applications in GRAPAL:** We demonstrate that GRAPAL can describe four important parallel graph algorithm benchmarks: Bellman-Ford to compute single-source shortest paths, the spreading activation query for the ConceptNet semantic network, a parallel graph algorithm for the netlist routing CAD problem, and the Push-Relabel method for single-source, single-sink Max Flow/Min Cut.

- **Performance benefits for graph algorithms in GRAPAL:** We compare our benchmark applications written in GRAPAL and executed on a platform of 4 FPGAs to sequential versions executed on a sequential processor. We show a mean speedup of 8 times with a maximum speedup of 28 times over the sequential programs. We show a mean speedup per chip of 2 times with a maximum speedup of 7 times. We also show the energy cost of GRAPAL applications compared to sequential versions has a mean ratio of $1/10$ with a minimum of $1/80$.

- **Compiler for GRAPAL:** We show how to compile GRAPAL programs to FPGA logic. Much of the compiler uses standard compilation techniques to get from the source program to FPGA logic. We develop algorithms specific to GRAPAL to check that the structure of the program conforms to GraphStep. These checks prevent race conditions and enable the use of small-distributed memories without bufferlock.

- **Customized logic architecture:** We introduce a high-performance, highly customized FPGA logic architecture for GRAPAL. This logic architecture is output by the compiler and is specialized to GraphStep computations in general, and also the compiled GRAPAL program in particular. We show how to pack many processing elements (PEs) into an FPGA with a packet-switched network. We show how to architect PE logic to input messages, perform node and edge operations, and output messages at a high throughput, at a rate of one edge per graph-step. We show how the PE architecture handles fine-

grained messages with no overhead. We show how to use small, distributed memories at high throughput.

- **Evaluation of optimizations:** We demonstrate optimizations for GRAPAL that are enabled by restrictions to its domain. We show a mean reduction of global barrier synchronization latency of 55% by dedicating networks to global broadcast and global reduce. We show a mean speedup of 1.3 due to decreasing network traffic by placing for locality. We show a mean speedup of 2.6 due to improving load balance by performing node decomposition. We tune the node decomposition transform and show that a good target size for decomposed nodes is the maximum size that fits in a PE. We evaluate three schemes for message synchronization that trade off between global barrier synchronization costs (l) on one hand and computation and communication throughput costs ($w + hg$) on the other. We show that the best synchronization scheme delivers a mean speedup of 4 over a scheme that does not use barrier synchronization. We show that the best synchronization scheme delivers a mean speedup of 1.7 over one that has extra barrier synchronizations used to decrease $w + hg$.
- **Automatic choice of logic parameters:** We show how the compiler can automatically choose parameters to specialize the logic architecture of each GRAPAL program to the target FPGA. With GRAPAL, we provide an example of a language that is abstract above a particular FPGA device while being able to target a range of devices. We show that our compiler can achieve a high utilization of the device, with 95% to 97% of the logic resources utilized and 89% to 94% of small BlockRAM memories utilized. We show that the mean performance achieved for the choices made by our compiler is within 1% of the optimal choice.

1.6 Chapters

The rest of this thesis is organized as follows: Chapter 2 gives an overview of parallel graph algorithms, shows how they are captured by the GraphStep compute model, and compares GraphStep to other parallel compute and programming models. Chapter 3 explains the GRAPAL programming language and how it represents parallel graph algorithms. Chap-

ter 4 presents the example applications in GRAPAL, and evaluates their performance when compiled to FPGAs compared to the performance for sequential versions. Chapter 5 explains the GRAPAL compiler and the logic architecture generated by the compiler. Chapter 6 gives our performance model for GraphStep, which is used to evaluate bottlenecks in GRAPAL applications and evaluate the benefit of various optimizations. Chapter 7 evaluates optimizations that improve the assignment of operations and data to PEs, optimizations that decrease critical path latency, and optimizations that decrease the cost of synchronization. Chapter 8 explains how the compiler chooses values for parameters of its output logic architecture as a function of the GRAPAL application and of FPGA resources. Finally Chapter 9 discusses future work for extending and further optimizing GRAPAL.

Chapter 2

Description and Structure of Parallel Graph Algorithms

This chapter starts by describing simple representative examples of parallel graph algorithms. We use the Bellman-Ford single-source shortest paths algorithm [12] to motivate the iterative nature of the GraphStep compute model. GraphStep is then described in detail. An overview is given of domains of applications that work on parallel graph algorithms. The GraphStep compute model is compared to related parallel compute models and performance models.

2.1 Demonstration of Simple Algorithms

First we describe simple versions of Reachability and Bellman-Ford in which parallel actions are asynchronous. Next, the iterative nature of GraphStep is motivated by showing that the iterative form Bellman-Ford has exponentially better time-complexity than the asynchronous form.

2.1.1 Reachability

Source to sink reachability is one of the simplest problems that can be solved with parallel graph algorithms. A reachability algorithm inputs a directed graph and a source node then labels each node for which there exists a path from the source to the node. Figure 2.1 shows the Reachability algorithm `setAllReachable`. `setAllReachable` initiates

```
class node
  boolean reachable
  set<node> neighbors
  setReachable()
    if not reachable
      reachable := true
      for each n in neighbors
        n.setReachable()

setAllReachable(node source)
  for each node n
    n.reachable := false
  source.setReachable()
```

Figure 2.1: Reachability pseudocode

activity by calling the `setReachable` method at the source node. When invoked on a node not yet labeled reachable, `setReachable` propagates itself to each successor node. If the node is labeled reachable then `setReachable` doesn't need to do anything. This algorithm can be interpreted as a sequential algorithm, in which case `setReachable` blocks on its recursive calls. It can also be interpreted as an asynchronous parallel algorithm, where calls are non-blocking. In the asynchronous case, `setReachable` should be atomic to prevent race conditions between the read of `reachable` in `if not reachable` and the `reachable := true` write. Although this algorithm is correct if `setReachable` is not atomic, without atomicity time is wasted when multiple threads enter the `if` statement at the same time and generate redundant messages. Activity in the asynchronous case propagates along graph edges eagerly, and the only synchronization between parallel events is the atomicity of `setReachable`.

2.1.2 Asynchronous Bellman-Ford

Bellman-Ford computes the shortest path from a source node to all other nodes. The input graph is directed and each edge is labeled with a weight. The distance of a path is the sum over its edges' weights. Bellman-Ford differs from Dijkstra's shortest paths algorithm in that it allows edges with negative weights. Bellman-Ford is slightly more complex than reachability and is one of the simplest algorithms that has most of the features we care about for parallel graph algorithms.

```
class node
  integer distance
  set<edge> neighboringEdges
  setDistance(integer newDist)
    if newDist < distance
      distance := newDist
      neighboringEdges.propagateDistance(newDist)

class edge
  integer weight
  node destination
  propagateDistance(integer sourceDistance)
    destination.setDistance(sourceDistance + weight)

bellmanFord(node source)
  for each node n
    n.distance := infinity
  source.setDistance(0)
```

Figure 2.2: Asynchronous Bellman-Ford

Figure 2.2 describes Bellman-Ford in an asynchronous style, analogous to the Reachability algorithm in Figure 2.1. An edge class is included to hold each edge's `weight` state and `propagateDistance` method. This method is required to add the edge's weight to the message passed along the edge. Like the Reachability algorithm this algorithm is sequential when calls are blocking and parallel when calls are non-blocking.

The problem with the asynchronous attempt at Bellman-Ford algorithm is that the number of operations invoked can be exponential in the number of nodes. An example graph with exponentially bad worst-case performance is the ladder graph in Figure 2.3. Each edge is labeled with its weight, the source is labeled with s and the sink is labeled with t. Each path from the source to sink can be represented by a binary number which lists the names of its nodes. The path "111" goes through all nodes labeled 1, and "110" goes through the first two nodes labeled 1 then the last node labeled 0. The distance of each path is the numerical value of its binary representation: The distance of "111" is 7 and the distance of "110" is 6. An adversary scheduling the order of operations will order paths discovered from s to t by counting down from the highest "111" to the lowest "000". In the general case, a graph with ladder length l has node count $2l+2$. The adversarial scheduling requires 2^l paths to be discovered, with $(l+1)2^l$ edge traversals. This is exponentially more

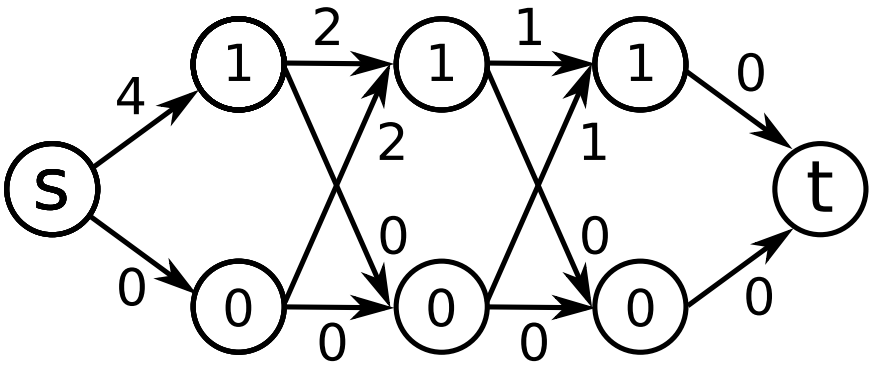


Figure 2.3: Ladder graph to illustrate pathological operation ordering for asynchronous Bellman-Ford

than a schedule ordered by breadth-first search, which requires only $4l$ edge traversals (one for each edge).

2.1.3 Iterative Bellman-Ford

To avoid a pathological ordering, firings of the `setDistance` method can be sequenced into an iteration over *graph-step*s: At each node, `setDistance` is invoked at most once per graph-step. In the asynchronous algorithm the node method, `setDistance`, first receives a single message, then sends messages. Now a single invocation of `setDistance` needs to input all messages from the previous graph-step. To do this, `setDistance` is split into two parts where the first receives messages and the second sends messages. Figure 2.4 shows the iterative version of Bellman-Ford. `reduce setDistance` receives messages from the previous graph-step, and `update setDistance` sends messages after all receives have completed. All of the messages destined to a particular node are reduced to a single message via a binary tree, with the binary operator `reduce setDistance`. To minimize latency and required memory capacity, this binary operator is applied to the first two messages to arrive at a node and again each time another message arrives. Since `update setDistance` fires only once per graph-step, it is in charge of reading and writing node state.

Synchronizing update operations into iterations is not the only way to prevent exponential work complexity. [13] gives an efficient, distributed version of Shortest Path with

```
class node
  integer distance
  set<edge> neighboringEdges
// The reduce method to handle messages when they arrive.
  reduce setDistance(integer newDist1, integer newDist2)
    return min(newDist1, newDist2)
// The method to read and write state and send messages is
//   the same as asynchronous setDistance.
  update setDistance(integer newDist)
    if newDist < distance
      distance := newDist
      neighboringEdges.propagateDistance(newDist)

class edge
  integer weight
  node destination
  propagateDistance(integer sourceDistance)
    destination.setDistance(sourceDistance + weight)

bellmanFord(node source)
  for each node n
    n.distance := infinity
  source.setDistance(0)
// An extra command is required to say what to do once all
//   messages have been received.
  iter_while_active()
```

Figure 2.4: Iterative Bellman-Ford

negative weights, in which all synchronization is local to nodes. This distributed algorithm is more complex than the iterative Bellman-Ford since it requires reasoning about complex orderings of events. The iterative nature of GraphStep gives the programmer a simple model, while preventing pathological operation orderings.

Iterative Bellman-Ford leaves it to the language implementation to first detect when all messages have been received by the `reduce` method at a node so the `update` method can fire. The implementation uses a global barrier synchronization to separate `reduce` and `update` methods. This global barrier synchronization provides a natural division between graph-steps. The command `iter_while_active` directs the machine to continue on the next graph-step whenever some message is pending at the end of a graph-step.

Continuing with iterations whenever there is a message pending is fine if there are no negative cycles in the graph. If a negative cycle exists then, this algorithm will iterate forever, continually decreasing the distance to nodes in the negative cycle. Since the maximum

```
// Return true iff there is a negative loop.
boolean bellmanFord(graph g, node source)
  for each node n
    n.distance := infinity
  source.setDistance(0)
  active = true
  for (i = 0; active && i < g.numNodes(); i++)
    active := step()
  return active
```

Figure 2.5: Iterative Bellman-Ford loop with negative cycle detection

length of a minimum path in a graph without negative cycles is the number of nodes, n, we can always stop after n steps. Figure 2.5 has a revised `bellmanFord` loop which runs for a maximum of n steps and returns true iff there exists a negative cycle. The command `step` performs a graph-step and returns true iff there were any active messages at the end of the step.

2.2 GraphStep Model

The GraphStep model [1, 2] generalizes the iterative Bellman-Ford algorithms in Figures 2.4 and 2.5. Just like Bellman-Ford, the computation follows the structure of the graph and operations are sequenced into iterative steps. Since most iterative graph algorithms need to broadcast from a sequential controller to nodes and reduce from nodes to a sequential controller, GraphStep also includes these global broadcast and reduce operations.

The graph is a static directed multigraph. Each directed edge with source node i and destination node j is called a successor edge of i and predecessor edge of j. If an edge from i to j exists then node j is a successor of i and i is a predecessor of j. Each node and each edge is labeled with its state. Operations local to nodes and edges read and write local state only. An operation at a node may send messages to its successor edges, and an operation at an edge may send a message to its destination node.

Computation and communication is sequenced into *graph-step*s. A graph-step consists of operations in the three phases:

1. **reduce:** Each node performs a reduce operation on the messages from its predecessor edges. This reduction should be commutative and associative.
2. **update:** Each node that received any reduce messages performs an update operation which inputs the resulting value from the reduce, reads and writes node state, then possibly sends messages to some of its successor edges.
3. **edge:** Each edge that received a message performs an operation which inputs it, reads and writes local state, then possibly sends a message to its destination node.

For each graph-step, at most one update operation occurs on each node and at most one operation occurs on each edge. A global barrier between graph-steps guarantees that each reduce operation only sees messages from the previous graph-step. In the reduce phase, reduce operations process all their messages before the update phase. The programmer is responsible for making the reduce operation commutative and associative so message ordering does not affect the result. Since there is at most one operation per node or edge per phase and inputs to an operation do not depend on message timing there are no race conditions or nondeterminism. This way, the programmer does not need to consider the possible timings of events that could occur across different machine architectures or operation schedules. Since there is at most one operation per node or edge per phase, the amount of memory required for messages is known at graph-load time. This way, the programmer only needs to consider whether the graph fits in memory and does not need to consider whether message passing can cause deadlock due to filling memories.

An ordinary sequential process controls graph-step activity by broadcasting to nodes, issuing step commands and reducing from nodes. These broadcast, step, and reduce commands are atomic operations from the perspective of the sequential process. The GraphStep program defines operators of each kind: reduce, update and edge. For each operator kind, multiple operators may be defined so different operators are invoked on different nodes or at different times. The global broadcast determines which operators are invoked in the next graph-step. The sequential controller uses results from global reduces to determine whether to issue another step command, issue a broadcast command, or finish. For example, the controller for Bellman-Ford in Figure 2.5 uses iteration count to determine whether to step or finish. This controller also uses the Boolean `active`, which is returned by the

step command, to indicate that node updates have not yet quiesced. Other graph algorithms need to perform a global reduction across node state. For example, Conjugate Gradient (Section 2.3.2) uses a global reduction to determine whether the error is below a threshold, and CNF SAT uses a global reduction to signal that variables are overconstrained.

A procedure implementing the general GraphStep algorithm is in Figure 2.6. In this formulation, the sequential controller calls the `graphStep` procedure to broadcast values to nodes, advance one graph-step, and get the result of the global reduce.

2.3 Graph Algorithm Examples

In this section we review applications and classes of applications which use parallel graph algorithms.

2.3.1 Graph Relaxation Algorithms

Both Reachability and Bellman-Ford are examples of a general class of algorithms that perform relaxation on directed edges to compute a fixed point on node values. Given an initial labeling of nodes, $l_0 \in L$, a fixed point is computed on a function from labelings to labelings: $f : L \to L$. A labeling is a map from nodes, V, to some values in the set A, so the set of labelings is $L = V \to A$. f updates each node's label based on its neighbors' labels:

$$f(v) = \bigvee \{\text{propagate}(l, u, v) : u \in \text{predecessors}(v)\}$$

The lattice meet operator, $\bigvee$, propagate, and the initial labeling define the graph relaxation algorithm. $\bigvee \in \mathbb{P}(L) \to L$ does an accumulation over the binary operator $\vee \in L \times L \to L$. $(L, \vee)$ is a semi-lattice where $l_1 \vee l_2$ is the greatest lower bound of l_1 and l_2. Propagate is a monotonic function which propagates the label of source node along an edge and can read but not write a value at the edge. For Bellman-Ford $\vee = \min$ and propagate adds the source's label to the edge's weight.

Graph relaxation algorithms are described in GraphStep with $\vee$ as the `node reduce` method and propagate as the `edge update` method. The `node update` method per-

```
GlobalVariable messagesToNodeReduce
(boolean, globalReduceValue) graphStep(doBcast, bcastNodes, bcastArg,
    nodeReduce, nodeUpdate, edgeUpdate, globalReduce, globalReduceId)
  g := globalReduceId
  edgeMessages := {}
  for each node n
    Variable nodeUpdateArg, fireUpdate
    if doBcast then
      nodeUpdateArg = bcastArg
      fireUpdate = n ∈ bcastNodes
    else
      nodeMessages = {m ∈ messagesToNodeReduce : destination(m) = n}
      nodeUpdateArg = reduce(nodeReduce, nodeMessages)
      fireUpdate = 0 < |nodeMessages|
    if fireUpdate then
      (s, x, y) = nodeUpdate(state(n), nodeUpdateArg)
      state(n) := s
      g := globalReduce(g, x)
      edgeMessages := union(edgeMessages, (e, y) : e ∈ successors(n))
  finalMessages := {}
  for each (e, y) ∈ edgeMessages
    (s, z) = edgeUpdate(state(e), y)
    state(e) := s
    finalMessages := union(finalMessages, (destination(e), z))
  active = 0 < |edgeMessages|
  messagesToNodeReduce = finalMessages
  return (active, g)
```

Figure 2.6: Here the GraphStep model is described as a procedure implementing a single graph-step, parameterized by the operators to use for node reduce, node update and edge update. This `graphStep` procedure is called from the sequential controller. If `doBcast` is true, `graphStep` starts by giving `bcastArg` to the `nodeUpdate` operator for each node in `bcastNodes`. Otherwise `nodeReduce` reduces all message to each node and gives the result to `nodeUpdate`. Messages used by `nodeReduce` are from `edgeUpdate`s in the previous graph-step and are stored in the global variable `messagesToNodeReduce`. The `globalReduce` operator must also be supplied with its identifier, `globalReduceId`. A Boolean indicating whether messages are still active and the result of the global reduce are returned to the sequential controller.

Algorithm	Label Type	Node Initial Value	Meet	Propagate u to v
Reachability	$\mathbb{B}$	$l(\text{source}) = T, l(\text{others}) = F$	or	$l(u)$
Bellman-Ford	$\mathbb{Z}_\infty$	$l(\text{source}) = 0, l(\text{others}) = \infty$	min	$l(u) + \text{weight}(u, v)$
DFS	$\mathbb{N}_\infty$ list	$l(\text{root}) = [], l(\text{others}) = [\infty]$	lexicographic min	$l(u) : \text{branch}(u, v)$
SCC	$\mathbb{N}$	$l(u) = u$	min	$l(u)$

Table 2.1: How various algorithms fit into graph-relaxation model

forms $\vee$ on the value it got from `reduce` and its current label to compute its next label. If the label changed then `node update` sends its new label to all successor nodes. The fixed point is found when the last graph-step has no message activity.

Other graph relaxation algorithms include depth-first search (DFS) tree construction, strongly connected component (SCC) identification, and compiler optimizations that perform dataflow analysis over a control flow graph [14]. In DFS tree construction, the value at each node is a string encoding the branches taken to get to the node from the root. The meet function is the minimum string in the lexicographic ordering of branches. In SCCs, nodes are initially given unique numerical values. The value at each node is the component it is in. For SCC, each node needs a self edge, and the meet function is minimum. The algorithm reaches a fixed point when all nodes in the same SCC have the same value. Table 2.1 describes Reachability, Bellman-Ford, DFS, and SCC in terms of their graph relaxation functions.

2.3.2 Iterative Numerical Methods

Iterative numerical methods solve linear algebra problems, which includes solving a system of linear equations (find x in $Ax = b$), finding eigenvalues or eigenvectors (find x or λ in $Ax = \lambda x$), and quadratic optimization (minimize $f(x) = \frac{1}{2}x^T Ax + b^T x$). One primary advantage of iterative numerical methods, as opposed to direct, is that the matrix can be represented in a sparse form, which minimizes computation work and minimizes memory requirements. One popular iterative method is Conjugate Gradient [15], which works when the matrix is symmetrical positive definite and can be used to solve a linear system of equations or perform quadratic optimization. Lanczos [16] finds eigenvalues and eigenvectors of a symmetrical matrix. Gauss-Jacobi [17] solves a linear system of equations when the

matrix is diagonally dominant. MINRES [18] solves least squares (finds x that minimizes $Ax - b$).

An $n \times n$ square sparse matrix corresponds to a sparse graph with n nodes with an edge from node j to node i iff there is a non-zero at row i and column j. The edge (j, i) is labeled with the non-zero value A_{ij}. A vector a can be represented with node state by assigning a_i to node i. When the graph is sparse, the computationally intensive kernel in iterative numerical methods is sparse matrix-vector multiply ($x = Ab$), where A is a sparse matrix, b is the input vector and x is the output vector. When a GraphStep algorithm performs matrix-vector multiply, an update method at node j sends the value b_j to its successor edges, the edge method at (j, i) multiplies $A_{ij}b_j$, then the node reduce method reduces input messages to get $x_i = \sum_{j=1}^{n} A_{ij}b_j$. Iterative numerical algorithms also compute dot products and vector-scalar multiplies. To perform a dot product ($c = a \cdot b$) where a_i and b_i are stored at node i, an update method computes $c_i = a_i b_i$ to send to the global reduce, which accumulates $c = \sum_{i=1}^{n} c_i$. For a scalar multiply (ka) where node i stores a_i, k is broadcast from the sequential controller to all nodes to compute at each node ka_i with an update method.

2.3.3 CAD Algorithms

CAD algorithms implement stages of a compilation of circuit graphs from a high-level circuit described in a Hardware Description Language to an FPGA or a custom VLSI layout, such as a standard cell array. The task for most CAD algorithms usually is to find an approximate solution to an NP-hard optimization problem. An FPGA router assigns nets in a netlist to switches and channels to connect logic elements. Routing can be solved with a parallel static graph algorithm, where the graph is the network of logic elements, switches and channels. Section 4.3 describes a router in GRAPAL, which is based on the hardware router described in [19]. Placement for FPGAs maps nodes in a netlist to a 2-dimensional fabric of logic elements. The placer in [20] uses hardware to place a circuit graph, which could be described using the GraphStep graph to represent an analogue of the hardware graph. Register retiming, which moves registers in a netlist to minimize the critical path re-

duces to Bellman-Ford. Section 4.1 describes the use of Bellman-Ford in register retiming.

2.3.4 Semantic Networks and Knowledge Bases

Parallel graph algorithms can be used to perform queries and inferences on semantic networks and knowledge bases. Examples are marker passing [21, 22], subgraph isomorphism, subgraph replacement, and spreading activation [23].

ConceptNet is a knowledge base for common-sense reasoning compiled from a web-based, collaborative effort to collect common-sense knowledge [23]. Nodes are concepts and edges are relations between concepts, each labeled with a relation-type. The spreading activation query is a key operation for ConceptNet used to find the context of concepts. Spreading activation works by propagating weights along edges in the graph. Section 4.2 describes our GRAPAL implementation of spreading activation for ConceptNet.

2.3.5 Web Algorithms

Algorithms used to search the web or categorize web pages are usually parallel graph algorithms. A simple and prominent example is PageRank, used to rank web pages [24]. Each web page is a node and each link is an edge. PageRank weights each page with the probability of a random walk ending up at the page. PageRank works by propagating weights along edges, similar to spreading activation in ConceptNet. PageRank can be formulated as an iterative numerical method (Section 2.3.2) on a sparse matrix. Ranks are the eigenvector with the largest eigenvalue of the sparse matrix $A + I \times E$, where A is the web graph, I is the identity matrix, and E is a vector denoting source of rank.

2.4 Compute Models and Programming Models

This section describes how graph algorithms fit relevant compute models and how GraphStep compares to related parallel compute models and performance models. The primary difference between GraphStep and other compute models is that GraphStep is customized

to its domain, so parallel graph algorithms are high level, and the compiler and runtime have knowledge of the structure of the computation.

2.4.1 Actors

Actors languages are essentially object-oriented where the objects (i.e. actors) are concurrently active and communication between methods is performed via non-blocking message passing. Actors languages include Act1 [25], ACTORS [26]. Pi-calculus [27] is a mathematical model for general concurrent computation, analogous to lambda calculus. Objects are first-class in actors languages.

Like GraphStep, all operations are atomic, mutate local object state only, and are triggered by and produce messages. Like algorithms in GraphStep, it is natural to describe a computation on a graph by using one actor to represent a graph node and one actor for each directed edge. Unlike GraphStep, actors languages are for describing any concurrent computation pattern on a low level, rather than being high level for a particular domain. Nothing is done to prevent race conditions, nondeterminism or deadlock. There is no primitive notion of barrier synchronizations or commutative and associative reduces for the compiler and runtime to optimize. Since objects are first-class, the graph structure can change, which makes processor assignment to load balance and minimize inter-processor communication difficult.

2.4.2 Streaming Dataflow

Streaming, persistent dataflow programs describe a graph of operators that are connected by streams (e.g. Kahn Networks [28], SCORE [29], Ptolemy [30], Synchronous Data Flow [31], Brook [32], Click [33]). These languages are suitable for high-performance applications such as packet switching and filtering, signal processing and real-time control. Like GraphStep, streaming dataflow languages are often high-level, domain-specific, and the static structure of a program can be used by the compiler. In particular, many streaming languages are suitable for or designed for compilation to FPGAs. The primary difference between persistent streaming languages and GraphStep is that the program is the graph,

rather than being data input at runtime. A dataflow program specifies an operator for each node and specifies the streams connecting nodes. Data at runtime is in the form of tokens that flow along each stream. There is a static number of streams into each operator, which are usually named by the program, so it can use inputs in a manner analogous to a procedure using input parameters. Some streaming models are deterministic (e.g. Kahn Networks, SCORE, SDF), and other allow nondeterminism via nondeterministic merge operators (e.g. Click). Bufferlock, a special case of deadlock, can occur in streaming languages if buffers in a cycle fill up [3]. Some streaming models prevent deadlock by allowing unbounded length buffers (e.g. Kahn Networks, SCORE), others constrain computations so needed buffer size is statically known (e.g. SDF), and in others the programmer must size buffers correctly to prevent deadlock. GraphStep's global synchronization frees the implementation from the need to track an unbounded length sequence of tokens, in the case of a model with unbounded buffers, or frees the programmer from sizing streams to prevent bufferlock, in the case of a model with bounded buffers.

The graph of operators could be constructed to match an input data graph either by using a language with a dynamic graph or by generating the program as a function of the graph. The most straight forward description of graph algorithms in this case has nodes eagerly send messages just like the initial asynchronous Bellman-Ford (Figure 2.2), leading to exponentially bad worst-case performance. In a more complex, iterative implementation, nodes firings could synchronize themselves into graph-steps by counting the number of messages received so far. First, nil-messages are used so all edges can pass one message on each graph-step. Second, each node knows the number of predecessor edges and it counts received messages up to its predecessor count before performing the update operation and continuing to the next iteration. This means the number of messages passed per graph-step equals the number of edges in the graph, which is usually at least an order of magnitude higher than in GraphStep's barrier-synchronized approach. In languages with nondeterministic merges (e.g. Click) the programmer can describe barrier synchronization, like GraphStep's implementation, to allow sparse activation with fewer messages. In this case the programmer will have to describe the same patterns for each graph algorithm, and avoid errors due to race-conditions and deadlock. Since the compiler would not be

specialized to GraphStep, it would have to allow an unbounded length sequence of tokens along each edge. Also, the implementation would not be able to do GraphStep specific optimizations, such as node decomposition.

2.4.3 Bulk-Synchronous Parallel

Bulk-synchronous parallelism (BSP) is a bridging model between parallel computer hardware and parallel programs [5]. BSP abstracts important performance parameters of parallel hardware to provide a simple performance model for parallel programs to target. BSP is also a compute model, where computation and communication are synchronized into *supersteps*, analogous to graph-steps. There are libraries for BSP, such as BSPlib [34] and BSPonMPI [35] for C and BSP ML [36] for Ocaml.

In each superstep, a fixed set of processors performs computation work and sends and receives messages. The time of each superstep is $w + hg + l$, where w is the maximum sequential computation work over processors, h is the maximum of total message sends and total message receives, g is the scaling factor for network load, and l is the barrier synchronization latency. The scaling factor g is platform dependent and is primarily determined by the inverse of network bandwidth. GraphStep can be thought of as a specialized form of BSP and the performance model is similar to GraphStep's performance model (Chapter 6). Like GraphStep, BSP's synchronized activity into supersteps makes it easy to reason about event timing. GraphStep is higher level, with program nodes and edges as the units of concurrency, rather than processors. A compiler and runtime for BSP programs cannot perform the optimizations that are performed for GraphStep since it does not have knowledge of the computation and graph structure. Our customization of FPGA logic (Section 5.2) to the structure of message passing between operations in a graph-step cannot be performed on ordinary processes. The knowledge of the graph structure and knowledge that reduce operations are commutative and associative allow GraphStep runtime optimizations that load balance data and operations (Section 7.2) and assign nodes to edges for locality (Section 7.4).

2.4.4 Message Passing Interface

Message Passing Interface (MPI) [37] is a popular standard for programming parallel clusters. Processes are identified with processors and communicate by sending coarse-grained messages. MPI is an example of a low-level model that presents difficulties to the programmer that GraphStep is designed to avoid. An MPI programmer must be careful to avoid race-conditions and deadlock. The programmer must decide how to assign operations and data to processors. Further, graph algorithms' naturally small messages are a mismatch for MPI's coarse-grained messages. For example, Bellman-Ford sends one scalar, representing distance, per message. MPI implementations get two orders of magnitude less throughput for messages of a few bytes than for kilobyte messages [11]. If Bellman-Ford's message contains a 4 byte scalar then the Bellman-Ford implementation must pack at least 256 fine-grained messages into one coarse-grained message. Extra effort is required by the programmer and at runtime to decide how to pack messages. Fragmentation may lead to too few fine-grained messages per coarse-grained message to utilize potential network bandwidth.

2.4.5 Data Parallel

Parallel activity can be described in a data parallel manner, in which operations are applied in parallel to elements of some sort of data-structure [38, 39, 40, 41]. The simplest data parallel operation is *map*, where an operation is applied to each element independently. Many data parallel languages include reduce or parallel-prefix operations [42, 41]. Some data parallel languages include the *filter* operator, which uses a predicate to remove elements from a collection. SIMD or vector languages use vectors as the parallel data structures, where each vector element is a scalar. NESL [38] and *Lisp [40] use nested vectors, where each vector element can be a scalar or another vector. Map can be applied at any level in the nested structure, and there is a concatenation operation for flattening the structure.

GraphStep is data parallel on nodes and edges and can be thought of as a data parallel language with a graph as the parallel structure rather than a vector. Like graph algorithms for GraphStep, algorithms that match the flat vector model are very efficient when mapped

to vector (or SIMD) machines. In data parallel languages the graph structure is not exposed so the computation cannot be specialized for graph algorithms. Further, most data parallel implementations are optimized for regular structures, not irregular structures.

2.4.6 GPGPU Programming Models

Programming models have been developed to provide efficient execution for General-Purpose Graphics Processing Units (GPGPUs) and to abstract above the particular device. OpenCL [43] and CUDA [44] capture the structure of GPGPUs where SIMD scalar processors are grouped into MIMD cores. Each core has one program counter which controls multiple SIMD scalar processors. OpenCL and CUDA make the memory hierarchy explicit, so the program explicitly specifies whether a data item goes in memories local to cores or memories shared between cores. To write an efficient program, the programmer uses a model of the architecture, including the number of SIMD scalar processors in each core and the sizes of memories in the hierarchy. OpenCL is general enough for programs to run on a wide range of architectures, including GPGPUs, the Cell architecture and CPUs, but requires programs to be customized to specific machines to be efficient. Although CUDA hides the number of SIMD scalar processors in each core with its Single Instruction Multiple Thread (SIMT) model, the programmer must be aware of this number in the target machine to write efficient programs. Unlike OpenCL and CUDA, GraphStep captures a class of algorithms so the programmer can write at a high level and the compiler can customize the program to target architecture.

Although parallel graph algorithms implemented on a single chip are memory bandwidth dominated, GPGPUs are usually unable to achieve a bandwidth close to peak for parallel graph algorithms. Contemporary GPGPUs do not efficiently handle reads and writes of irregular data to memory. Usually the graph structure is irregular and optimized implementations achieve about 20% of peak memory bandwidth. Conjugate Gradient is a heavily studied sparse graph algorithm implemented on GPGPUs. Conjugate Gradient (Section 2.3.2) uses Sparse Matrix Vector Multiply as its main kernel and can be described as a simple GraphStep program. For Conjugate Gradient the Concurrent Number

Cruncher [45] achieves 20% of peak memory bandwidth and a multi-GPU implementation [46] achieves 22% of peak memory bandwidth.

GraphStep enables the logic architecture to couple data with compute, so nodes and edge data is located in the same PE that performs operations on nodes and edges. Coupling data with compute allows GraphStep implementations to use small, local memories with higher peak bandwidth than off-chip main memory.

2.4.7 MapReduce

MapReduce [41] is a simple model with data parallel operators on a flat structure. Example uses are distributed grep, URL access counting, reversing web-links, word counting, and distributed sort. In MapReduce, the central data structure is a collection of key, value pairs. First, an operator is mapped to each pair to generate a collection of intermediate key, value pairs. Second, a reduce operator is applied to all intermediate values associated with each key to produce one value per key. MapReduces are chained together like graph-steps.

Like GraphStep, MapReduce gives the programmer a simple, high-level, specialized model that avoids race conditions, nondeterminism and deadlock. This simple model helps MapReduce implementers achieve fault-tolerance on large-scale machines. Although it is possible to describe parallel graph algorithms with MapReduce, MapReduce is not specialized for graph algorithms, so it is not convenient for describing graph algorithms and does not get good performance for graph algorithms. GraphLab [47], a framework for parallel graph algorithms similar to GraphStep, outperformed the Hadoop [48] implementation of MapReduce by 20 to 60 times [47].

An additional combiner operator may be specified by the programmer to perform the same function as the reduce operator, except on the processor generating intermediate key, value pairs. The combiner then sends a single key, value pair per processor per key. A combiner, reducer pair of operators in MapReduce is analogous to a node reduce method in GraphStep: In both cases the reduce function should be commutative and associative to allow it to be split into pieces located at different processors. Section 7.2 explains how this decomposition is performed for GraphStep.

2.4.8 Programming Models for Parallel Graph Algorithms

In this section we survey programming frameworks designed for parallel graph algorithms. Pregel [49] has an iterative compute model similar to GraphStep. Signal/Collect [50] supports both iterative synchronization and asynchronous message passing and is designed for (semantic) web queries. GraphLab [47] is designed for parallel machine learning algorithms. CombinatorialBLAS [51] is designed for linear algebra on sparse matrices. In these frameworks, the structure of the computation is based on the structure of the graph, which is known to the runtime. Pregel and CombinatorialBLAS computations and optionally Signal/Collect and GraphLab computations are iterations over parallel activity at nodes and edges with one operation per node per step. GraphStep restricts the structure of the computation to a greater extent than these frameworks, which makes it safer to program and easier for the compiler and runtime to optimize (especially for FPGAs), but narrows the domain of algorithms.

Pregel [49] is a C++ library for parallel graph algorithms on large clusters. Like GraphStep, operations are local to nodes and communicate by sending messages to successor nodes. Operations on nodes are sequenced into steps, and operators are similar to GraphStep's operators: A Compute() method in Pregel is analogous to a node update method, but also does the work of successor edge methods. Compute() has access to successor edge state and is in charge of sending messages to successor nodes. A Combine() method in Pregel is analogous to a node reduce method and permits the implementation to break reductions into pieces, similar to our decomposition optimization (Section 7.2). An Aggregator is analogous to a fused global reduce and global broadcast in GraphStep. After a global reduce is performed in one step, its value is available to Compute() methods in the next. Instead of using a sequential controller process to broadcast values to nodes, Pregel shares global state for Compute() methods. Unlike GraphStep, nodes are first class to allow Compute() to mutate the graph. Pregel example applications in [49], PageRank, ShortestPaths, Bipartite Matching, and Semi-Clustering, do not mutate the graph. The Pregel implementation is fault-tolerant which allows it to scale to large clusters. Pregel's fault-tolerance scheme uses rollback to return the computation to a previously checkpointed state when a

fault occurs. Its barrier-synchronized, iterative structure enables checkpointing rollback by providing a clean state at each barrier.

Signal/Collect [50] is a Scala library designed to perform queries on the web, especially the semantic web. Operations are local to nodes and communicate by sending messages to successor nodes. Signal/Collect supports multiple synchronization styles: Operations on nodes can be sequenced into graph-steps, they can fire in an asynchronous style, whenever they receive a message, or they can use a scheme to fire asynchronously only for high-priority messages. Operations on nodes are broken into Collect, to process incoming messages, and Signal, to send messages. Collect is analogous to GraphStep's node reduce methods, and Signal is analogous to GraphStep's node update messages. Like GraphStep, Signal/Collect is for static graphs.

GraphLab [47] is a C++ library for parallel graph algorithms for machine learning. Like GraphStep, GraphLab's computation is based on the graph structure. Unlike GraphStep, GraphLab's node operation reads and write to neighboring nodes' state instead of sending messages between nodes. The schedule of node updates can be synchronous, like GraphStep, or asynchronous, or specified by the programmer. Node operations are atomic (which implies sequential consistency), which is enforced by synchronizing a node's operations with its neighbors. This synchronization is implemented by either coloring to pre-schedule operations, or dynamically locking all nodes in a neighborhood. GraphLab's neighborhood locking reduces the available parallelism, especially when there are nodes with many neighbors (which occur graphs with a power-law distribution over node degree).

Combinatorial BLAS [51] is a C++ library designed for linear algebra on sparse matrices and can be used for iterative numerical methods (Section 2.3.2). It provides combinators for a variety of operations on sparse matrices and dense vectors. Each vector element corresponds to a graph node, and each non-zero in a sparse matrix corresponds to an edge. Each supported linear algebra operation is a combinator, which takes the definition of a scalar field as a parameter. Defining the scalar field corresponds to defining methods in GraphStep. Each matrix operation is atomic, which sequences parallel operations on into steps, like GraphStep. Unlike GraphStep, new graph structures can be created by multiplying two sparse matrices.

2.4.9 High-Level Synthesis for FPGAs

Since GRAPAL is designed to target FPGAs efficiently we compare to other approaches for supporting high-level languages mapped to FPGAs. A common approach is to compile C or a modification of C to Register-Transfer Level (RTL). The RTL can be described with VHDL or Verilog and is the level of abstraction at which FPGA programmers typically work. Similarly, our compiler maps the high-level GRAPAL program to RTL in VHDL (Section 5.1.1.1).

There is usually not enough instruction-level parallelism in a C program for it to be worthwhile to use an FPGA. Instead of using instruction-level parallelism, compilers and languages for high-level synthesis use process-level parallelism and data-parallel inner loops. An efficient program in Handel-C [52], Transmogrifier C [53], Streams-C [54], RaPiD-C [55] or SystemC [56] is a static set of processes that communicate on static channels. Like persistent streaming languages, (Section 2.4.2) Streams-C, Handel-C, SystemC, Catapult-C [57], and AutoESL [58] support streaming channels between processes. The primary difference between high-level synthesis approaches and GraphStep is the same as the difference between persistent dataflow and GraphStep: the program is the graph instead of being data input at runtime. Each process is a complex sequential process like Communicating Sequential Processes [59]. Like methods in GRAPAL, processes cannot perform recursion or memory allocation. This restriction enables processes to be compiled to efficient FPGA logic. Unlike methods in GRAPAL, processes can contain loops, and each process is typically a collection of loop nests. In Handel-C, Transmogrifier C, Streams-C, and RaPiD-C each process is made parallel by the programmer explicitly specifying which inner loops are parallel.

Another approach couples a sequential Instruction Set Architecture (ISA) processor with a reconfigurable coprocessor. In this a approach, a compiler extracts kernels from an ordinary C program to be run on the reconfigurable fabric. These kernels group instructions into large RTL operators, so each group can be executed in parallel on the reconfigurable fabric. For example, the GARP compiler [60] identifies VLIW-style hyperblocks [61] to be translated into RTL operators. Like GRAPAL, each of GARP's RTL operators is

transformed from a feed-forward control-flow structure. More recent projects extend the coprocessor approach by mapping looping functions to multiple, parallel RTL modules. LegUp [62] and Altera's C2H [63] map sub-procedures that loop but do not recurse or allocate memory to RTL modules. These modules use local memories and also have access to main memory to enable access to large data structures.

Chapter 3

GRAPAL Definition and Programming Model

GRAph Parallel Actor Language (GRAPAL) is our high-level programming language for the GraphStep compute model. GRAPAL describes the parallel kernels of GraphStep with nodes and edge objects (i.e. actors) passing messages along the graph structure. Along with GRAPAL as the kernel language, a sequential controller is described in C. Nodes and edges are statically typed as classes to catch programming errors and to enable an efficient implementation. Since objects communicate with message passing and GRAPAL's types are static, we use familiar syntax similar to the statically typed, object-oriented language Java. We use Bellman-Ford as an example to explain language features. Figure 3.1 shows Bellman-Ford's kernels in GRAPAL, Figure 3.6 shows the sequential controller in C, and Figure 3.7 shows the C header file which declares the interface for the sequential controller to use to access GRAPAL kernels. Appendix A contains GRAPAL's context-free grammar.

3.1 GRAPAL Kernel Language

A GRAPAL program is a set of class and function definitions, where each class contains methods, `out` fields and value fields. Each class has one of three attributes `node`, `edge`, or `global`. A GRAPAL program has any number of `node` and `edge` classes and exactly one `global` class. Our Bellman-Ford example only has one `node` class, `Node`, and one `edge` class, `Edge`. The `global` class (`Glob` in Bellman-Ford) defines the interface between the sequential controller and the parallel code.

```
global Glob {
  out Node source;
  bcast bcastToSource (int<32>) source.update;
}

node Node {
  out Edge successorEdges;
  boolean inf;
  int<32> dist;

  reduce tree update(int<32> dist1, int<32> dist2) {
    return dist1 < dist2 ? dist1 : dist2;
  }
  send update(int<32> distArg) {
    if (inf || distArg < dist) {
      inf = false;
      dist = distArg;
      successorEdges.propagate(dist);
    }
  }
}

edge Edge {
  out Node dest;
  int<32> len;

  fwd propagate(int<32> dist) {
    dest.update(len + dist);
  }
}
```

Figure 3.1: Bellman-Ford in GRAPAL

Sets of objects are typed and named in GRAPAL with class fields labeled with the attribute `out`. These `out` sets determine the structure of the graph and are used to send messages. They are used for all message sends, which includes messages from nodes to edges, from edges to nodes, from a global broadcast to nodes, and from nodes to a global reduce. Since the graph is static, the elements of each `out` set are specified by the input graph. In Bellman-Ford `Node`s have only one type of neighbor so `successorEdges` is their only `out` set. After the graph is loaded, the only use for `out` sets is to send messages. A dispatch statement of the form `<out-set>.<dest-method>(<arguments>)` is used by some method to send a message to each object in `<out-set>`. When a message arrives, it invokes `<dest-method>` applied to `<arguments>`. In Bellman-Ford, the dispatch statement `successorEdges.propagate(dist)` in `send update` sends identical messages to each successor `Edge` of the sending `Node`. In general, different `out` sets in a class may be used by different methods or by the same method. Since each edge has a single successor `node`, an edge class can only declare one `out` set which contains exactly one element.

To enable global broadcast to a subset of Nodes, the `global` object declares `out` sets. Many algorithms use an `out` set in the `global` class which contains all nodes. Bellman-Ford only broadcasts to the source node, which is a set of size one and is determined by the input graph. Each node which sends a message to a global reduce dispatches on an `out` set whose only element is the `global` object. There is at most one `out` set declared to point to the `global` class per `node` class.

Methods in GRAPAL are used to represent GraphStep operators. Each graph-step is a subsequence of four phases, and a method's attributes declare the phase the method is in:

- **Node Reduce:** A **`reduce tree`** method in a **`node`** class defines a binary operator used to reduce all pending messages to a single message.
- **Update:** A **`send`** method in a **`node`** class defines an update operator, which inputs a message, can read and write node state, and can send messages on `out` sets. A `send` method's arguments are supplied by the message which triggers it. This input message may be the result of a `reduce tree` method, may be from a global broadcast, or from

an edge.

- **Edge:** A **`fwd`** method in a **`edge`** class is the only kind of `edge` method. The input message is received from the `edge`'s predecessor node and an output message may be sent to the successor node. `fwd` methods may read and write edge state.
- **Global Reduce:** A **`reduce tree`** method in the **`global`** class defines a binary operator used to reduce all global reduce messages, which are sent from `send` methods, to a single message. Each `global reduce tree` method defines an identity value to be the result of the global reduce when there are no global reduce messages.

The fifth and final kind of method in GRAPAL is **`bcast`**, which goes in the **`global`** class. A `bcast` method's contents are of the form `<out-set>.<dest-method>`, which specifies the set of nodes to broadcast to and the destination `send` method. Figure 3.5 shows the communication structure of `bcast`, `node reduce tree`, `send`, `fwd`, and `global reduce tree` methods on the simple graph in Figure 3.2. In Figure 3.3 graph-step i is initiated with a `bcast` method and in Figure 3.4 graph-step i follows another graph-step.

To specify where messages go that are produced by each accumulation over `node reduce tree` binary operators, the programmer pairs `node reduce tree` and `send` methods by giving them the same name. A `node reduce tree` method uses a `return` statement to output its value rather than a dispatch statement because the implementation decides whether output messages go back to the `node reduce tree` accumulate or forward to the paired `send` method. Although a `node reduce tree` method cannot exist by itself, a `send` methods can since it may receive messages from `bcast` or `edge fwd` methods.

GRAPAL methods are designed to be very lightweight so they can be compiled to simple and efficient FPGA logic. Methods do not contain loops or call recursive functions to allow them to be transformed into primitive operations with feed-forward data dependencies. Section 5.2.1 describes how this feed-forward structure allows operations to flow through PE logic at a rate of one per clock-cycle. To exclude loops, the Java-like control-flow syntax includes `if-then-else` statements, but not `for` or `while` loops. An iteration local to an object can be performed with multiple method firings in successive

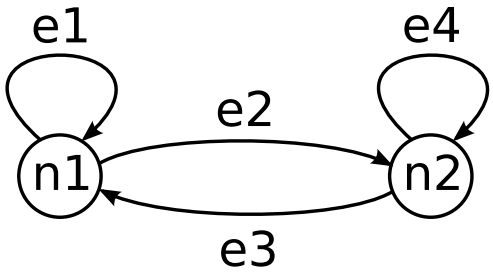


Figure 3.2: Simple graph

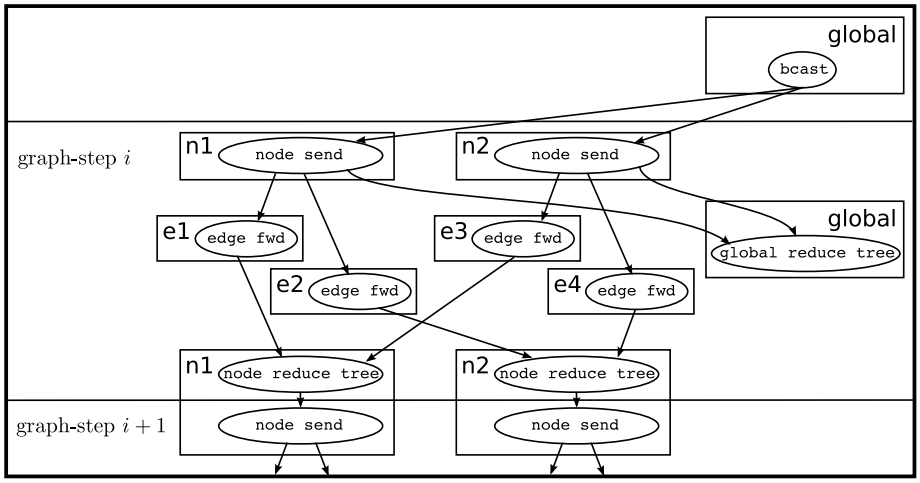


Figure 3.3: Graph-step i follows a `bcast` send.

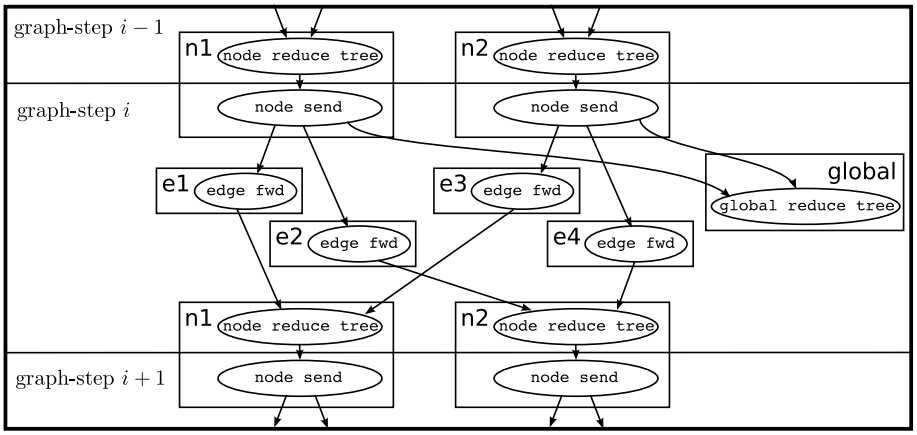


Figure 3.4: Graph-step i follows graph-step $i+1$.

Figure 3.5: The computation structure of a graph-step following a `bcast` call (Figure 3.3) and a graph-step following another graph-step (Figure 3.4) on the graph in Figure 3.2 is shown here. Methods firings are located in nodes n1 or n2 or edges e1, e2, e3, e4 or in the global object.

```
#include <gsutil.h>
#include "gsface.h"

int gs_main(int argc, char** argv) {
  Graph g = get_main_graph();
  bcast_bcastToSource(0);
  char active = 1;
  int i;
  for (i = 0; active && i <= num_nodes(g); i++)
    step(&active);
  return 0;
}
```

Figure 3.6: Sequential controller for Bellman-Ford in C

```
#include <gsutil.h>
#define Graph unsigned

Graph get_main_graph();

// number of nodes or edges of any class
unsigned num_nodes(Graph g);
unsigned num_edges(Graph g);

// number of objects of each defined class
unsigned num_Node(Graph g);
unsigned num_Edge(Graph g);

// broadcast to each defined bcast method
void bcast_bcastToSource(int in0);

// advance one graph-step; set *pactive = no pending messages
void step(char* pactive);
// advance until graph-step reached with no pending messages
void iter();
```

Figure 3.7: The header file that bridges between the Bellman-Ford sequential controller and the Bellman-Ford GRAPAL kernels. It includes functions for node and edge counts, broadcast and reduce methods, and graph-step advancing commands.

graph-steps. All functions are pure with no side-effects, and can be called from methods or other functions. Functions also do not have loop statements and the compiler checks that they are no recursive cycles in the call graph. This lack of recursion allows function calls to be inlined into methods so methods can be transformed into FPGA logic.

Each container that holds data values has a type that determines its values' sizes. Containers are data-fields in classes, and variables and parameters in methods and functions. The fixed size types for data-fields tell the compiler how many bits are required for object. This lets the compiler size memory words so the entire object state can be read or written in one clock cycle (Section 5.2.1). Fixed sizes are needed to size datapaths in operators, PEs, and the network (Section 5.2). Further, fixed sizes for message data allows small, distributed buffers to be given enough capacity to avoid bufferlock.

Scalar types are `boolean`, signed integers, `int<N>`, and unsigned integers, `unsigned<N>`. A `boolean` is one bit, and `int<N>` and `unsigned<N>` are `N` bits. Since we are primarily targeting FPGAs and FPGA have no natural word size, it would be arbitrary to choose a default bit-width for integer types. For this reason, all integer types are parameterized by their width. Composite values are constructed with n-ary tuples written `(A, B, ...)`. Tuples are destructed with indices starting at `0`, so `x == (x, y)[0]` and `y == (x, y)[1]`. The current version of GRAPAL does not have records, but constructor, getter, and setter functions can be defined to name tuple elements.

3.2 Sequential Controller Program

A sequential C program controls the GRAPAL kernels by sending broadcasts, receiving global reduce results and issuing commands to advance graph-steps. To generate the header file that bridges GRAPAL kernels to C, the programmer runs `graph_step init` on the command line. Section 5.1 explains how the `graph_step` utility is used to compile and run GRAPAL programs. For Bellman-Ford, Figure 3.6 shows the sequential controller, and Figure 3.7 shows the header file it uses. This interface declares global broadcast and global reduce methods defined in the `global` class, stepper commands, and procedures to query

the size of the graph. Procedures declared in the interface use the same names and types as the `global` methods. The C program broadcasts by calling `bcast_<name>`, where `<name>` is the name of the broadcast method in the `global` class. In Bellman-Ford, `bcast_bcastToSource` calls the only bcast method, `bcastToSource`.

The `step` command causes one graph-step then writes to the passed `active` Boolean pointer. The value written to `active` says whether there were any pending messages sent from the last graph-step. In Bellman-Ford, no pending messages mean the iteration has reached a fixed point, so it stops after the first step `active` is false. Instead of `step`, the controller may call `iter`, which steps until quiescence (i.e. there are no active messages). Since negative cycles will cause infinite iteration, the Bellman-Ford controller cannot use `iter`, and must stop after `n` iterations.

When global reduce methods are not used, `step` and `iter` suffice to advance graph-steps. Since each global reduce produces a value at the end of its graph-step, we add versions of `step` and `iter` that output global reduce results. A graph-step which ends with the global reduce method `<gr>` is stepped with the procedure `step_<gr>`. `step_<gr>` sets the active Boolean pointer, like `step`, and also sets a pointer (or pointers in the case of a tuple) to the result of the global reduce. The controller may also call `iter_<gr>`, which advances graph-steps until a step is reached which contains the global reduce method `<gr>`.

3.3 Structural Constraints

The GRAPAL compiler checks constraints that ensure the source program conforms to the GraphStep model and enable compilation to efficient FPGA logic.

- **No-Recursion:** To enable compilation of methods to feed-forward FPGA logic, the function call graph is checked to ensure that there are no recursive cycles.
- **Send-Receive:** For each message dispatch statement, the kind of the sender method and the kind of the receiver method is constrained.
- **Single-Firing:** Conservative, static checks are performed to ensure that there is at most one update operation per node per graph-step, at most one operation per edge per graph-

K_{source}	K_{dest}
`bcast`	`node send`
`node reduce`	`node send`
`node send`	`edge fwd` OR `global reduce`
`edge fwd`	`node reduce`
`global reduce`	

Table 3.1: This table shows structural constraints on message passing. Each method with kind K_{source} is only allowed to send to methods with one of the K_{dest} kinds in its row.

step, and at most one global reduce per graph-step.

GraphStep defines the message passing structure between `bcast`, `node reduce`, `node send`, `edge fwd` and `global reduce` methods. To enforce this structure, the GRAPAL compiler checks that the source program conforms to the **Send-Receive** constraint. For each source method of kind K_{source} that sends to a destination method of kind K_{dest}, the pair K_{source}, K_{dest} must appear in Table 3.1. For each dispatch statement, `<out-set>.<dest-method>(<arguments>)`, the source method contains the dispatch statement and the destination method is `<dest-method>`. Sends from `node reduce tree` to `node send` methods do not need to be checked because they are implicitly specified by giving the pair of methods the same name.

The **Single-Firing** constraint enforces GraphStep's synchronization style which sequences parallel activity into graph-steps: At each node, operations of kind `node send` are sequenced with graph-steps and at each edge, operations of kind `edge fwd` are sequenced with graph-steps. This means at most one `node send` and at most one `edge fwd` operation can fire per object per graph-step. Reduce methods are invoked multiple times to handle multiple messages per graph-step. However, all messages in a graph-step to a `node reduce` at a particular node must invoke the same method so there is at most one value for one `node send` method. Likewise, all messages in a graph-step to a `global reduce` must invoke the same method so there is only one reduce value for the sequential program.

Single-Firing is enforced with two sub-constraints:

- **One-Method-per-Class:** For each class (c), each method kind (k) and each graph-step, all operations at objects in c with kind k invoke the same method. For example, if class `C` has two `node send` methods, `C.M1` and `C.M2`, then `C.M1` and `C.M2` cannot both

fire in the same graph-step.

- **One-Message-per-Pointer:** Each pointer in an `out` set transmits at most one message per graph-step.

To see how **One-Method-per-Class** and **One-Message-per-Pointer** imply **Single-Firing**, each method kind must be considered:

- `node send`: Due to **One-Method-per-Class** all `node send` operations at a particular object have the same method. A `node reduce` or `bcast` produces only one message for each `node send` at each object. Since there is at most one message for at most one `node send`, only one `node send` operation can fire.
- `edge fwd`: Due to **Single-Firing** for `node send`, at most one `node send` fires at each edge's predecessor node. Due to **One-Message-per-Pointer**, this `node send` sends at most one message each edge. Since an edge can receive at most one message, at most one `edge fwd` operation can be invoked at each edge.
- `node reduce`: Since each `node reduce` is followed by one unique `node send` method, if there are two `node reduce` methods of the same class in the same graph-step then there are two `node send` methods of the same class in the same graph-step. Therefore, for **One-Method-per-Class** to hold for `node send`, it must hold for `node reduce`.
- `global reduce`: **One-Method-per-Class** is the only constraint on `global reduce` methods.

One-Method-per-Class is conservatively enforced by first classifying graph-step into *graph-step types*, then checking that at most one method can fire per class per graph-step type. Section 5.1.1.2 explains how the compiler constructs graph-step types, based on the graph of possible message sends in the chain of graph-steps after a `bcast`. **One-Message-per-Pointer** is conservatively enforced by checking that the dispatch statements in each path through a method's control flow graph contain each `out` set at most once (Section 5.1.1.2).

Chapter 4

Applications in GRAPAL

This chapter describes four benchmark applications implemented in GRAPAL: Bellman-Ford (Section 4.1), ConceptNet (Section 4.2), Spatial Router (Section 4.3) and Push-Relabel (Section 4.4). For each application, we show the GRAPAL program and sequential controller in C. Spatial Router and Push-Relabel are fairly complex so we describe how to adapt them to GRAPAL. Each application is tested on a set of benchmark graphs.

Section 4.5 describes the speedups and energy savings of all applications over all graphs compared to sequential applications. For Bellman-Ford and ConceptNet, the sequential algorithms are just sequentially scheduled versions of the parallel algorithms. They both have the same iterative structure as GraphStep with an outer loop that iterates over graph-steps and an inner loop that iterates over active nodes. They each use a FIFO to keep track of the active nodes. For Spatial Router, the sequential program is part of a highly optimized package that solves routing. For Push-Relabel, the sequential program is a performance contest winner for the solving single-source, single-sink Max Flow/Min Cut problem.

4.1 Bellman-Ford

The Bellman-Ford algorithm solves the single-source shortest paths problem for directed graphs with negative edge weights. Given an input graph and source node, it labels every node with the shortest path to it from the source. This algorithm is naturally parallel and does not require any specialization to adapt it to GRAPAL. Bellman-Ford was used for GRAPAL example code in Chapter 3 with GRAPAL code in Figure 3.1 and C code in

Figure 3.6.

We test Bellman-Ford on the circuit-retiming CAD problem. Circuit retiming moves registers in a circuit graph to minimize the critical path [64]. Our circuits are from the Toronto 20 benchmark suite [65]. A circuit graph, or netlist, is a network of primitive logic gates and registers. Retiming works on synchronous logic, which means that the subgraph of logic gates is a Directed Acyclic Graph (DAG). This subgraph of logic gates is the original netlist with all registers and their adjacent edges removed. A netlist's critical path is the maximum over path delays through the logic gate DAG. Moving registers changes critical path by changing the depth of the logic gate DAG. Decreasing the critical path decreases the clock period, thus increasing performance.

We use a simple timing model in which the delay of each logic gate is 1. When the delay of each logic gate is one, Leiserson's [64] generalization of systolic retiming can be used. This retiming algorithm first finds the minimum depth, d_{min} for which a retiming exists. Then it determines the actual registers placement in some retiming with depth d_{min}. To find d_{min}, a binary search is performed over depth d using the procedure `retime_for_depth(`G, d`)`. The graph G represents the netlist being retimed. `retime_for_depth(`G, d`)` weights the edges of graph G so that no negative cycle exists iff there exists a retiming of the netlist with depth less than or equal to d. `retime_for_depth(`G, d`)` then runs Bellman-Ford on the weighted graph to determine whether a negative cycle exists. The register placement that satisfies d_{min} is a simple function of the shortest paths computed by `retime_for_depth(`G, d_{min}`)`.

The graph G represents a netlist with one node for each logic gate, one edge for each wire connecting logic gates and one edge for each chain of registers connecting logic gates. Rather than representing each input pin with a node, all input pins are collapsed into the source node. A single register that connects two logic gates is collapsed to a single edge. A chain of registers connecting two logic gates is also collapsed to a single edge. The weight of each edge, $w(e)$ depends on depth d and on the number of registers collapsed to form it, $r(e)$, so $w(e) = r(e) - 1/d$. `retime_for_depth(`G, d`)` first weights the graph, then runs Bellman-Ford. To extract register placement we use the distance to each node, $D(v)$, computed by `retime_for_depth(`G, d_{min}`)`. Each node v subtracts $\lceil D(v) \rceil$ registers

from its input edges, and adds $\lceil D(v) \rceil$ to its output edges.

Section 4.5 compares performance between GRAPAL and a sequential implementation with Figure 4.3. For both the sequential and the GRAPAL implementation most work is in edge relaxation, with one edge relaxation per active edge per iteration. Compared to the sequential implementation the GRAPAL implementation has extra overhead for node iteration and barrier synchronization. For node iteration, the GRAPAL implementation iterates over each node per iteration while the sequential implementation iterates over only active nodes that were updated on the last iteration. The GRAPAL implementation must perform a barrier synchronization on each iteration, or graph-step. Figure 4.3 sorts graphs from smallest to largest and shows that the largest graphs get greater speedups due to a lower relative overhead.

4.2 ConceptNet

ConceptNet is a knowledge base for common-sense reasoning compiled from a Web-based, collaborative effort to collect common-sense knowledge [23]. Nodes are concepts and edges are relations between concepts, each labeled with a relation-type. An important query on a ConceptNet graph is spreading activation, which is used to find the context of a set of concepts. Spreading activation inputs a set of concepts (`initial`) and a weight for each relation-type (`weights`). It then assigns the input `weights` to edges in the graph according on their types. Each concept in `initial` corresponds to a node in the graph to which it assigns a high activity factor of 1. All other nodes are given an activity factor of 0. Activities factors are propagated through the graph, stimulating related concepts. After a fixed number of iterations, nodes with high activities are identified as the most relevant to the query.

Figure 4.1 shows the GRAPAL kernels for spreading activation, and Figure 4.2 shows the query procedure in the C sequential controller. The main computation is performed by the `node` methods `reduce tree update`, `send update`, and the `edge` method `prop`. The `update` methods are in charge of accumulating incoming activation to add to state and propagate to successor edges. The `edge` method reweights and transmits

activation. The `nextact` function is used to combine activations by the binary reduce and by the activation state update.

All activations, weights and discounts are in the range $[0, 1]$. In order to use FPGA logic efficiently we use fixed-point arithmetic by representing numbers with 1 integer digit and 8 fractional digits. GRAPAL currently does not have a fixed-point type, so we use the type `unsigned<9>` and define `mult_fixed_point` for multiplication.

The sequential controller iterates spreading activation by first broadcasting to nodes with `bcast_start_spreading_activation()` then issuing `step()` commands. Before iterating, the query procedure (`spreading_activation`) must first set the source nodes and set the edge weights. Source nodes are set to have an initial activation of 1 with `set_source` and other nodes are set to 0 with `clear_sources`. Edges are weighted by their relation types by `set_edge_weights`, which is first broadcast to nodes then forwarded to successor edges. After each initializing broadcast command, a `step()` command is issued to perform the graph-step that makes the changes to node and edge state.

To test ConceptNet, we used a small version of the ConceptNet semantic network (`cnet_small`), which has $15,000$ nodes and $27,000$ edges. Our tests run spreading activation for 8 iterations.

Figure 4.3 shows that the speedup of the GRAPAL implementation of ConceptNet over the sequential implementation is 7 times per chip. The relation between the GRAPAL implementation and the sequential implementation of ConceptNet is analogous to the two implementations of Bellman-Ford: For both implementations, most work is for operations on active edges. The sequential implementation keeps a FIFO of active nodes so edges are sparsely active and work is performed for only the active edges. There is overhead for the GRAPAL implementation due to iterating over all nodes, rather than just active nodes, and due to the cost of the barrier synchronization. Larger graphs generally have less relative overhead due to barrier synchronization. The ConceptNet graph is larger than the Bellman-Ford graphs so the ConceptNet speedup is better than the Bellman-Ford speedup.

```
global Glob {
  out Node nodes;

  // query initialization
  bcast clear_sources() nodes.clear_sources;
  bcast set_source(int<20>) nodes.set_source;
  bcast set_edge_weights(boolean, int<6>, unsigned) nodes.set_edge_weights;

  // query computation
  bcast start_spreading_activation() nodes.start_spreading_activation;
}

node Node {
  out Edge edges;

  boolean is_source;
  int<20> idx;
  unsigned<9> discount, act;

  // query initialization
  send clear_sources() {
    is_source = false;
    act = 0;
  }
  send set_source(int<20> set_idx) {
    if (idx == set_idx) is_source = true;
  }
  send set_edge_weights(boolean all_types, int<6> type, unsigned<9> weight) {
    edges.set_weight(all_types, type, weight);
  }

  // query computation
  send start_spreading_activation() {
    if (is_source) {
      edges.prop(one_fixed_point());
      act = one_fixed_point();
    }
  }
  reduce tree update(unsigned<9> act1, unsigned<9> act2) {
    return nextact(act1, act2);
  }
  send update(unsigned<9> more_act) {
    edges.prop(mult_fixed_point(more_act, discount));
    act = nextact(act, more_act);
  }
}

edge Edge {
  out Node to;
  int<6> type;
  unsigned<9> weight;

  // query initialization
  fwd set_weight(boolean all_types, int<6> set_type, unsigned<9> new_weight) {
    if (all_types || type == set_type) weight = new_weight;
  }

  // query computation
  fwd prop(unsigned<9> act) {
    to.update(mult_fixed_point(act, weight));
  }
}

unsigned<9> nextact(unsigned<9> act1, unsigned<9> act2) {
  return act1 + act2 - mult_fixed_point(act1, act2);
}
unsigned<9> one_fixed_point() {
  return ((unsigned<9>) 1) << 8;
}
unsigned<9> mult_fixed_point(unsigned<9> x, unsigned<9> y) {
  return (((unsigned<17>) x) * y) >> 8;
}
```

Figure 4.1: ConceptNet's spreading activation in GRAPAL

```
void spreading_activation(int n_source_nodes, int* source_node_idxs,
    float* default_weight, int n_rel_weights, int* rel_weight_types, float* rel_weights,
    int n_iterations) {
  int i;

  // initialize source nodes
  bcast_clear_sources();
  step();
  for (i = 0; i < n_source_nodes; i++) {
    bcast_set_source(source_node_idxs[i]);
    step();
  }

  // initialize relation weights
  bcast_set_edge_weights(true, -1, to_fixed_point(default_weight));
  step();
  for (i = 0; i < n_rel_weights; i++) {
    bcast_set_edge_weights(false, rel_weight_types[i], to_fixed_point(rel_weights[i]));
    step();
  }

  // run spreading activation iterations
  bcast_start_spreading_activation();
  for (i = 0; i < n_iterations; i++) {
    step();
  }
}
```

Figure 4.2: Sequential controller for ConceptNet's spreading activation in C

4.3 Spatial Router

The Spatial Router routes a netlist on a 2-dimensional FPGA fabric. The FPGA fabric is a graph of 4-LUTs, switches and I/O pads and is taken from the FPGA Place-and-Route Challenge [65]. LUTs are placed in a 2-dimensional mesh bordered by I/O pads. Switches route signals between LUTs and pads, with wires connecting switches to each other and to LUTs and I/O pads. Switches are arranged in horizontal and vertical channels that run between each row and column, respectively, of LUT and pads. The channel width, W, of the architecture is the width of each channel and is proportional to the number of switches.

The netlist to be routed is a set of LUTs and pads that are connected by nets. Each net is used to transmit a single bit from one LUT or pad to a set of LUTs and pads. Before the routing stage, virtual LUTs and pads in the netlist were assigned to LUTs and pads in the FPGA. The job of the router is to allocate paths through switches so they implement the nets. Each net, z, has a source LUT or pad, $src(z)$ and a set of sink LUTs and pads, $sinks(z)$. For each net and each sink, $t \in sinks(z)$, the router must choose one of the possible paths through switches from $src(z)$ to t. Also, nets cannot share switches, which

makes routing a difficult combinatorial optimization problem. The existence of a route for a netlist on a particular topology is NP-Complete. Our router, and the router we compare to, tries to find a route for the FPGA-challenge architecture parameterized by the channel-width, W.

Our router is an adaptation of Huang and DeHon's Stochastic, Spatial Router [66, 67]. The GRAPAL code for the router is in Appendix C. The Stochastic, Spatial Router accelerates routing by adding logic to FPGA hardware so the FPGA can self-route. Logic and state in hardware switches search for acceptable routes in a decentralized fashion. In our router a static graph models the FPGA topology, rather than extra hardware. This graph has a node for each LUT and pad and a node for each switch. Adjacent nodes are connected by a pair of directed edges, with one in each direction. For these routers, one source, sink pair is routed at a time with parallel search over possible paths to each sink.

Spatial Router is based on simple source to sink reachability (Section 2.1.1) but has many heuristics and optimizations that make it much more complex. Its GRAPAL code is 340 lines, and the sequential controller is 180 lines. A simple, non-victimizing, version of the router routes each source, sink pair (s, t) of a net z sequentially, with higher fanout nets first:

- **Search:** The search phase propagates reachability from source node to other nodes in the graph. In each graph-step, nodes on the reachable frontier propagate activation to their successor nodes. Each successor node that is newly reachable propagates on the next-graph step. A global reduce is performed on each graph-step to say if the sink has been reached. Once the sink is reached, graph-step iteration stops. If there is no route to the sink (because too many switches have been allocated to other paths) then then activity will quiesce before a sink is reached. The first path to reach the sink is the shortest one, so this algorithm chooses the shortest available path.
- **Allocate:** Switches in the path found by search are locked so future searches cannot use them. To allocate, messages are sent backwards along the path found by search from the sink to the source. Search recorded the predecessor of each node touched in the reachability propagation so the path can be recovered.

  We add a number of optimizations to the reachability based router:

- **Fanout Reuse:** Switches allocated for the net z should be reused for paths to different sinks in $sinks(z)$. If there is already a path to sink $t_1 \in sinks(z)$, and t_2 is close to t_1 but far from $src(z)$, then t_2 should use a path that branches off of t_1's path. Search with fanout is performed by setting all nodes currently allocate to z as sources. The source node closest to t_2 is then used as the branch point. Allocate is modified so it records the index of z. This way, after a broadcast to all nodes each node knows whether to mark itself as a source for the current net.
- **Victimize:** In the simple algorithm, if the initial search fails to reach the sink then the netlist cannot be routed. A victimizing search is allowed to use switches that were already allocated so the current sink can always be reached. The old path through a reallocated switch must be ripped up, deallocating its use of other switches, and its source, sink pair must be reinserted into a queue of unrouted pairs. Victimizing, ripping up and rerouting is a common technique in routers. Victimizing search and allocation phases are only run if the initial search fails.

  Rip up is performed in the allocation phase by propagating rip-up signals along a victimized path. The allocation signal propagates backwards along the path it is allocation, from the sink to the source. When the allocation signal touches a victimized switch, it sparks rip-up signals to that travel along the old path that was routed through the switch. One rip-up signal travels forward along the old path, towards the sink LUT, and one rip-up signal travels backward along the old path, towards the source LUT. Since rip-up signals are initiated as the allocation signal travels, many can be active at the same time. If two rip-up signals traveling in opposite directions on the same path collide then they cancel out. A forward rip-up signal must branch into multiple rip-up signals when it reaches a switch at which the path branches. A backward rip-up signal will usually stop when it gets to a branching switch so the other paths through the switch are preserved. Each switch has a counter that keeps track of the number of times its path branches. This counter is decremented each time a backward rip-up removes on of its branches, and incremented each time alloc adds a branch. In the case where multiple backward rip-up signals arrive at a branching switch in the same graph-step, a switch compares the current count to the number of input rip-ups. If the number of input rip-ups equals the

current count then the switch sends a rip-up signal back.

- **Congestion Delay:** Victimizing old paths should be discouraged so the router does not get trapped and cycle over mutually victimizing source, sink pairs. Each routed path has a congestion factor that discourages new routes through its switches. This congestion factor is unique to each net and is increased each time one of the net's sinks is ripped up. It starts at 4 and doubles on each rip up. This doubling makes paths more likely to avoid high-congestion regions when they can. Congestion delay is not used in Huang and DeHon [66, 67]. Instead congestion delay is inspired by the standard sequential FPGA router Pathfinder's [68] use of congestion factors in its cost function. In both cases, congestion is used detect hot-spots and encourage routes to avoid them.

  Without a congestion factor, the search algorithm naturally finds the shortest path since each leg takes one graph-step, and the first path to reach the sink is chosen. To implement the congestion factor we reuse this technique by delaying the search frontier wherever it reaches already-used switches. A congestion factor of d simply delays the search by d graph-steps. To delay the search frontier, an already-allocated node sends a wait message on a self edge d times.
- **High-Fanout Locking:** High-fanout nets are routed first and locked so they are never ripped up. The current algorithm locks all nets with fanout at least 20.
- **Coincident Paths:** A new path, P_1, that causes an old path, P_2, to be ripped up only does damage to P_2 the first time it victimizes a switch of P_2's. So P_1 should not be penalized for later victimizations of P_2's switches. Since GRAPAL messages are a fixed size, we cannot keep track of the identifiers for all paths P_1 victimized. Instead, each search signal keeps track of the identifier for the last victimized path, so P_1 is only penalized once for each segment of P_2 it intersects that goes in the same direction. We have not implemented a similar treatment for oppositely aligned coincident segments (which would also use local, distributed node state).

We use netlists from the LGSynth93 Benchmark Set [69] that are small enough to fit on FPGA fabrics that are small enough for our on-chip memories. To compare Stochastic Router's performance to a sequential router, we use VPR 4.30's router [70, 71]. VPR has been highly optimized for place and route and is the standard benchmark for FPGA

router studies. For each netlist, G, before we run VPR we find the minimum channel width routable by Stochastic Route, $W_{min}(G)$. For each G, VPR successfully routes with a channel width of $W_{min}(G)$. We compare the times of the two routers with a fixed channel width of $W_{min}(G)$. VPR is run with the option `-fast`, which increases its speed but makes some netlists not route able. VPR is run with no `router_algorithm` parameter, so timing-driven mode is not used.

Figure 4.3 shows the performance per chip of the GRAPAL implementation of Spatial Router over the sequential implementation is between $1/4$x to 2.5x. Spatial Router does not give large speedups because our current implementation is not parallel enough. Across benchmark graphs, the average number of active edges per graph-step is only 65 to 147. This means that each of the 26 PEs has an average of 2.5 to 6 active edges per graph-step, which is not enough to fill the PE pipeline and diminish the impact of overhead due to barrier-synchronization and node iteration. A more efficient Spatial Router would increase parallelism and increase edge activity by finding routes for multiple sinks in parallel.

The performance of the current implementation suffers by up to 20% due to a bad dynamic load balance: Neighboring switches in the mesh are usually assigned to the same PE to minimize communication traffic (Subsection 7.4). However, neighboring switches' activity is correlated so a majority of operations occurs in one or a few PEs in any particular graph-step. This means work is sequentialized instead of being distributed to many parallel PEs. Performing multiple routes in parallel, that are not localized to the same area of the mesh, would improve the dynamic load balance.

4.4 Push-Relabel

The push-relabel method solves the single-source, single-sink max flow/min cut problem. The input graph is $G = (V, E, s, t, u)$, where V is the set of nodes, E is the set of directed edges, s is the source node, t is the sink node, and u labels each edge with its capacity. The objective is to find the maximum flow through edges from s to t that respects edge capacities. The flow on each edge e is $f(e)$ and $f(e) \leq u(e)$

Goldberg and Tarjan [72, 73] originally developed the push-relabel method. Push-

Relabel proceeds by performing two operations: $push$ is applied to edges and $relabel$ is applied to nodes. The operations are local to edge and node neighborhoods and can be performed on many nodes and edges in parallel. The operation $push(e)$, where $e = (u, v)$, reads and writes e's state, u's state and v's state. The operation $relabel(v)$ reads and writes v's state and reads the state of neighboring nodes and edges. $push$ must be atomic but two $relabel$ operations can occur on neighbors in parallel.

Our implementation of Push-Relabel in GRAPAL is in Appendix B. The main push-relabel algorithm consists of two alternating phases in GRAPAL: In the push phase, all nodes try to `push` in parallel, and in the relabel phase, all nodes try to `relabel` in parallel. Since $push(e)$ is atomic, e can be the only edge neighboring either u or v that is $push$ed in a graph-step. To implement this atomicity, request and acknowledge messages are sent between nodes in the push phase. Each node first uses the `push_request` method to request exclusive access for an edge from neighboring nodes. Each node that receives a request then sends an acknowledgment to one of its requesters, with `push_ack`. Each node that receives any acknowledgments chooses one and propagates the change to the acknowledged edge, with `push_do`. Nodes are labeled with unique priorities to arbitrate these decisions.

In the relabel phase, each node sends relevant state to its neighbors so each neighbor can decide whether to perform $relabel$. Since $relabel(v)$ only reads neighboring state and is not atomic, it does not need to use request and acknowledge to acquire exclusive access to neighbors. The method `relabel_start` sends node state to neighbors, and `relabel_height` inputs and acts on neighbors' state.

Modern, sequential implementations of Push-Relabel utilize heuristics that give large speedups. The most commonly used heuristics are global relabeling and gap relabeling [74]. Global relabeling accelerates relabeling using a shortest path search from the sink to all reachable nodes in the residual flow network. Global relabeling is common in parallel implementations of Push-Relabel [75]. Our implementation uses Bellman-Ford to perform global relabeling. The methods named `global_relabel` in the `Edge` and `Node` classes implement Bellman-Ford. Gap relabeling also accelerates relabeling. Gap relabeling works by inserting and removing nodes from doubly linked lists when their labels

change. We chose not to implement gap relabeling because a straightforward implementation of gap relabeling in GRAPAL would sequentialize relabel operations.

Figure 4.3 shows that the performance of the GRAPAL implementation of Push-Relabel is $1/10$ the sequential implementation. The GRAPAL implementation has low relative performance because it performs many more push and relabel operations than the sequential implementation. This increase in total work counteracts GRAPAL's performing edge operations faster. In the sequential case, at any point in time there are many push and relabel operations that could be applied next. Goldberg and Tarjan's efficient sequential implementation uses a dynamic tree data structure [72] to choose the next push or relabel operation, which decreases the total number of operations. In contrast, the natural scheduling in a graph-step algorithm eagerly performs all pending push operations on the next graph-step. It may be possible to utilize the dynamic tree data structure of Goldberg and Tarjan in GRAPAL since the trees are embedded in the static graph structure. We have yet to see whether a GRAPAL implementation including dynamic trees significantly reduces the required work. Another factor leading to more work performed by the GRAPAL implementation is that it does not perform the gap relabeling heuristic. Gap relabeling uses a dynamic data structure which is mismatched to GRAPAL's static graph data-structure. Although gap relabeling can be adapted to parallel implementations of Push-Relabel (e.g. [76]), it is impossible to implement as a computation on a static graph without sequentializing relabel operations. An efficient GRAPAL algorithm for Push-Relabel would require significantly more work in finding a good operation scheduling technique and a good replacement for gap relabeling.

4.5 Performance

To test the performance of GRAPAL we compare our multi-FPGA GRAPAL implementation to the performance of a sequential processor. The multi-FPGA platform is a BEE3 containing four Virtex-5 XC5VSX95T FPGAs and is described in Section 5.2. The sequential processor used is a single core of the dual core 3 GHz Xeon 5160. Both chips are of the same technology generation, with a feature size of 65 nm. GRAPAL program runs use all optimizations discussed in Chapter 7. In particular, the runtime performs node

decomposition and local assignment.

Figure 4.3 shows the speedup of the GRAPAL version over the sequential version per chip for each benchmark application and each benchmark graph. GRAPAL varies from $1/10$ of the sequential performance to 7 times the sequential performance, with a mean of 2 times. The simple graph applications ConceptNet and Bellman-Ford perform well, especially for larger graphs. Push-Relabel provides no speedup since we have not incorporated heuristics used by the sequential version. For future work our push-relabel algorithm should incorporate techniques that deliver speedups for parallel implementations (such as parallel gap relabeling) that are explored by [75, 76, 77, 78].

We compare the energy used for each run by the GRAPAL implementation and the sequential implementation. To calculate energy use for GRAPAL programs on the BEE3 system and sequential programs on the Xeon system we first measured the current of sample runs of benchmark applications. We used an Amp meter with a resolution of 0.1 Amps. We measured a constant $A_{GRAPAL} = 0.4$ Amps for many sample runs on the BEE3. We measured a constant $A_{Seq} = 1.2$ Amps for many sample runs on the Xeon system. Both systems were plugged into the same AC power supply of $V_{RMS} = 208$ RMS Volts. Watts for the BEE3 are $W_{GRAPAL} = A_{GRAPAL}V_{RMS} = 83$ and Watts for the Xeon system are $W_{Seq} = A_{Seq}V_{RMS} = 250$. The times for a GRAPAL and Sequential run are T_{GRAPAL} and T_{Seq}, respectively. We calculate energy consumption for a particular run as $E = WT$:

$$E_{GRAPAL} = 83T_{GRAPAL} \text{ Joules} \tag{4.1}$$

$$E_{Seq} = 250T_{Seq} \text{ Joules} \tag{4.2}$$

The GRAPAL implementation delivers significantly better energy performance since the BEE3 system of 4 chips uses $1/3$ the power of the sequential system of 1 chip. Figure 4.4 show the relative energy use of the GRAPAL version compared to the sequential version for each benchmark application and each benchmark graph. The mean energy used for GRAPAL is $1/10$ that for sequential, the minimum is $1/80$ and maximum is 82%.

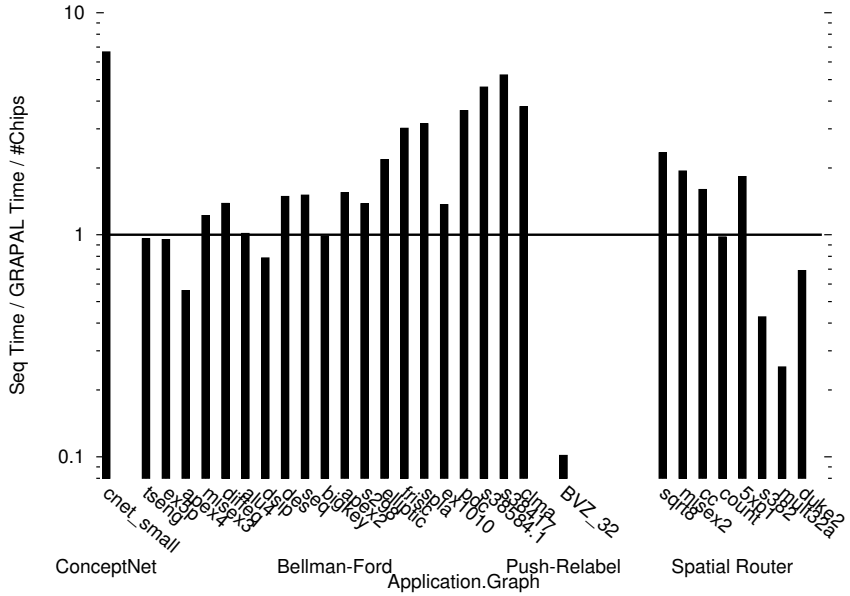


Figure 4.3: The speedup per chip of the GRAPAL implementation on the BEE3 platform over the sequential implementation on a Xeon 5160 for each application and each graph

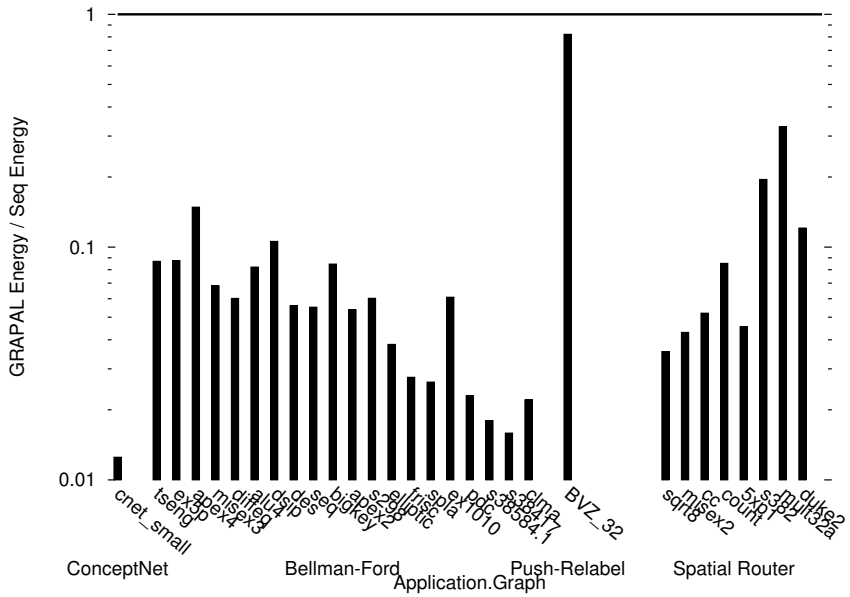


Figure 4.4: Energy use of GRAPAL implementation on BEE3 platform relative to energy use of sequential implementation on Xeon 5160 system, for each application and each graph

Chapter 5

Implementation

This chapter explains the design and implementation of our GRAPAL compiler and our FPGA logic architecture. The platform we target is a BEE3 which has a circuit board of four Virtex-5 XC5VSX95T FPGAs.

5.1 Compiler

The GRAPAL compiler translates the source level language to a logic architecture that can be loaded and run on an FPGA. The *core* of our compiler translates the source program to VHDL modules that describe the logic architecture. The primary pieces of the compiler:

- Perform standard compilation steps: Parse, type check, and canonicalize on an intermediate representation.
- Check that the control flow structure and method and function call structure conforms to the constraints specified in Section 3.3.
- A library that represents HDL makes it easy for the compiler to describe arbitrary, highly parameterized HDL structures.
- Transform source language methods into HDL operators.
- Describe the HDL for the logic architecture (Section 5.2), which is parameterized by data from the intermediate representation of the GRAPAL program.
- Choose the parameters for the logic architecture that affect performance but are not a implied by the GRAPAL program (Chapter 8).
- Manage the FPGA tool-chain which translates the logic architecture described in VHDL

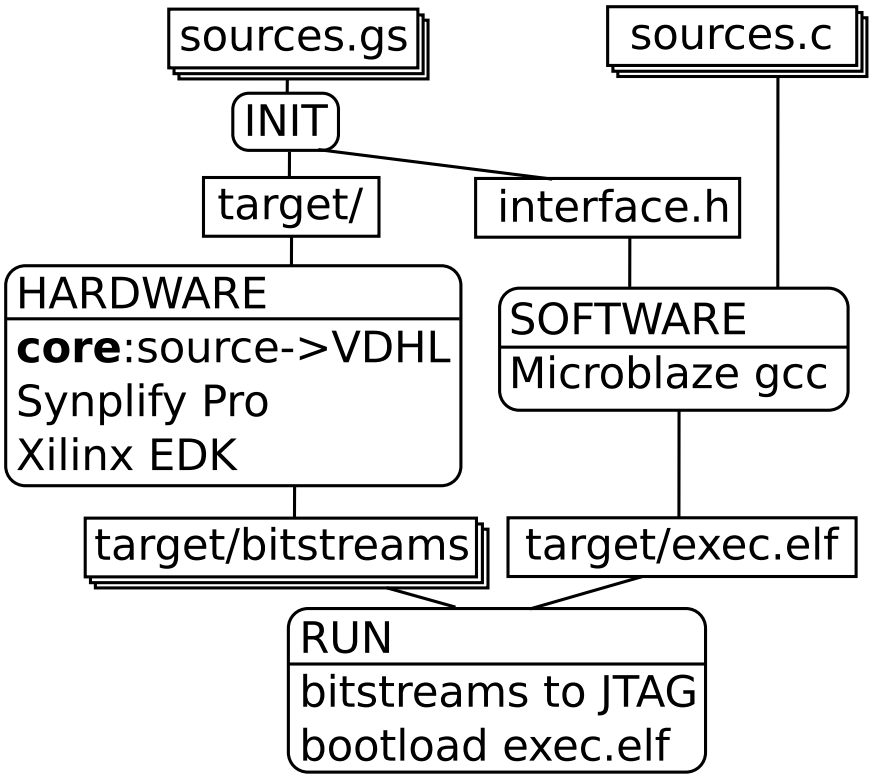


Figure 5.1: Entire compilation and run flow for GRAPAL programs

output by the compiler's backend to bitstreams for execution on FPGAs.

Figure 5.1 shows the entire compilation and runtime flow for GRAPAL programs. `sources.gs` are the parallel kernel code in GRAPAL, and `sources.c` are the sequential controller controller code. The compiler and runner is composed of the stages INIT, HARDWARE, SOFTWARE and RUN. These stages can be run independently, or the three compilation stages, INIT, HARDWARE and SOFTWARE, can be run at once. The INIT stage declares global broadcast and reduce methods in the GRAPAL program as C function signatures in `interface.h` for the sequential controller to call. A programmer typically develops the GRAPAL and C code together, and INIT needs to be a separate stage so it can be part of the development process. INIT also creates a target directory for files used and generated by later stages. The HARDWARE stage performs the core translation of the source program to VHDL logic. HARDWARE then passes the VHDL through the external Synplify Pro and Xilinx EDK CAD tools to generate bitstreams, one for each of the four FPGAs in our BEE3 target platform. The SOFTWARE stage compiles all C code,

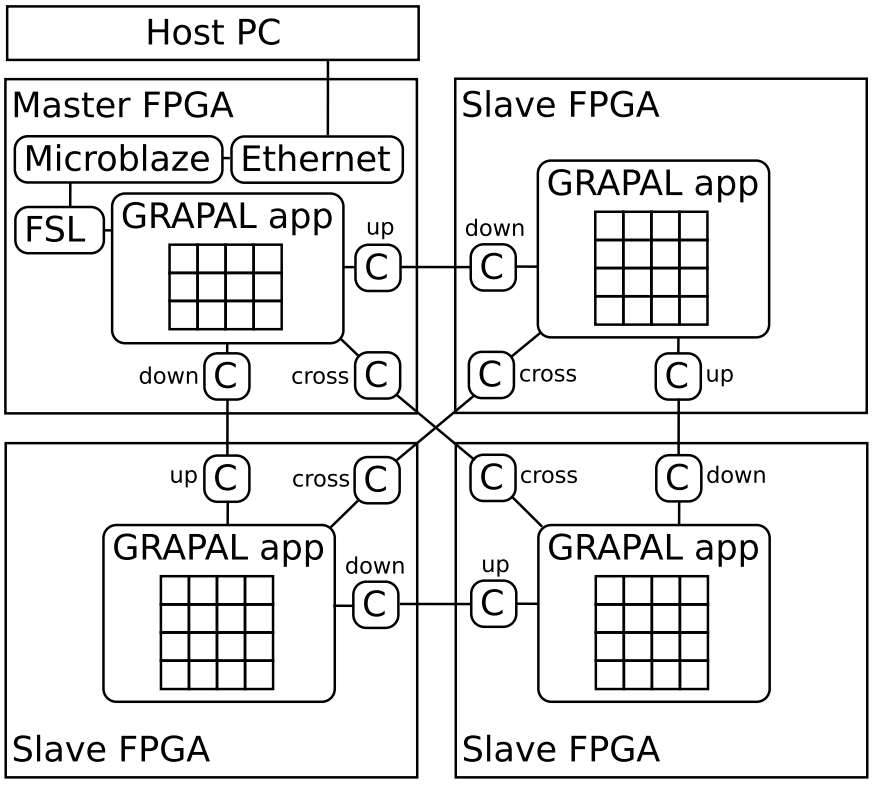


Figure 5.2: Embedded architecture on the BEE3 platform

`sources.c` and `interface.h`, with Xilinx's modified GCC compiler that targets the MicroBlaze soft processor [79]. SOFTWARE is a separate stage to allow the programmer to make changes to the sequential controller without waiting for the CAD tools to recompile the bitstreams. The RUN stage first loads bitstreams onto FPGAs with the JTAG hardware interface, then uses our custom bootloader to load the sequential controller binary onto the MicroBlaze.

5.1.1 Entire Compilation and Runtime Flow

Bitstreams generated by the HARDWARE stage implement the embedded level architecture shown in Figure 5.2. There is one bitstream for each of the four FPGAs in the BEE3 platform. The core logic on each FPGA is the GRAPAL application-specific logic, which consists of PE logic and memory, the packet-switched network and global broadcast and reduce networks. VHDL files describing this application-specific logic is generated by the

core GRAPAL compiler, shown in Figure 5.3. After VHDL generation, Synplify Pro performs logic synthesis to translate each FPGA's top-level VHDL module into a netlist. The netlist for each FPGA is then composed with inter-chip routing channels. The netlist for the master FPGA is also composed with a MicroBlaze soft processor, an Ethernet controller, and FSL queues. Inter-chip communication channels form a cross-bar connecting each pair of FPGAs. Up, Down and Cross logic components, labeled C in Figure 5.2, translate between application logic channels and inter-chip channels. The MicroBlaze processor runs the sequential controller program described by `sources.c`. It is linked to a host PC with Ethernet to load the sequential controller, load and unload graphs, and perform miscellaneous communication. There is one Fast Simplex Link (FSL) queue to send tokens from the MicroBlaze process to application-specific logic, and one FSL queue to send tokens from application-specific logic to the MicroBlaze. The compiler packages each full FPGA design as a Xilinx Embedded Development Kit (EDK) project. The EDK tool then maps, places and routes each EDK project to generate a bitstream for each FPGA. The RUN stage running on the host PC loads bitstreams onto the four FPGAs in sequence via the JTAG hardware interface.

Most standard compiler optimizations are excluded from compilation down to HDL. Instead, we rely on the logic synthesis tool (Synplify Pro), the first step of the FPGA tool chain, to perform constant folding and common subexpression elimination on the HDL level. Since GRAPAL has simple semantics with no loops, recursion, or pointers there is no need to perform other standard optimizations. However, dead code elimination is performed to improve compiler efficiency, and translation to HDL requires all functions to be inlined.

The core of the GRAPAL compiler checks source code and transforms it into VHDL files describing the logic architecture. Figure 5.3 shows the compiler stages between source code and VHDL files. Representations of the program are drawn with sharp corners and checking and transformation steps are listed in boxes with rounded corners. In the canonical intermediate representation used by the compiler, methods and functions are represented as control flow graphs (CFGs) with static single assignment (SSA) [80].

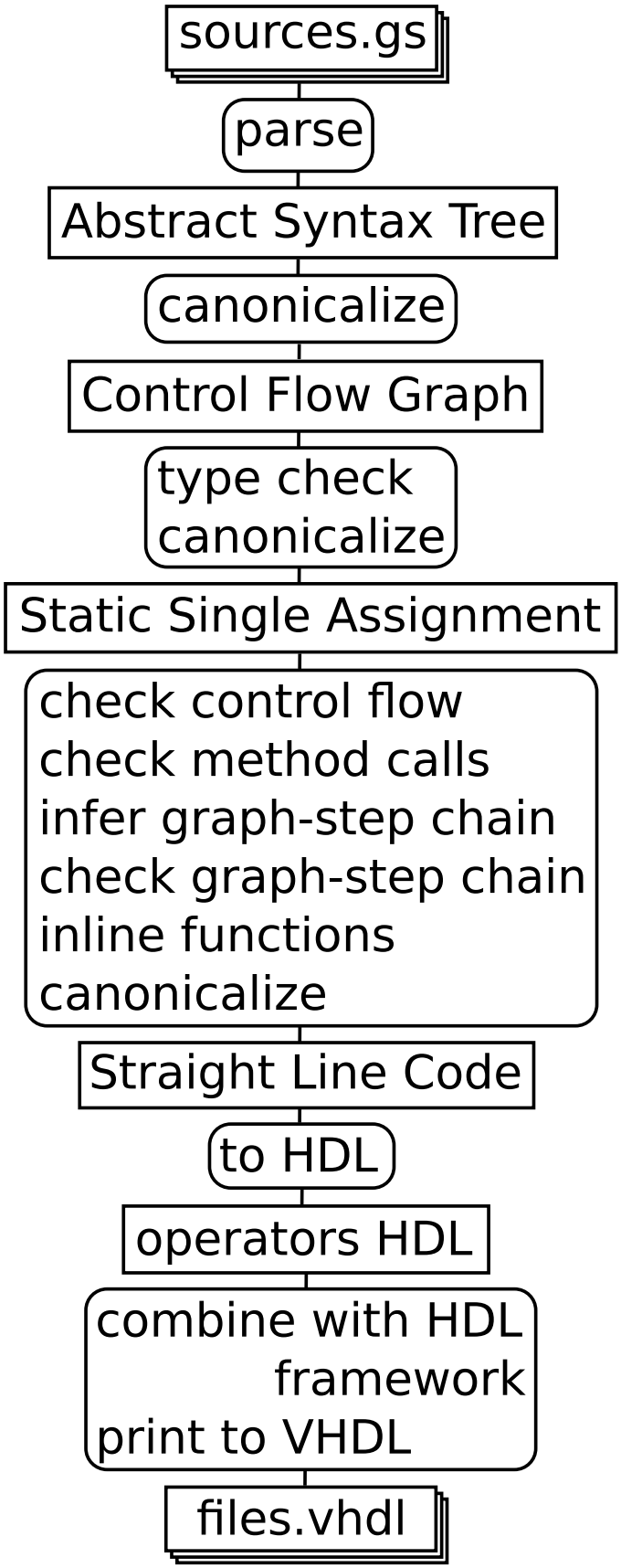


Figure 5.3: The core of the compiler translates source GRAPAL files to VHDL files. This is the first step of the HARDWARE stage (Figure 5.1)

.

5.1.1.1 Translation from Source to VHDL

First the parser converts source files into an Abstract Syntax Tree (AST) and reports syntax and undeclared identifier errors. The control flow in each method and function is transformed from nested control-flow structures and expression trees in the AST to a control flow graph (CFG) [81]. The initial control flow graph box in Figure 5.4 shows an example CFG immediately after the transform from an AST. Each node in the CFG is a basic block, which is straight-line code ending in a control flow transfer. This flat structure is easier to perform transforms on than a nested AST structure. Each CFG has a source basic block, where execution starts, and possibly multiple sink basic blocks where execution ends. Every non-sink ends in a branch or jump to transfer control to another basic block. Sinks end with a return statement in the case of functions and returning methods and a nil statement otherwise.

After type checking, variable declarations and assignments are transformed into static single assignment (SSA) form [80]. In SSA, variables cannot be assigned values after they are declared; each is assigned a value once in its definition statement. This makes the dataflow structure explicit which simplifies transforms and checks on the code. Figure 5.4 shows how assignments in the initial CFG are transformed into definitions with SSA. Multiple assignments to a variable which occur in sequence are transformed into a sequence of definitions of different variables. Each reference to the variable is then updated to the last definition. When a variable which is assigned different values in different paths, a new variable must defined at the point of convergence for any later references to use. The new variable at the point of convergence is defined to be a Φ function of the variables in the convergent paths. At execution time, the Φ function selects the variable on the path that was traversed.

A series of canonicalization steps on control and data flow is performed:

1. Case expressions of the form `Predicate ? ThenExpr : ElseExpr` are transformed to basic blocks.
2. Return statements are unified into a single return statement in a unique sink basic block.
3. All function calls are inlined into ancestor methods. This is possible because the **No-**

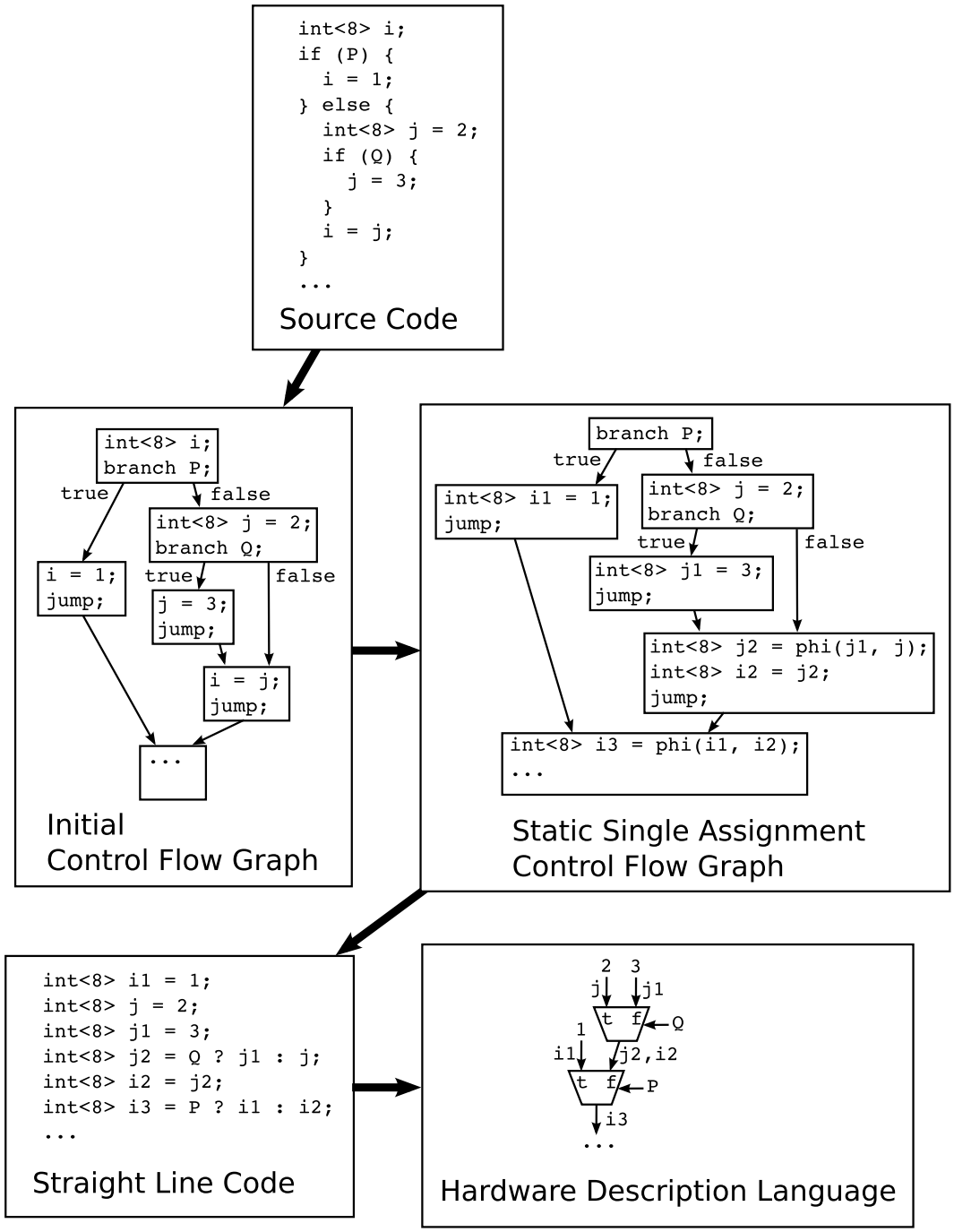


Figure 5.4: The control flow representations used by the compiler after inputting Source Code are: the Initial Control Flow Graph, the Control Flow Graph with Static Single Assignment, Straight Line Code, and Hardware Description Language. Control flow graphs consist of basic blocks with transfer edges. Control out of a branching basic block follows the edge labeled with the Boolean value supplied by the branch's predicate. Static single assignment has Φ (`phi`) functions to select a value based on the input edge control followed. Straight line code uses case expressions to implement Φ functions. Hardware description language substitutes case expressions with multiplexers. The HDL's wires are labeled with the variables from previous stages that they correspond to.

Recursion constraint (Section 3.3) disallows recursion.

4. Multiple message send statements on the same `out` field are unified into a single statement, with all send statements moved to the sink basic block. This is possible because the **One-Message-per-Pointer** constraint disallowed calls on the same `out` field that are on the same control flow graph.
5. All field reads are moved to variable definitions in the source basic block and all field writes are moved to the destination basic block. New variables are introduced when fields are read from after being written to. This will allow object state read to be packed into a single large read at the beginning of operation and write to be packed into a single large write at the end.
6. Tuple expressions are flattened into atomic expressions, tuple types are flattened into atomic types, and tuple references are eliminated.

The last canonicalization step transforms each method's CFG into straight-line code, which is almost equivalent to the HDL description of the method. The CFG is a DAG, so basic blocks can be sequenced with a topological ordering. Case expressions of the form `Predicate ? ThenExpr : ElseExpr` are reintroduced to replace Φ expressions. Figure 5.4 shows how the SSA's Φ functions are transformed into straight-line code's case expressions. For each Φ expression, nested case expressions are constructed whose predicates are the predicates used by branches which are post-dominated by the Φ. Variables input to the Φ become case expressions' then and else expressions. Nested case expressions are once again flattened into a sequence of statements.

The straight-line code for each method is transformed into HDL directly by replacing each case expression with a two-input multiplexer and ignoring the order of statements. The statement order can be ignored because statements SSA never change variables. Each variable declaration is typed as a Boolean or signed or unsigned integer parameterized by width so it can be converted into a wire with the correct width. Each arithmetic operation corresponds directly to an HDL arithmetic operation. Figure 5.4 shows how variables are converted to wires and case expressions are converted to multiplexers.

HDL modules described by the compiler includes the operators generated from GRAPAL methods as well as the entire logic architecture. The compiler uses its own library

for constructing HDL modules. It is more convenient to describe arbitrary, highly parameterized HDL modules with functions and data structures than to rely on externally defined modules in a language like VHDL or Verilog. The Java libraries JHDL [82] and BOOM [83] and Haskell libraries Lava [84] and Kansas Lava [85] take the same approach to supporting highly parameterized structures. Code to print the target VHDL operators can go in a single function in the library, rather than appearing in each part of the compiler that generates HDL. Parameters which depend on the compiled program are pervasive throughout the generated logic. By using a library we can pass these parameters to functions which encapsulate HDL modules, which simplifies the number of changes that need to be made when the meaning of or the set of parameters changes during compiler development. Using ordinary functions to describe complex structures like the network is also more convenient than using VHDL or Verilog functions.

5.1.1.2 Structure Checking

Once method control-flow is in SSA form, checks are performed to ensure that the program conforms to the structural constraints given in Section 3.3.

The **Send-Receive** constraint forces message-passing between methods in a graph-step to conform to the structure displayed in Figure 3.5 (rules are in Table 3.1). The compiler checks dispatch statements to enforce this message-passing structure.

According to the **Single-Firing** constraint at most one `node send` method and at most one `edge fwd` method fires per object per graph-step. **Single-Firing** is guaranteed by the **One-Method-per-Class** constraint and **One-Message-per-Pointer** constraint (Section 3.3).

The **One-Method-per-Class** means that at most one method of each kind in each class can fire in any given graph-step. The compiler proves **One-Method-per-Class** by categorizing dynamic graph-steps into static *graph-step types*. A graph-step type is defined as the set of methods that can possibly fire in a graph-step. If no two methods of the same kind and same class are present in any graph-step type then no two methods of the same kind and same class can fire in any graph-step. At runtime, each chain of graph-steps is initiated with a `bcast` method. The compiler uses the method-call graph to construct a chain of

graph-step types for each `bcast` method in the program. Figure 5.5 shows the compiler's algorithm to construct and check graph-step types. Each successive graph-step type lists the methods which can follow the methods listed in its predecessor. If a graph-step type's methods have no successors then the chain ends and graph-step type construction terminates. If there are successors, but they are identical to a previous graph-step type's methods then the chain loops back to the previous graph-step type and graph-step type construction terminates. Since the method-call graph follows the graph-step structure of method kinds `node send`, `edge fwd`, `global reduce` and `node reduce` (Figure 3.5), each graph-step type must be broken into one phase for each of the four method kinds.

One-Message-per-Pointer states that each `node send` method firing sends at most one message on each of its node's `out` sets, and analogously each `edge fwd` method firing sends at most one message of each of its edge's `out` sets. To prove that there will be at most message on each `out` set the compiler checks dispatch statements in each method's CFG. A violation is reported if there exists a path through the CFG that contains multiple dispatch statements to the same `out` set. This check does allow multiple dispatch statements in different branches of an if statement, for example. The feed-forward structure of GRAPAL methods allows this **One-Message-per-Pointer** check to be performed statically at compile-time. To simplify implementation logic, sends in the same method on the same `out` set must have the same destination method.

No loops and no recursion are allowed to enable compilation to logic which can be pipelined to execute one edge per clock-cycle. Loop syntax is absent from the language definition. **No-Recursion** is checked by constructing a graph of function calls and checking it for cycles. After the **No-Recursion** check all functions can be inlined into their ancestor methods.

5.2 Logic Architecture

The performance of sparse graph algorithms implemented on conventional architectures is typically dominated by the cost of memory access and by the cost of message-passing. FPGAs have high on-chip memory bandwidth to stream graph data between memory and op-

```
// Check each chain of graph-step types starting at each global broadcast.
checkOneMethodPerClass(program)
  for each method m ∈ methods(program)
    if kind(m) = bcast
      checkGraphStepType({}, {bcast})

// Recursively construct chain of graph-step types and check each phase
// of a graph-step type. A call to checkGraphStepType checks the four
// phases following lastPhase. There is one phase for each method kind:
//   node send, global reduce, edge fwd and node reduce.
checkGraphStepType(stepsChecked, lastPhase)
  nodeSendPhase = successorPhase(nodeSend, lastPhase)
  globalReducePhase = successorPhase(globalReduce, nodeSendPhase)
  edgeFwdPhase = successorPhase(edgeFwd, nodeSendPhase)
  nodeReducePhase = successorPhase(nodeReduce, edgeFwdPhase)
  graphStepType = (nodeSendPhase, globalReducePhase,
                   edgeFwdPhase, nodeReducePhase)
  // No two methods in the same class with the same kind can appear in
  // a graph-step type.
  for each phase in graphStepType
    if ∃ m_1, m_2 ∈ phase : m_1 ≠ m_2 ∧ class(m_1) = class(m_2)
      raise violation exception
  // The chain ends with an empty graph-step type or
  // a repeated graph-step type.
  if graphStepType ≠ ({}, {}, {}, {}) ∧ graphStepType ∉ stepsChecked
    checkGraphStepType(stepsChecked ∪ {graphStepType}, nodeReducePhase)

// Union method successors to make phase successors.
successorPhase(kind, phase)
  union m ∈ phase
    {n in successorMethods(m) : methodKind(n) = kind}
```

Figure 5.5: The compiler checks **One-Method-per-Class** by constructing a chain of graph-step types for starting at each global broadcast. Each phase of a graph-step type corresponds to one of the method kinds: `node send`, `edge fwd`, `global reduce` and `node reduce`. The method call graph, represented by the successorMethods function, is used to construct graph-step types.

erators and a high on-chip communication bandwidth to enable high throughput message-passing. Compared to a 3 GHz Xeon 5160 dual-core processor, the Virtex-5 XC5VSX95T FPGA we use has 5 times greater memory bandwidth and 9 times greater communication bandwidth (Table 1.1). Further, by using knowledge of the graph-step compute structure and the particular GRAPAL application's data widths, the logic architecture can be customized to perform one operation per clock cycle. Logic for communication between parallel components can be customized for the styles of signals needed to implement GRAPAL applications: A packet-switched network can implement high-throughput message-passing and dedicated global broadcast and reduce networks can implement low-latency barrier synchronization.

Parallel graph algorithms usually need to access the entire graph on each graph-step. In a graph-step, all active node and edge state needs to be loaded and possibly stored resulting in little temporal memory locality. This makes it difficult to utilize caches to hide high memory latency and low memory bandwidth. To achieve high bandwidth and low latency we group node and edges operators and node and edge memories into Processing Elements (PEs). Each PE supports all nodes and edges. Each operator has direct, exclusive access to the memory storing the nodes or edges it acts on. Typically, each PE is much smaller than an FPGA so our logic architecture fills up the platform's FPGAs with PEs, resulting in an average of 41 PEs across applications (Table 5.1). Message-passing between PEs allows communication without having to provide shared memory access between PEs. Since most sparse graphs have many edges per node, the quantity our logic architecture minimizes is memory transfer work per edge. Our benchmark graphs have between 1.9 and 4.8 edges per node with an average of 3.6. Table 5.2 shows the number of edges per node for each benchmark graph. Our logic architecture utilizes dual ported BlockRAMs to provide one edge load concurrent with one edge store per cycle.

Message-passing can be a performance bottleneck due to a large overhead for sending or receiving each message, or too little network bandwidth. In particular we would like to avoid the large overhead for each message typical to clusters. Our logic architecture is customized to the GRAPAL application so one message is input or output per clock-cycle. Graph algorithms generate one message for each active edge. Since the logic can

Application	N_{PEs}	N_{PEs}/N_{chips}
ConceptNet	51	13
Bellman-Ford	51	13
Push-Relabel	26	7
Spatial Router	37	9

Table 5.1: Number of PEs and number of PEs per FPGA

Application	Graph	$\|V\|$	$\|E\|$	$\|E\|/\|V\|$
ConceptNet	cnet_small	15000	27000	1.87
Bellman-Ford	tseng	1000	3800	3.59
	ex5p	1100	4000	3.76
	apex4	1300	4500	3.55
	misex3	1400	41000	3.55
	diffeq	1500	5300	3.54
	alu4	1500	5400	3.55
	dsip	1400	5600	4.12
	des	1600	6100	3.84
	seq	1800	6200	3.54
	bigkey	1700	6300	3.70
	apex2	1900	6700	3.56
	s298	1900	61000	3.60
	elliptic	3600	13000	3.50
	frisc	3600	13000	3.59
	spla	3700	14000	3.74
	ex1010	4600	16000	3.50
	pdc	4600	17000	3.76
	s38584.1	6400	21000	3.23
	s38417	6400	21000	3.33
	clma	8400	30000	3.63
Push-Relabel	BVZ_128	870	2800	3.18
	BVZ_64	1700	7000	4.06
	BVZ_32	3500	17000	4.79
Spatial Router	sqrt8	1400	13000	9.59
	misex2	1600	16000	9.62
	cc	1600	16000	9.62
	count	2600	26000	9.70
	5xp1	2600	28000	10.79
	s382	31000	43000	10.86
	mult32a	7400	81000	10.95
	duke2	7800	98000	12.47

Table 5.2: Number of nodes, edges and edges per node for each benchmark graph

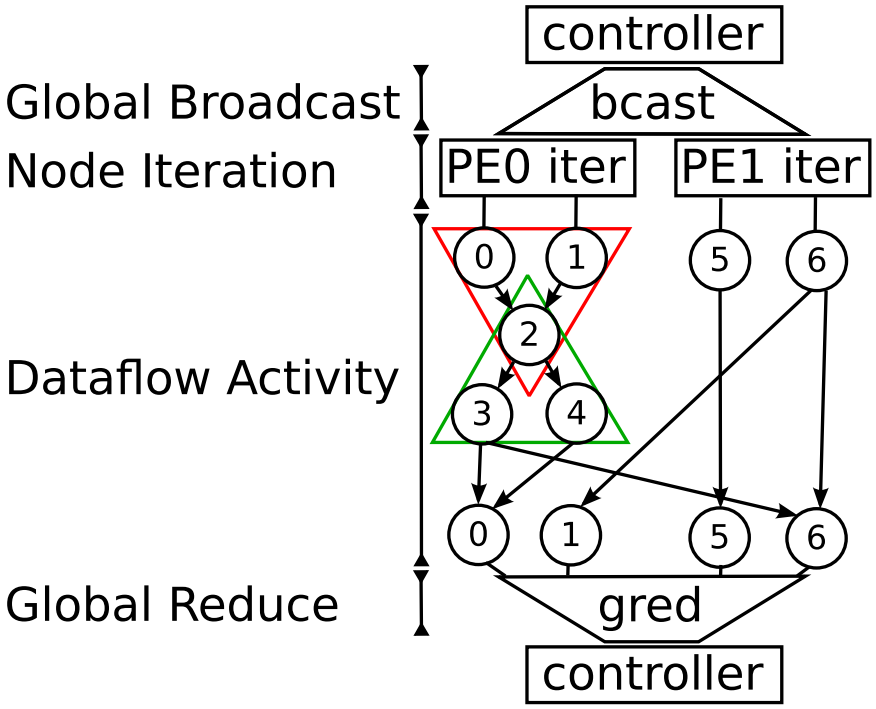


Figure 5.6: Computation structure of a graph-step is divided into four phases. In Global Broadcast the sequential controller broadcasts to all PEs. In Node Iteration each PE iterates over nodes assigned to it and initiate operations on them. PE0 initiates operations on its nodes 0 and 1, and PE1 initiates operations on its nodes 5 and 6. In Dataflow Activity, locally synchronized operations fire on nodes and edges and pass messages between PEs. First fanin nodes fire and send messages (nodes 0 and 1), then root nodes fire (nodes 2, 5 and 6), then fanout nodes fire (nodes 3 and 4), and finally node reduce operations on fanin-tree leaves fire (nodes 0, 1, 5 and 6). The red triangle is a fanin tree and the green triangle is a fanout tree. In Global Reduce, the global reduce values and quiescence information is accumulated to give to the sequential controller.

be customized to each application, message send and receive logic is pipelined to handle one message per cycle. Using knowledge of method input sizes the compiler generates datapaths that are large enough to handle one message per cycle. Graph algorithm implementations on conventional architectures must pack multiple, small, edge-sized values into large messages to amortize out message overhead. This is difficult since graphs structures are often irregular and the set of active edges on each graphs-step is a difficult-to-predict subset of all graph-edges. Our specialized PE architecture is an improvement over an equivalent PE implementation for a conventional, sequential architecture. Figure 5.8 shows an assembly language implementation of a PE that requires 30 instructions per edge.

The logic architecture implements a graph-step with the following four phases (Figure 5.6):

- **Global Broadcast:** The sequential controller broadcasts instructions to all PEs on a dedicated global broadcast network. The broadcast instructions say which GRAPAL methods are active in the current graph-step. If the graph-step is the first one after a `bcast` call by the user-level sequential program then the global broadcast also carries the value broadcast.
- **Node Iteration:** Each PE iterates over its nodes to initiate node firings.
- **Dataflow Activity:** Operations in PEs on nodes and edges act on local state and send messages. Messages travel over the packet-switched network and are received by other operations which may send more messages.
- **Global Reduce:** A dedicated global reduce network accumulates values for the current `global reduce` method and detects when operations and messages have quiesced. Once the global reduce network reports quiescence, the sequential controller can proceed with the next global broadcast. The network accumulates a count of message receive events and message send events so that quiescence is considered reached when the send count equals the receive count.

5.2.1 Processing Element Design

The entire PE datapath (Figure 5.7) is designed to stream one message in and one message out per cycle. This means that in each clock-cycle a PE can perform one edge-fwd operation concurrent with sending a message to a successor edge. Knowledge of the structure of graph-step computations allows us to fuse edge-fwd, node-reduce, node-update, and global-reduce methods together in the PE datapath. An implementation of a more generic actors language (Section 2.4.1) or concurrent object-oriented language would have to send a message between each pair of operations. Knowledge of the GRAPAL application widths allows the width of each bus in the datapath to be matched to the application. Memory word-widths are sized so a whole node or edge is loaded or stored in one cycle.

The mostly linear pipeline shown in Figure 5.7 is oriented around nodes. First nodes

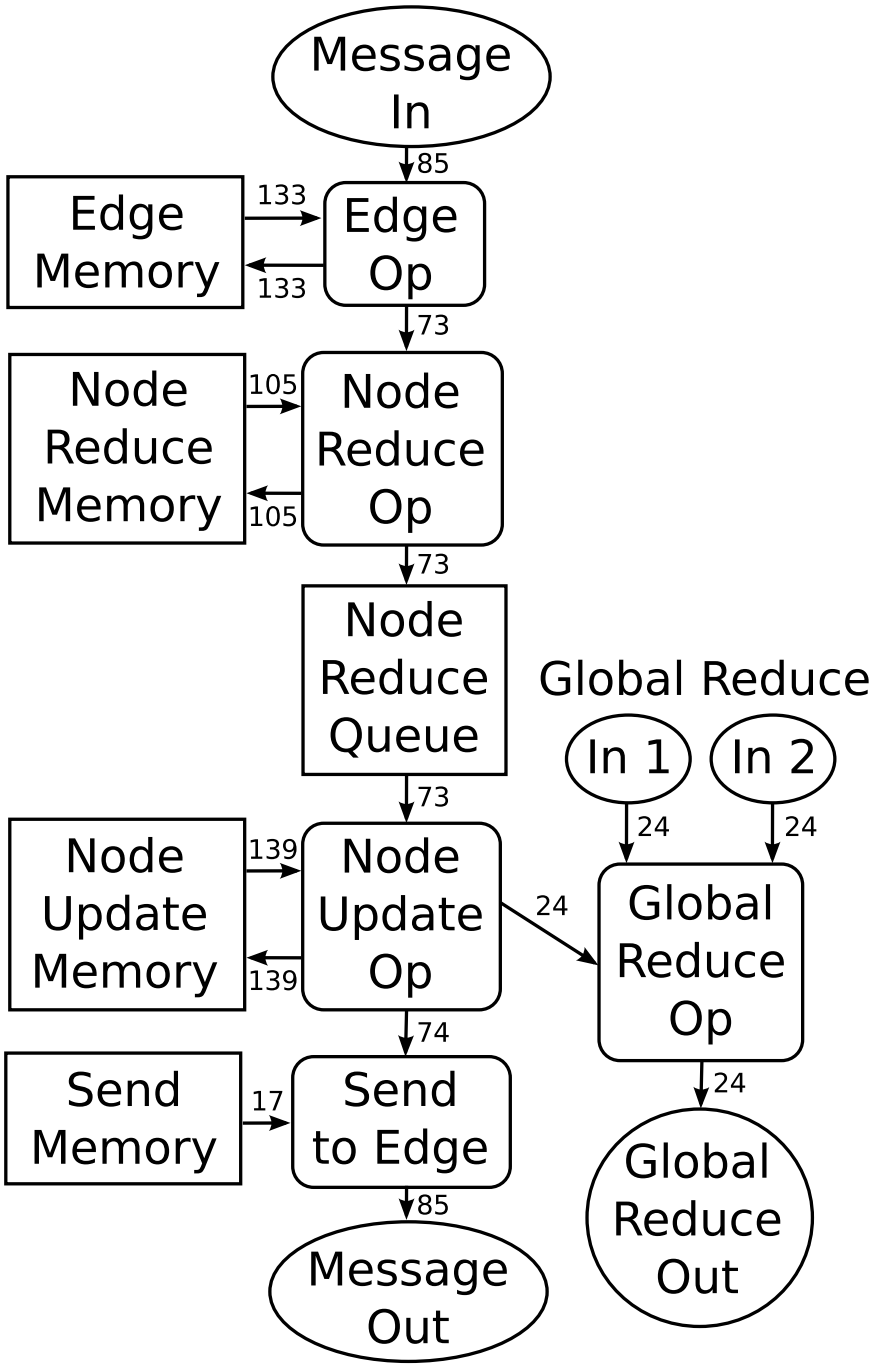


Figure 5.7: Processing element datapaths. Each channel is labeled with the width of its data in bits for the Push-Relabel application.

```
Message receive phase:
  First read message from receive_buffer,
  then execute edge_op and node_reduce_op.
for msg_idx = 0 to receive_count - 1
  msg = &(receive_buffer[msg_idx])
  edge_idx = msg->edge_idx
  msg_val = msg->val
  edge = &(edge_mem[edge_idx])
  edge_state = edge->edge_state
  dest_node_idx = edge->dest_node
  edge_op_out = edge_op(edge_state, msg_val) // edge_op
  edge_state_new = edge_op_out->edge_state
  edge_word->edge_state = edge_state_new
  edge_return = edge_op_out->return
  node = &(node_mem[dest_node_idx])
  // reduce_valid is false only for the first node_reduce_op for a node
  reduce_valid = node->reduce_valid
  reduce_val = node->reduce_val
  reduce_out = node_reduce_op(reduce_val, edge_return)
  reduce_val_new = reduce_valid ? reduce_out : edge_return
  node->reduce_val = true
  node->reduce_val = reduce_val_new

Message Send Phase:
  For each node, first execute node_update_op
  then write messages to send_buffer.
send_buffer_idx = send_buffer_base
for node_idx = 0 to nnodes - 1
  node = &(node_mem[node_idx])
  reduce_valid = node->reduce_valid
  if reduce_valid then
    node->reduce_valid = false
    node_state = node->node_state
    reduce_val = node->reduce_val
    node_op_out = node_update_op(node_state, reduce_val)
    send_valid = node_op_out->send_valid
    if send_valid then
      send_val = node_op_out->send_val
      send_base_idx = node->send_base_idx
      send_bound_idx = node->send_bound_idx
      for send_idx = send_base_idx to send_bound_idx - 1
        send_struct = &(send_mem[send_idx])
        dest_address = send_struct->dest_address
        edge_idx = send_struct->edge_idx
        msg = &(send_buffer[send_buffer_idx])
        send_buffer_idx = send_buffer_idx + 1
        msg->dest_address = dest_address
        msg->edge_idx = edge_idx
        msg->val = send_val
```

Figure 5.8: A PE is described as a sequential program in pseudo assembly. Each C-style statement corresponds to one MIPS instruction, except loops which take two instructions. We simplify the GraphStep operators `edge_op`, `node_reduce_op`, and `node_update_op`, so they each take one instruction. Each of the 30 statements colored green is executed once for each active edge.

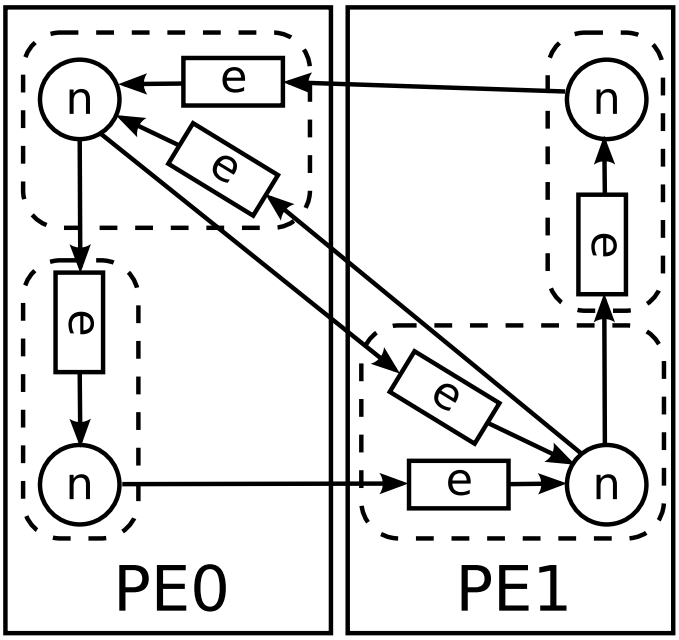


Figure 5.9: Nodes (n) and edges (e) are assigned to PEs by first grouping nodes with their predecessor edges, then assigning each group to a PE. This example shows four nodes grouped with their predecessor edges. The four groups are then split between two PEs.

are grouped with their predecessor edges and assigned to PEs (Figure 5.9). For each node assigned to a PE some of its state is stored in Node Reduce Memory and some is stored in Node Update Memory. Edge state for each edge which is a predecessor of one of the PE's nodes is stored in Edge Memory. Each node points to successor edges, and these pointers are stored in Send Memory. Each memory in Figure 5.7 has an arrow to its associated operator to represent a read. An arrow from the operator to the memory represents a write. Each channel of the PE is labeled with the width of its data in bits for the Push-Relabel application. Since the datapath widths are customized for the application, these widths vary between applications.

The Node Update Operator contains and selects between the GRAPAL application's `send` methods, which occur in the update phase of a graph-step. Figure 5.10 shows the contents of the Node Update Operator. This operator contains logic for each `send` method in each `node` class in the GRAPAL application. It must input method arguments and the firing node's address, load node state, multiplex to choose the correct method, store resulting node state, and output data for sent messages. The compiler checks that at most

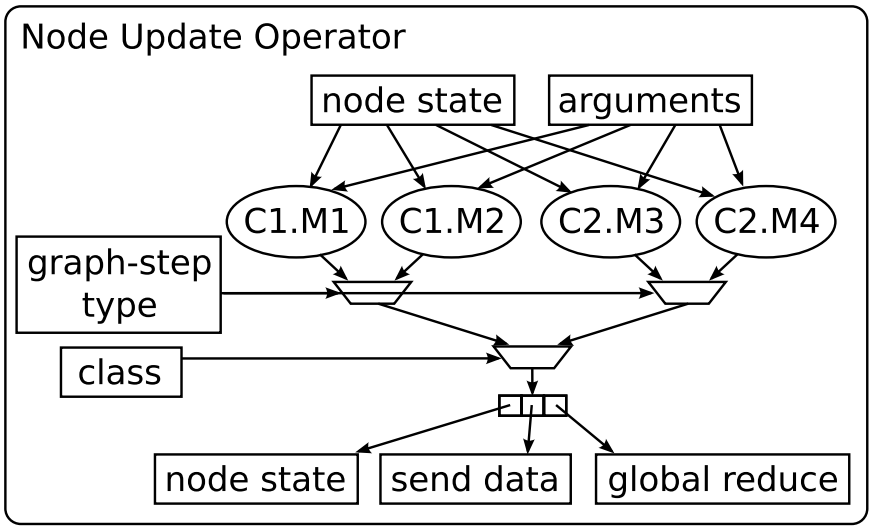


Figure 5.10: Node Update Operator containing the logic for four `node send` methods: `C1.M1` and `C1.M2` in class `C1`, and `C2.M3` and `C2.M4` in class `C2`. Each method inputs node state and arguments and outputs node state, send data for messages, and a global reduce value. Multiplexers are used to select which method fires on each cycle. The methods in each `class` are selected based on the current graph-step type for the currently executing graph-step. A class identifier, which was read from Node Memory along with node state, selects which class's method fires.

one method of each class can fire in any graph-step type (Section 5.1.1.2), which allows us to select the method as a function of the node's class and the graph-step type. The graph-step type is provided by the global broadcast at the beginning of each graph-step. The node class is stored in Node Update Memory along with node state. The Node Reduce Operator wraps `nodes`' `reduce tree` methods, the Edge Operator wraps `edges`' `fwd` methods, and the Global Reduce Operator wraps `global reduce tree` methods. Similar to Node Update Operator, these operators input node or edge addresses along with method arguments, output the result of the method, and possibly load and store to their associated memories. An Edge Operator chooses from possible methods from graph-step type and edge class, Node Reduce Operator chooses from graph-step type and node class, and Global Reduce Operator chooses from graph-step type only. The Send to Edge Operator inputs message data from a node firing and outputs a message for each of the nodes' successor edges.

At the beginning of a graph-step, PE control logic iterates over all nodes assigned to

the PE. Each active node iterated over feeds a token containing the node reduce result from the previous graph-step containing to the Node Reduce Queue. After this initial node iteration, all inter-operator data elements are wrapped as self-synchronized tokens. Tokens carry data-presence and tokens' channels are backpressured so control and timing of a channel's source operator is decoupled from control and timing of the destination operator. The packet-switched, message-passing network has data presence and back-pressure so messages are considered tokens. This pervasive token-passing makes all computation and communication in a graphs step between the initial node iteration and the final global reduce asynchronous.

Without asynchronous operation firing, a single locus of control (per PE) must iterate over potentially active elements. For example, an *input-phase* could iterate over a PE's input edges controlling firing for Edge Operator, Node Reduce Operator and Node Update Operator. Next, an *output-phase* would iterate over pointers to successor edges generating messages to be stored at their destination PEs for the next inputs-phase. These two sub-phases per graph-step would waste cycles on non-active edges, with each edge located in a PE using 2 cycles per graph-step. Since there are between 1.9 and 4.8 edges per node in our benchmark graphs, it is critical for performance to iterate over nodes only as our token-synchronized design does. Critical path latency is also increased since an extra global synchronization is required to ensure that all messages have arrived from output-phase before input-phase can start.

The initial iteration over nodes assigned to a PE fetches the result of the previous graph-step's Node Reduce Op from the Node Reduce Memory and inserts it into the Node Reduce Queue. These reduce results can only be used after the barrier synchronization dividing graph-steps because the number of messages received by a node is a dynamic quantity. Tokens from Edge Op to Node Reduce Op are blocked until the iteration finishes so they do not overwrite node reduce results from the previous graph-step. Although each token-passing channel functions as a small queue, the large Node Reduce Queue has enough space for a token corresponding to each node assigned to the PE. This large queue is required to prevent bufferlock due to tokens filling up slots in channels and the network. Node Update Op inputs the result of the reduce and the node state stored in Node Update Memory to

a `send` method. The `send` method may generate data for the global reduce, handled by Global Reduce Op, and may generate data for messages to successor edges. Global Reduce Op accumulates values generated by Node Update Op in its PE along with values from at most two neighboring PEs (on ports Global Reduce in1 and in2). Since global reduce methods are commutative and associative, the order of accumulation does not matter. Once the global reduce values arrive from neighboring PEs and all local nodes have fired, the PE sends the global reduce value on Global Reduce Out. Each node points to a sequence of successor edges in Send Memory and requires one load to send each message through the port Message Out. The packet-switched, message-passing network routes each message to the Message In port at the destination PE. Once a message arrives Edge Op fires, reading state from Edge Memory and possibly updating state. For each edge fired, Node Reduce Op fires to execute a `node reduce tree` method. Node Reduce Op accumulates a value for each node which is stored in Node Reduce Memory. Node Reduce Op is pipelined so Read After Write hazards exist between different firings on the same node. Hazard detection is required here to stall tokens entering Node Reduce Op.

5.2.1.1 Support for Node Decomposition

The decomposition transform (analyzed in Section 7.2) impacts PE design by allowing PEs to have small memories and by relying on PE subcomponents' dataflow-style synchronization. If PEs can have small memories then we can allocate a large number of PEs to each FPGA with minimal memory resources per PE. When a node is assigned to a PE it uses one word of Edge Memory for each of its predecessor edges and one word of Send Memory for each of its pointers to a successor edge (Figure 5.9). This prevents PEs with small memories from supporting high-degree nodes. Many graphs have a small number of high-degree nodes, so small PEs could severely restrict the set of runnable graphs. In our benchmark graphs, the degree of the largest node relative to average node degree varies from 1.3 to 760 with a mean of 185. By performing decomposition, memory requirements per PE are reduced by 10 times, averaged across benchmark applications and graphs (Section 7.2). Computation cost in a PE is one cycle per edge so breaking up large nodes allows load-balancing operations as well as data. The speedup from decomposition is 2.6, as explained

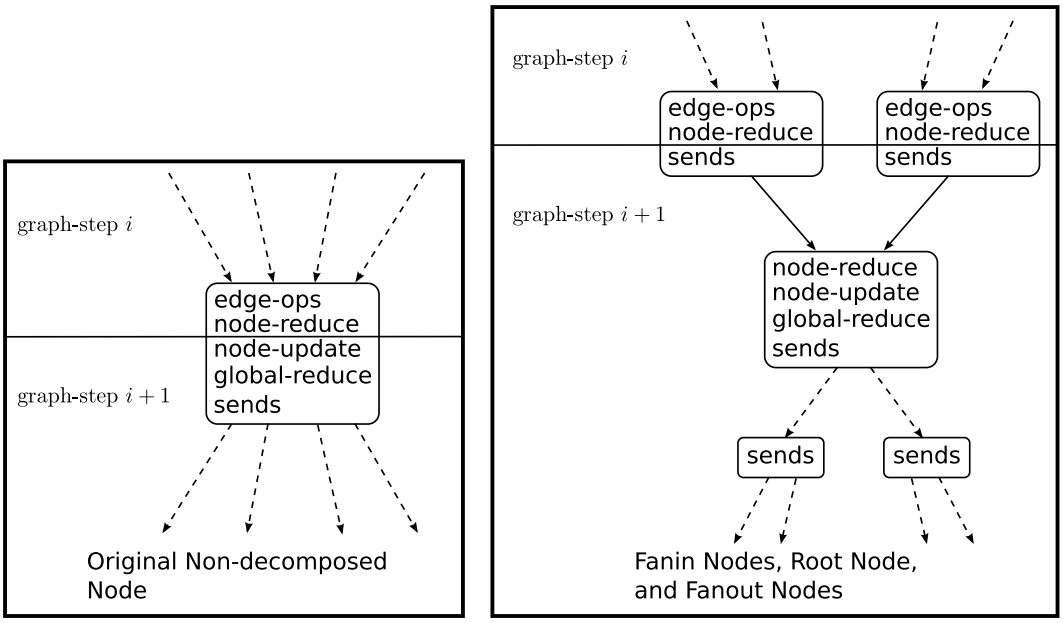


Figure 5.11: An original node with 4 input messages and 4 output messages is decomposed into 2 fanin nodes, 1 root node, and 2 fanout nodes. The global barrier synchronization between graph-steps cuts operations at the original node. For the decomposed case, the global barrier synchronization only cuts operations at leaves of the fanin tree. A sparse subset of the arrows with dashed lines carry messages on each graph-step. All arrows with solid lines carry one message on each graph-step.

in Section 7.2.

Figure 1.5 shows how decomposition breaks each *original node* into a *root node* with a tree of *fanin nodes* and a tree of *fanout nodes*. Each fanout node simply copies its input message to send to other fanout nodes or edges. Fanin nodes must perform GraphStep's node reduce operation to combine multiple messages into one message. We are only allowed to replace a large node with a tree of fanin nodes because GraphStep reduce operators are commutative and associative.

Figure 5.11 shows how decomposition relates to barrier synchronization and which PE operators (Figure 5.7) occur in which types of decomposed nodes in our PE design. After decomposition only fanin-trees leaves wait for the barrier synchronization before sending messages. Other nodes are dataflow synchronized and send messages once they get all their input messages. We make the number of messages into each dataflow synchronized node a constant so the node can count inputs to detect when it has received all inputs. Each fanout

node only gets one message so it only needs to count to one. Without decomposition, all edges are sparsely active, so their destination nodes do not know the number of messages they will receive on each input. To make this number constant, edges internal to a fanin tree are densely active. Dense activation of fanin-tree messages requires that each fanin-tree leaf sends a message on each graph-step. At the beginning of a graph-step, each PE iterates over its fanin-tree leaves to send messages that are stored from the previous graph-step's node reduce and send nil messages for nodes which have no stored message.

We place the barrier immediately after node reduce operators to minimize memory requirements for token storage. Whichever tokens or messages are cut by the barrier need to be stored when they arrive at the end of a graph-step so they can be used at the beginning of the next graph-step. If the barrier were before node reduce operators then memory would need to be reserved to store one value for each edge. We instead store the results of node reduce operators since they combine multiple edge values into one value. These tokens are stored in Node Reduce Memory (Figure 5.7). This node reduce state is not visible to the GRAPAL program, so moving node reduce operations across the barrier does not change semantics.

One alternative logic design would perform one global synchronization for each stage of the fanin and fanout tress so all edges are sparsely active. Another alternative logic design would perform no global synchronizations and make all edges densely active. In Section 7.3 we show that our mixed barrier, dataflow synchronized architecture has 1.7 times the performance of the fully sparse style, and 4 times the performance of the fully dense style.

To make our decomposition scheme work, extra information is added to PE memories and the logic is customized. Each node and edge object needs to store its decomposition-type so the logic knows how to synchronize it and knows which operations to fire on it. We were motivated to design the PE as self-synchronized, token-passing components to support dataflow-synchronized nodes. If all operations are synchronized by a barrier then a central locus of control can iterate over nodes and edges, invoking all actions in a pre-scheduled manner. To handle messages and tokens arriving at any time, a PE's node and edge operators in Figure 5.7 are invoked by token arrival.

Application	W_{msg}	W_{flit}	$\lceil W_{msg}/W_{flit} \rceil$
ConceptNet	30	30	1.0
Bellman-Ford	40	30	2.0
Push-Relabel	90	30	4.0
Spatial Router	40	30	2.0

Table 5.3: This table shows the datapath widths of messages, widths of flits, and flits per message for each application.

5.2.2 Interconnect

The logic architecture's interconnect routes messages between PEs. Figure 5.9 shows how nodes are mapped to PEs. All inter-PE messages are along active edges, from `node send` methods to `edge` methods. Since the set of active edges in any graph-step is dynamic, route scheduling is performed dynamically by packet-switching. Packet-switched messages contain their destination PE and destination edge address, along with payload data. The interconnect topology has two stages, with the top stage for inter-FPGA messages and the bottom stage for intra-FPGA messages (Figure 5.12). The inter-FPGA network connects the four FPGAs in the BEE3 platform with a crossbar. Each intra-FPGA network uses a Butterfly Fat-Tree to connect PEs on the same FPGA and connect PEs to the inter-FPGA stage.

PE datapaths for messages are wide enough for all address and data bits to transmit one message per cycle. Messages routed in the interconnect are split into multiple flits per message with one flit per cycle. Flits make interconnect datapaths smaller which prevents network switches from taking excessive logic resources and allows applications with wide data-widths to route messages over the fixed-width inter-FPGA channels. Table 5.3 includes the size of messages before being broken into flits, and the size of flits for each application. One flit per message is equivalent to not using flits. The flit-width is automatically chosen by the compiler (Section 8.2). Self messages, whose source PE is the same as the destination PE, are never serialized into flits. Not serializing self messages is significant for throughput since, with node placement for locality, a significant fraction of messages are self messages. The fraction of self-messages varies from 0.15 to 0.7 with an mean of 0.4 for benchmark applications.

The inter-FGPA network connects each FPGA in the BEE3 to the other three FPGAs. Each message can go clockwise *up* the four-FPGA ring, counterclockwise *down* the ring, or *cross* the ring. Figure 5.2 shows the embedded architecture on the four FPGAs. Each logic component labeled with C connects the on-chip network to I/O pins. FPGAs are connected with circuit board traces. Each C component bridges between an internal channel, clocked at the application logic frequency, and an external channel, with a DDR400 format providing a data rate of 400 MHz. Tested applications usually have an application logic frequency of 100 MHz, so to match internal and external bandwidth, an internal channel's words are larger than an external channel's words. Up and down channels have 36 bits in each direction at DDR400 and cross channels have 18 bits in each direction. C components resize words of internal streams to match the bandwidth offered by the external channels.

The on-chip network connects PEs on the same chip to each other and to the external channels (Figure 5.12). Each FPGA connects to the three others in the up (clockwise), down (counterclockwise) and cross directions. PEs are connected to each other with a Butterfly Fat-Tree (BFT) and a bridge of splitters and multiplexers connects the BFT to the external channels. Each of the C_{top} channels coming up from the BFT is routed to each of the three up, down and cross directions.

We use Kare's [86] architecture for switches, which are composed of split and merge components. Each of the m components in the Bridge consists of a splitter and a merge. A splitter directs each input message on a single input channel to one of multiple outputs. Splitters from the BFT to external channels use destination PE addresses to direct each message to the correct FPGA. Splitters from external channels to the BFT choose output channels in a round-robin order with a preference for uncongested channels. A merge merges messages on multiple input channels to share the single output channel. Each splitter has a buffer on its outputs so if one destination channel is congested, it will not block messages to other destinations until the buffer fills up.

Although Figure 5.12 shows only one channel for each external direction, in general there can be multiple channels for each external direction. The number of external channels depends flit-width and frequency, which are application dependent. The C_{top} channels coming from the BFT are evenly divided between C_{ext} external channels so each external

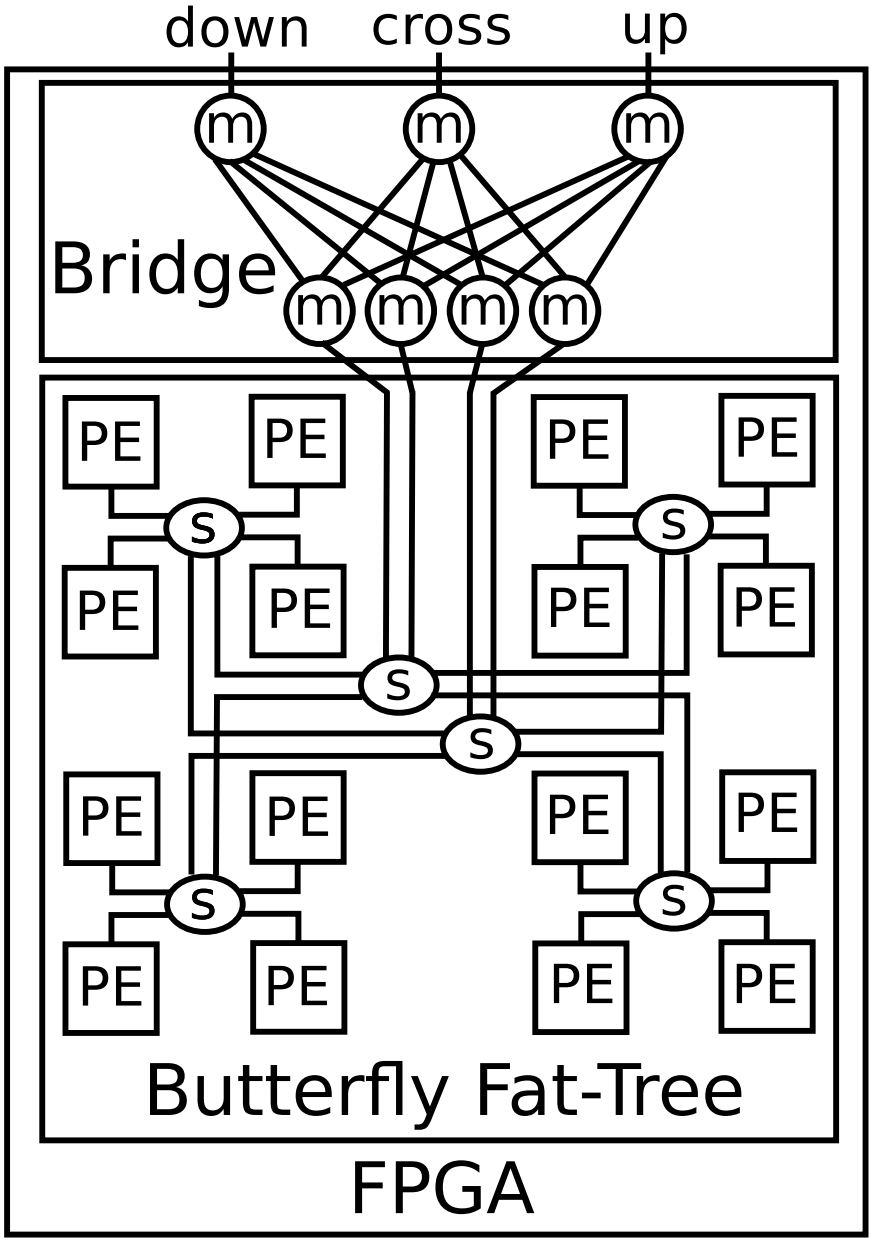


Figure 5.12: The on-chip network contains a Butterfly Fat-Tree (BFT) connecting PEs and a bridge connecting the BFT with inter-chip channels. Drawn channels are bidirectional.

channel is shared $\lfloor C_{top}/C_{ext} \rfloor$ or $\lceil C_{top}/C_{ext} \rceil$ times.

5.2.2.1 Butterfly Fat-Tree

The Butterfly Fat-Tree network connects PEs on the same chip to each other and to the bridge to inter-chip channels. A BFT with a rent parameter of $P = 0.5$ was chosen for its efficient use of two-dimensional FPGA fabric and for simplicity. A mesh network also efficiently uses two-dimensional fabric but has more complex routing algorithms and is more difficult to compose into an inter-FPGA network. To compose a BFT into an inter-FPGA network the top-level switches are simply connected up to other FPGAs.

The $P = 0.5$ BFT is constructed recursively by connecting four child BFTs with new top-level 4-2 switches. Figure 5.12 shows a two level BFT. Each 4-2 switch (Figure 5.13) has four bottom channels and two top channels. The address of each message from a bottom channel into a 4-2 switch determines whether the message goes to one of the other three bottom channels or a top channel. The address of a message from a top channel determines which bottom channel it goes to. The two top channels are chosen in a round-robin order with a preference for uncongested channels.

In general, BFTs are parameterized around their rent parameter P, with $0 \leq P \leq 1$. P determines the bandwidth between sections of the BFT: If a subtree has n PEs then the bandwidth out of the subtree is proportional to n^P. With $P = 0.5$ the bandwidth out of a subtree is proportional to $n^{1/2}$, which is the maximum bandwidth afforded by an $n^{1/2}$ size perimeter out of a region of n PEs on a two-dimensional fabric. This match to two-dimensional fabric makes the $P = 0.5$ BFT area-universal: network bandwidth out of a subregion per unit area is asymptotically optimal.

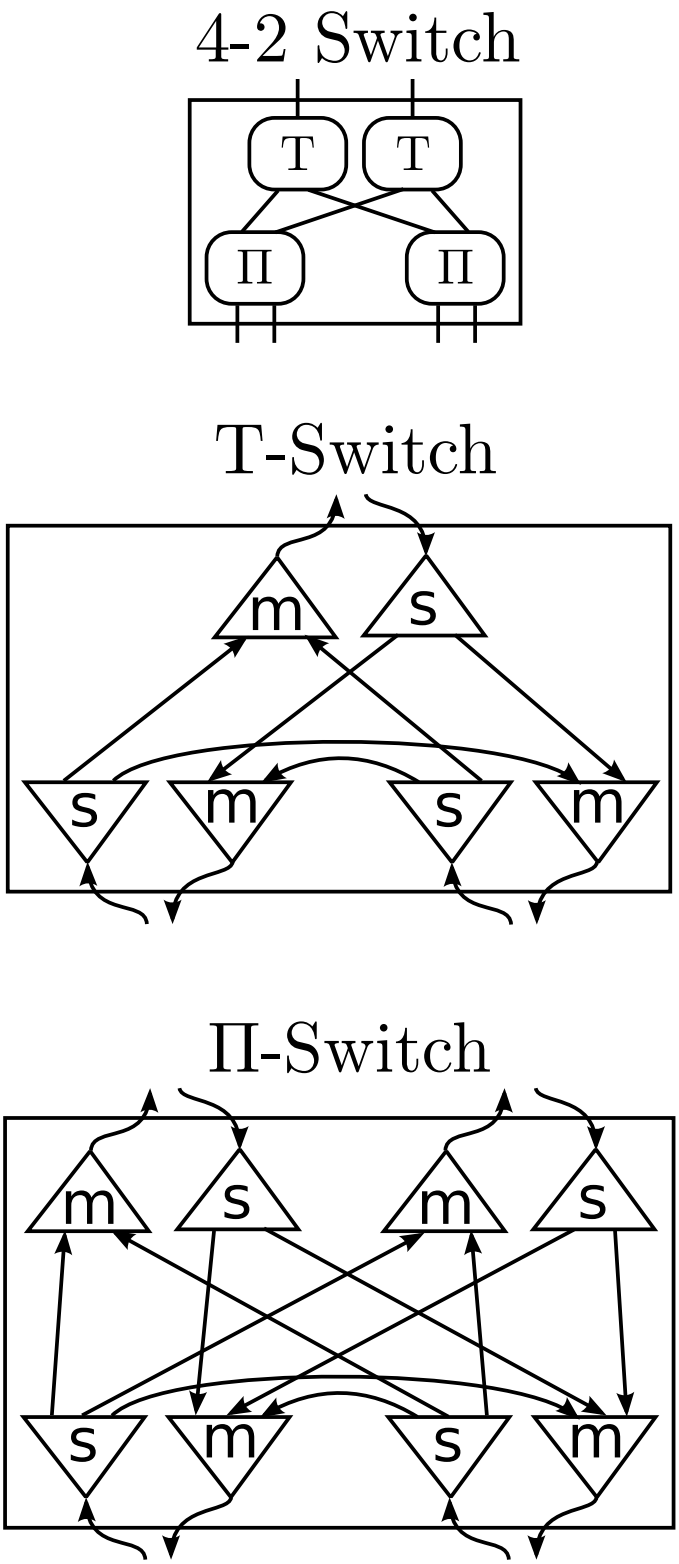


Figure 5.13: Each $P = 0.5$ Butterfly Fat-Tree node is a 4-2 switch. The 4-2 switch has two T-switches and two Π-switches. The T-switch has three 2-to-1 merges and three 1-to-2 splitters, where each splitter directs a message based on its address. The Π-switch has two 2-to-1 merges, two 1-to-2 splitters, two 3-to-1 merges, and two 1-to-3 splitters. Each of the 3-to-1 splitters decides whether to route a message in one of the two up directions or in down based on its address, and it routes messages going in the up direction to the least congested switch. All splitters in both T- and Π-switches have buffers on their outputs to prevent congestion in one output direction from blocking messages going to other directions.

Chapter 6

Performance Model

We developed a performance model for GraphStep to inform decisions about the logic architecture and runtime optimizations. In general, the performance model could be used to estimate performance for a new platform to see if it is likely to be worth the implementation effort to target that platform. The performance model helps us understand where the throughput bottlenecks are and which components of the critical path are most significant. Section 7.1 shows which components of the critical path are important to optimize, and what the effect of each optimization is on each component rather than just on total runtime. Section 7.3 explores the benefit of alternative synchronization styles using the performance model. Since it is difficult to instrument FPGA logic, it is especially important to have a good performance model when targeting FPGAs. We use our GRAPAL implementation on the BEE3 platform to supply concrete values for modeled times. Section 6.2 discusses the accuracy of the performance model when used to measured application runtimes on the BEE3 platform. The mean error runtime predicted by the model, compared to actual runtime, is 11%.

The GraphStep performance model approximates total runtime by summing time over graph-steps. The time to perform each graph-step (T_{step}) is a function of communication and computation operation latency along with throughput costs of communication and computation operations on hardware resources. Components of the time are latencies (L_*), throughput-limited times (R_*), and composite times that include both L_* components and R_* components. Our performance model can be thought of as an elaboration of BSP [5] (Section 2.4.3) for GraphStep. Like BSP, the time of a step is a function of latency and

throughput components, and the significant pieces are computation work on processors (w in BSP), network load (h in BSP), network bandwidth (g in BSP), and barrier synchronization latency (l in BSP).

6.1 Model Definition

Graph-step time is the sum of the time taken by the four subphases: global broadcast, node iteration, dataflow activity, and global reduce (Section 5.2). Figure 6.1 illustrates the work done by the four subphases.

$$T_{step} = L_{bcast} + L_{nodes} + T_{dataflow} + L_{reduce}$$

1. L_{bcast} is the latency of the global broadcast from the sequential controller to all PEs (Section 6.1.1).
2. L_{nodes} is the time taken to iterate over logical nodes assigned to each PE to initiate node firings (Section 6.1.2). Each PE starts iterating once it receives the global broadcast. It takes one cycle per node for each node stored at the PE, including nodes which are not fired initially. In the example (Figure 6.1) there are two PEs which each initiate activity in their fanin-tree leaf nodes.
3. $T_{dataflow}$ is the time taken by the dataflow-synchronized message passing and operation firing activity that was initiated by the node iteration (Section 6.1.3). This includes the time taken by node and edge operations and messages in fanin and fanout trees until the final operations before the end of the graph-step.
4. L_{reduce} is the time taken by the global reduce from all PEs to the sequential controller (Section 6.1.1). The global reduce network is used for both detecting message and operation quiescence. Although the global reduce network is also used to for the high-level `gred` methods' global reduce, only quiescence detection is on the critical path of a graph-step. Since quiescence detection works by counting message sends and receives, the increment from the last message receive, through the global reduce network, to the

	Fanin Leaf	Roots
AVG	0.97	0.97
MIN	0.91	0.92
MAX	1.00	1.00

Table 6.1: Fraction of nodes that are fanin-tree leaves and root nodes across all benchmarks graphs. A node with no fanin-tree nodes is both a fanin-tree leaf and a root.

sequential controller is on the critical path.

N_{PEs} is the number of processing elements. The graph is a pair of nodes and edges: $G = (V, E)$. The set of nodes assigned to PE i is V_i, so $|V| = \sum_{i=1}^{N_{PEs}} V_i$. The function $pred : V \rightarrow E$ maps nodes to their predecessor edges and $succ : V \rightarrow E$ maps nodes to the successor edges.

6.1.1 Global Latency

Our logic architecture dedicates specialized communication channels for global broadcast and global reduce. A global broadcast signal is first generated by control logic, then serialized, then sent to the destination FPGA, then deserialized, and then is forwarded between neighboring PEs in the on-chip mesh. The critical-path latency of a global broadcast from the controller to the farthest PE is:

$$L_{bcast} = L_{bcast\_control} + L_{interchip} + 2L_{bcast\_deserialize} + L_{mesh} \tag{6.1}$$

1. $L_{bcast\_control}$ is the time between the controller deciding to start a graph-step and the broadcast word reaching the serializer. Serialization time is included since it takes only one cycle for the serializer to begin sending bits after it receives the broadcast word.
2. $L_{interchip}$ is the latency for a bit to travel from the source FPGA to another FPGA.
3. $L_{bcast\_deserialize} = W_{bcast}/2$ is the time for the deserializer to receive all bits of the broadcast word. It must receive all W_{bcast} bits before it can pass the full word on to PEs. The logic architecture serializes global broadcast and reduce words to 2 bits per cycle, resulting in $W_{bcast}/2$ cycles total. A coefficient of 2 appears next to $L_{bcast\_deserialize}$ in

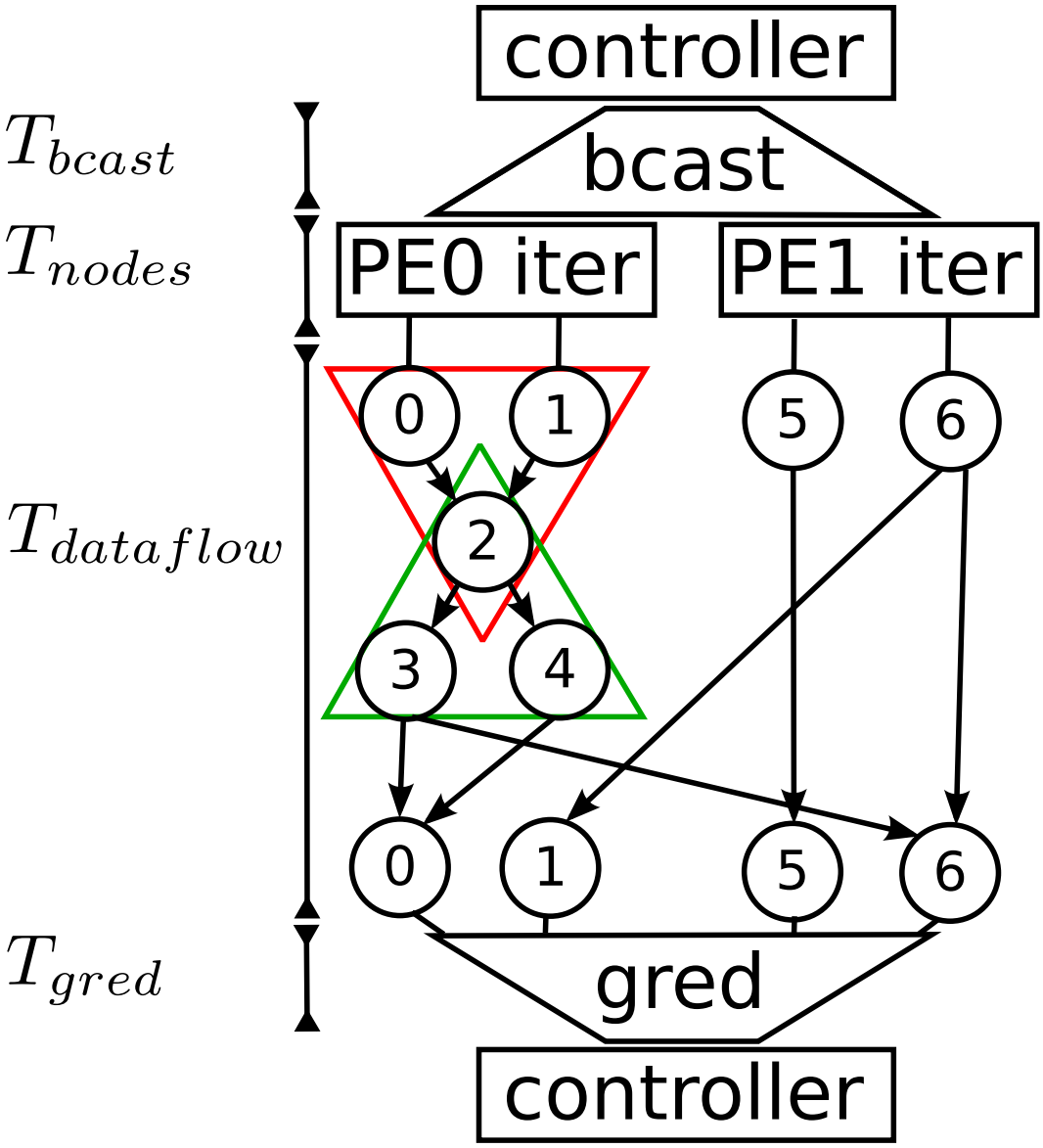


Figure 6.1: Computation structure of a single graph-step

Application	L_{bcast}	$L_{bcast\_control}$	$L_{interchip}$	$L_{bcast\_deserialize}$	L_{mesh}
ConceptNet	108	4	32	56	16
Bellman-Ford	100	4	32	48	16
Push-Relabel	114	4	32	66	12
Spatial Router	132	4	32	80	16

Table 6.2: Latency of global broadcast on the BEE3 platform for each application

Application	L_{reduce}	$L_{reduce\_control}$	$L_{interchip}$	$L_{reduce\_deserialize}$	L_{mesh}
ConceptNet	66	4	32	14	16
Bellman-Ford	66	4	32	14	16
Push-Relabel	74	4	32	26	12
Spatial Router	76	4	32	24	16

Table 6.3: Latency of global reduce on the BEE3 platform

the total latency, L_{bcast}, because a done signal is sent from the controller to PEs at the end of each graph-step. This done signal must finish serializing and deserializing before the next graph-step can start its global broadcast. Effectively, this means the controller must wait an extra $L_{bcast\_deserialize}$ cycles after it receives the global-reduce and before starting the next graph-step.

4. $L_{mesh} = 2(\text{width} + \text{height} - 1)$ is the latency for the broadcast word to pass through the mesh of PEs. It takes 2 cycles for each vertical hop, then 2 cycles for each horizontal hop.

$L_{bcast\_control}$ and $L_{interchip}$ are constant across applications, but $L_{bcast\_deserialize}$ and L_{mesh} are application dependent since they depend on W_{bcast} and PE count, respectively. Table 6.2 shows the cycle counts for each component of L_{bcast} for each application.

Global reduce latency has analogous components in reverse:

$$L_{reduce} = L_{mesh} + L_{interchip} + L_{reduce\_deserialize} + T_{reduce\_control} \tag{6.2}$$

Unlike broadcast, $L_{reduce\_deserialize}$ is only counted once. Table 6.3 shows the cycles counts for each component of L_{reduce} for each application.

6.1.2 Node Iteration

Once a PE receives the global broadcast signal it iterates over its nodes to initiate operation firing and message passing activity. In the first graph-step following a `bcast` call from the sequential controller, activity is initiated with root node firings. On successive graph-steps activity is initiated with fanin-tree leaf firings. For simplicity, in our implementation, each PE iterates over all of its nodes including nodes which are not fired. Each node takes one cycle, so L_{nodes} is the maximum of node count over all PEs: $L_{nodes} = \max_{i=1}^{N_{PEs}} |V_i|$. Using a cycle for each node in $|V_i|$ could be wasteful if a significant number of nodes do not fire. For a graph-step following a `bcast` each root nodes fires but each fanin node and fanout node use a cycle without firing. For successive graph-steps each fanin-tree leaf node fires but every other node uses a cycle without firing. Table 6.1 shows that the large majority of nodes in our benchmark-graphs fire in each node iteration: the mean number of nodes which are roots is 96% and the mean number of nodes which are fanin-tree leaves is 95%. Note that a node with no fanin-tree nodes is both a fanin-tree leaf and a root.

6.1.3 Operation Firing and Message Passing

Node firings initiated by node iteration cause further operation firing and message passing. As shown by Figure 6.1, activity flows up fanin-trees to root nodes, then down fanout-trees, then along edges, until finally node reduce operators at fanin-tree leaves fire. This activity is dataflow-synchronized, where each node after the fanin-tree leaves fires as soon as it gets one message from each of its predecessors. Similarly, each edge fires as soon as it gets an input message. This dataflow style requires only a local count at each node, which allows multiple stages of message passing to be performed without incurring the cost of global synchronization.

The time for the dataflow computation is:

$$T_{dataflow} = \max(L_{dataflow}, R_{net}, R_{ser})$$

	$\frac{L_{bcast}}{T_{step}}$	$\frac{L_{reduce}}{T_{step}}$	$\frac{L_{nodes}}{T_{step}}$	$\frac{T_{dataflow}}{T_{step}}$
mean	0.28	0.18	0.21	0.34
min	0.12	0.07	0.07	0.14
max	0.41	0.26	0.54	0.52

Table 6.4: Breakdown of T_{step} into each of the four stages of a graph-step aggregate across all applications and graph

1. $L_{dataflow} = \sum_{k=1}^{D} L_{PE} + M_k$: $L_{dataflow}$ is the latency of the critical path through nodes and edges. The length of the path in terms of operation firings is D. $D = D_{in} + D_{out}$ where D_{in} is the depth of the fanin tree and D_{out} is the depth of the fanout tree. The kth hop includes the latency of node and edge operations inside a PE (L_{PE}), and the latency of the messages sent through the packet-switched network to the next PE (M_k). Message latency depends on the distance between the source PE and the destination PE.
2. $R_{net} = \max_{j=1}^{S}\{N_{flit} \times N_{switch}^{(j)}\}$: R_{net} is the throughput cost of passing messages on their network switch resources. The jth switch routes $N_{switch}^{(j)}$ messages in the current graph-step, each using $N_{flit} = W_{msg}/W_{flit}$ cycles. R_{net} is the lower bound calculated by counting the number of cycles that each switch is occupied by a message and performing useful work.
3. $R_{ser} = \max_{i}^{N}\{\max(N_{sends}^{(i)}, N_{recvs}^{(i)})\}$: R_{ser} is the throughput cost of processing message sends and receives as well as node and edge operation firings. PE logic can concurrently input one message per cycle and send one message per cycle. The throughput lower-bound for messages at each PE is then the maximum over input messages and output messages. R_{ser} is the maximum of this lower-bound over all PEs. There is one edge operation per message receive, and at least one message send per node operation. Operations do not add extra cycles to R_{ser} since they have their own dedicated logic that acts concurrently with message sends and receives.

The mean times taken by the four graph-step stages, L_{bcast}, L_{reduce}, L_{nodes}, and $T_{dataflow}$, as a fraction of total T_{step} are 0.28, 0.18, 0.21, and 0.34, respectively, (Table 6.4). These times are over all applications and graphs and use the optimizations in Chapter 7. None of these times dominates the rest so we should not focus all optimization effort on one stage. None of the stages has time so insignificant that it is not useful to optimize the

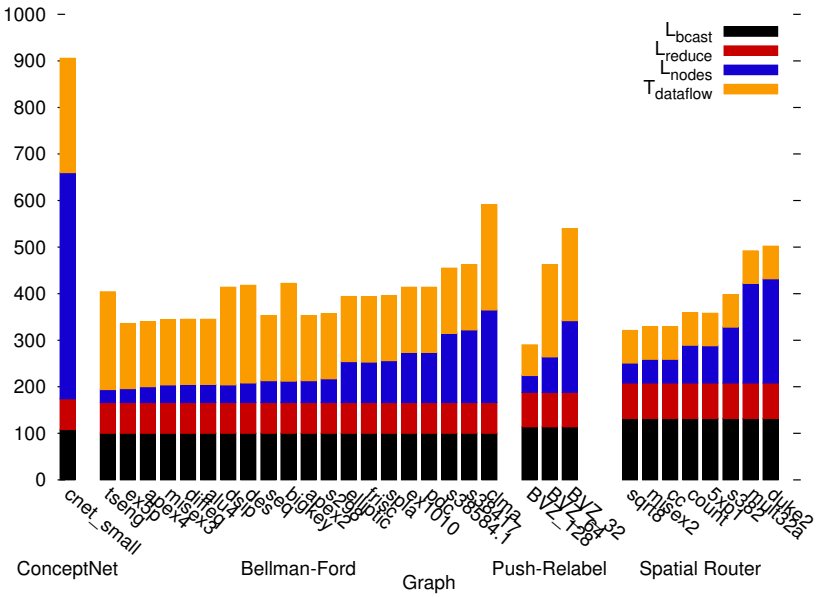


Figure 6.2: Graph-step time (T_{step}) as a sum of the four stages for each application and graph

stage. Figure 6.2 shows the time taken by each of the four stages for each application and graph.

6.2 Accuracy of Performance Model

To gauge the value of our performance model, we measure its accuracy for our applications run on the BEE3 platform. Over all applications and graphs, the mean error of modeled runtime ($T_{step}^{(model)}$) compared to actual runtime ($T_{step}^{(actual)}$) is 11% and the maximum error is 46%. The error is calculated as:

$$|\frac{T_{step}^{(model)}}{T_{step}^{(actual)}} - 1|$$

The performance model tends to underpredict performance and not overpredict. For our benchmark applications and graphs the mean prediction is 90% of actual performance. Figure 6.3 shows that the model under predicts for the Push Relabel and Spatial Router applications. Our model does not include congestion between messages in the network, which causes it to under predict R_{net}. Congestion occurs when contention between messages through a merge backs up predecessor switches to cause messages not destined for

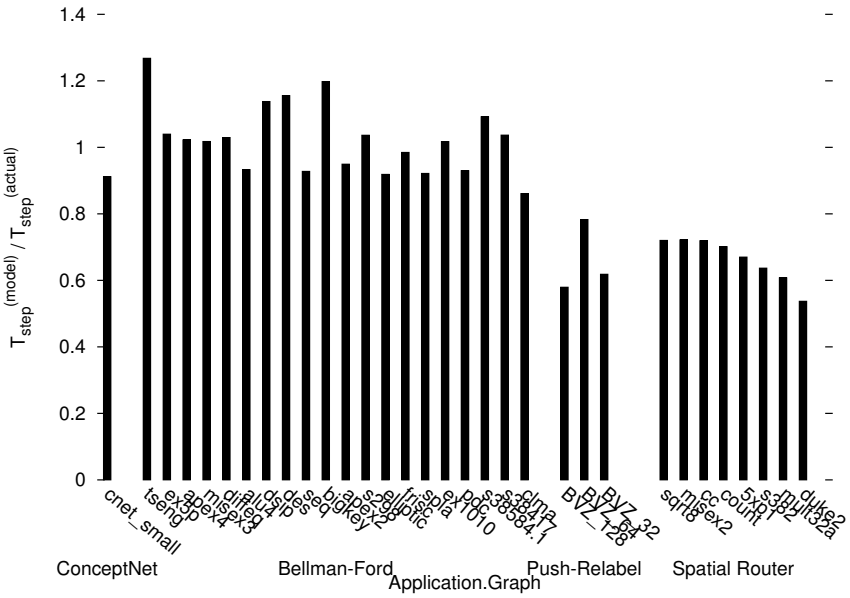


Figure 6.3: Modeled cycles divided by measured cycles for each application and graph. Maximum and minimum deviations are on the right.

the merge to stall. Our model also ignores the case where a message that is not on the critical path, blocks a message on the critical path.

In general, computation in the four phases global broadcast, node iteration, dataflow activity, and global reduce can overlap. This overlap can cause the model to over predict when actions on the critical path in a particular phase occur before all actions in the previous phase have completed. For example, node iteration on a PE close to the sequential controller starts before time L_{bcast}, when the global broadcast reaches the farthest PEs. If one of the nodes close to the sequential controller is on the critical path then L_{bcast} does not fully contribute to T_{step}. Node iteration and dataflow activity overlap so when the last node is not on the critical path, the impact of node iteration is not the full T_{node}. Finally, when the last message received is close to the sequential controller L_{reduce} does not fully contribute to T_{step}. This under prediction indicates that time saved due to overlapping computation is not having a large impact on performance.

Chapter 7

Optimizations

This chapter explores optimizations for the logic architecture and data placement for GraphStep. We show the benefit of dedicated global broadcast and reduce networks in Section 7.1. Node decomposition (Section 7.2) and placement for locality (Section 7.4) optimize how operations and data are assigned to PEs. Section 7.3 compares alternative synchronization styles for message passing and node firing. Table 7.1 shows the speedup of each optimization.

We evaluate our optimizations on GRAPAL programs mapped to the BEE3 platform. The use of GraphStep allows us to try these optimizations without changing GRAPAL source program. However, most of these optimizations are relevant for graph algorithms on multiprocessor architectures in general. In particular, a parallel graph algorithm written in a general model run on a conventional multiprocessor architecture can use node decomposition, placement for locality, and any of the synchronization styles in Section 7.3. Section 7.1 shows the benefit of choosing a multiprocessor architecture with dedicated global networks.

	Decomposition	Locality	Latency Ordering	Dedicated Global Networks
AVG	2.18	1.20	1.01	1.51
MIN	1.00	0.79	0.98	1.17
MAX	5.53	1.81	1.03	1.72

Table 7.1: Speedup due to each optimization

	$\frac{L_{bcast}}{L_{step}}$	$\frac{L_{nodes}}{L_{step}}$	$\frac{L_{dataflow}}{L_{step}}$	$\frac{L_{reduce}}{L_{step}}$
AVG	0.28	0.21	0.33	0.18
MIN	0.14	0.07	0.14	0.08
MAX	0.41	0.61	0.52	0.26

Table 7.2: Contributions, after optimization, to the graph-step critical path, L_{step}

7.1 Critical Path Latency Minimization

For small graphs the critical path latency dominates graph-step time. To model critical path latency, we remove throughput times from the model for T_{step} defined in Section 6.1. Like T_{step}, the critical path latency of a graph-step (L_{step}) contains the global broadcast latency (L_{bcast}), node iteration latency (L_{nodes}), and global reduce latency (L_{reduce}). Instead of $T_{dataflow}$, the critical path of dataflow activity, $L_{dataflow}$, is used.

$$L_{step} = L_{bcast} + L_{nodes} + L_{dataflow} + L_{reduce}$$

Table 7.2 shows the contribution of each latency component to L_{step} for the optimized case. These latencies are calculated with our performance model and averaged over all applications and graphs. To allow averaging, each component of L_{step} is reported as a fraction of total latency.

We optimize the graph-step latency by providing dedicated networks for global broadcast and reduce, and we evaluate an optimization that orders the iteration over nodes performed by each PE.

7.1.1 Global Broadcast and Reduce Optimization

One way in which we customize our logic architecture to GraphStep is that we devote a global broadcast and reduce network to minimize L_{bcast} and L_{reduce}. This is enable by the knowledge of the types of communication in GraphStep. A general actors implementation would have to use the message passing network for this critical global synchronization. In a nonspecialized implementation of global broadcast, a controller PE sends one message to each PE. For global reduce each PE must send to the controller PE, which performs the

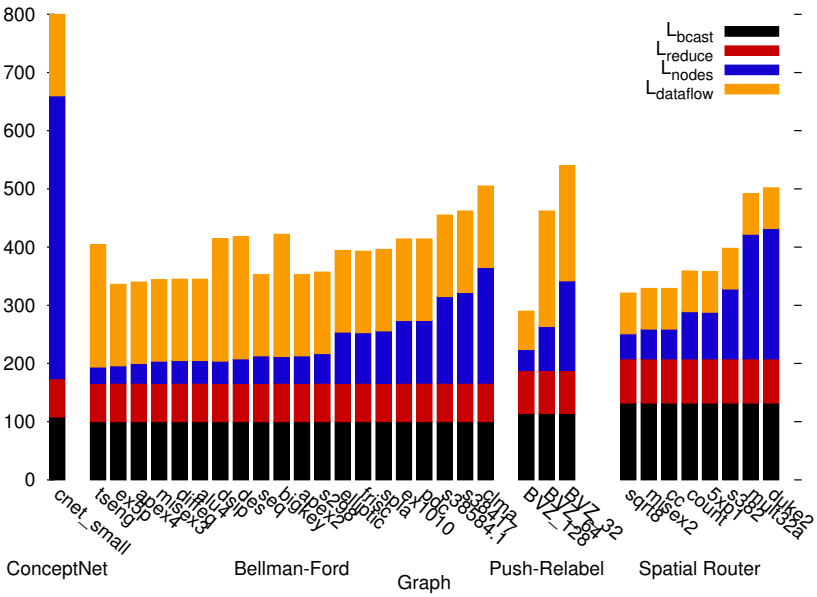


Figure 7.1: Latency of a graph-step (L_{step}) as a sum of the latencies of the four stages for each application and graph

global reduce computations. The unoptimized global reduce implementation would have to trade off between frequency of quiescence updates and network congestion. The unoptimized case is that messages travel over the packet-switched network and are sent or received one at a time. These unoptimized times may be optimistic since network congestion is not modeled here. The unoptimized global broadcast and reduce are:

$$L_{bcast}^{(unopt)} = \max_{i=1}^{N_{PEs}} M_i + 2N_{PEs}\lceil\frac{W_{bcast}}{W_{flit}}\rceil$$

$$L_{reduce}^{(unopt)} = \max_{i=1}^{N_{PEs}} M_i + N_{PEs}\lceil\frac{W_{reduce}}{W_{flit}}\rceil$$

The network latency to PE i is M_i. There are N_{PEs} messages sent or received where the number of cycles for each one is the number of flits it requires. Similar to the optimized broadcast (Equation 6.1), there is a factor of 2 because there is one broadcast at the beginning of a graph-step and one at the end. Table 7.3 compares the unoptimized global broadcast and reduce to the optimized L_{bcast} and L_{reduce} (Equations 6.1,6.2). The speedups due to optimization on global broadcast and reduce are 2.5 and 1.8, respectively. For the total L_{step} the speedups is 1.6.

$\frac{L_{bcast}^{(unopt)}}{L_{bcast}}$	$\frac{L_{reduce}^{(unopt)}}{L_{reduce}}$	$\frac{L_{step}^{(unopt)}}{L_{step}}$
2.5	1.8	1.6

Table 7.3: Speedup due to using global broadcast and reduce network on the L_{bcast} and L_{reduce} stages themselves and on L_{step}

7.1.2 Node Iteration Optimization

We attempted to minimize T_{step} by ordering nodes in each PE to be sensitive to the critical path in message passing and operation firing. The node iteration and dataflow activity stages in a graph-step should be made to overlap so their critical path latency is close to $\max\{L_{nodes}, L_{dataflow}\}$ instead of $L_{nodes} + L_{dataflow}$. We define $L_{dataflow}(v)$ to be the latency of messages and operations from node v's firing to the end of the dataflow stage. This latency includes message hops through fanin trees and fanout trees. Now,

$$L_{dataflow} = \max_{v \in V} L_{dataflow}(v)$$

This calculation uses a model of the network latencies from PE to PE as well as the latencies of operators in each PE. The node iteration optimization orders nodes in each PE by $L_{dataflow}(v)$ so the higher latency nodes come earlier in the iteration.

Unfortunately, optimized node ordering makes little difference on total runtime. Comparing T_{step} for the optimized case to the case where nodes are ordered randomly in PEs gives an average speedup of 1.01, with a minimum of 0.98 and maximum of 1.03.

7.2 Node Decomposition

Section 5.2.1.1 explains how node decomposition allows the logic architecture to use many small PEs with high throughput and how decomposition is implemented. Figure 1.5 provides and example of the decomposition transform on a node. Node decomposition restructures the graph so each large node is broken into a fanin-tree and fanout-tree of small nodes. In-degree, $\Delta_{in}(v) = |pred(v)|$ and out-degree $\Delta_{out}(v) = |succ(v)|$ determine node v's data load and computation load. For load balancing we care about the number prede-

	Original	Decomposed	Speedup
R_{dense}	1395	490	5.5
R_{ser}	268	60	4.1

Table 7.4: Decomposition reduces both R_{dense} and R_{ser}.

cessor and successor edges into nodes assigned to a PE. The set of predecessor edges into PE i is $P_i = pred(V_i)$ and the set of successors is $S_i = succ(V_i)$. So $|P_i| = \sum_{v \in V_i} \Delta_{in}(v)$ and $|S_i| = \sum_{v \in V_i} \Delta_{out}(v)$. The load across PEs is the maximum over predecessors and successors:

$$L = \max_{i=1}^{N_{PEs}} \{\max(|P_i|, |S_i|)\} \tag{7.1}$$

Edge Memory and Send Memory in each PE have capacity D_{edges} and must fit all assigned predecessor and successor nodes, so:

$$D_{edges} \geq L$$

In a simple model where all edges are active in a graph-step, the time to input and output messages for the most imbalanced PE is:

$$R_{dense} = L$$

R_{ser} is the component of $T_{dataflow}$ that models edge operation time and message input and output time (Section 6.1.3). When all edges are active $R_{dense} = R_{ser}$, however edges are usually sparsely active so $R_{ser} \leq R_{dense}$. We find that minimizing R_{dense} helps decrease R_{ser}: Table 7.4 uses the performance model to show the impact of decomposition on both R_{dense} and R_{ser}. On average across benchmark graphs, decomposition reduces R_{dense} by 6.3 times which helps decrease R_{ser} by 4.7 times.

Figure 7.2 shows the speedup from performing decomposition. Graphs with and without decomposition are placed for locality, and nodes are ordered by fanin/fanout tree latency-depth. The mean speedup is 2.6, the minimum is 1.0 and the maximum is 5.6.

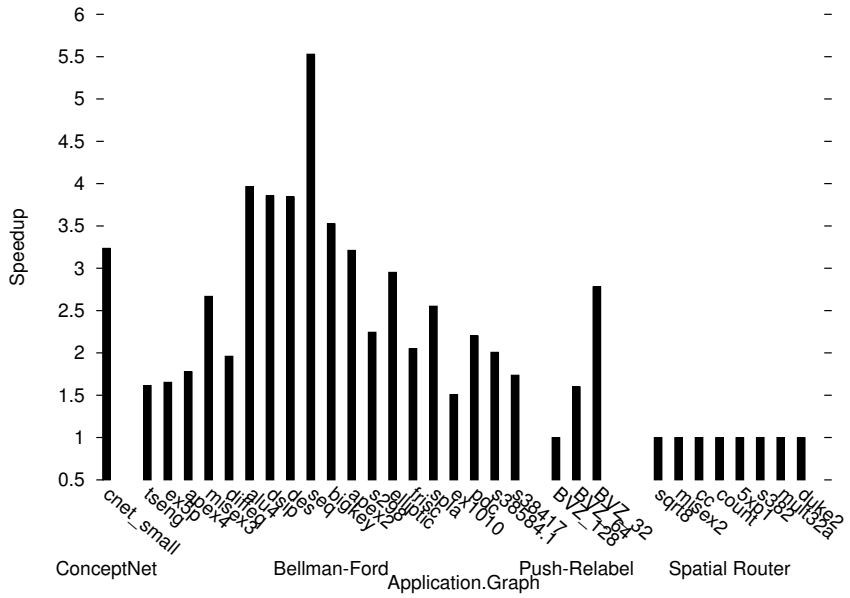


Figure 7.2: Speedup due to decomposition on T_{step} for each graph which fits without decomposition.

7.2.1 Choosing Δ_{limit}

The decomposition transform is performed on graphs at runtime before they are loaded into memory. The runtime algorithm places the limit Δ_{limit} on in- and out-degrees:

$$\forall v \in V : \Delta_{limit} \geq \Delta_{in}(v) \wedge \Delta_{limit} \geq \Delta_{out}(v)$$

Figure 1.5 shows an example transform where $\Delta_{limit} = 2$. Δ_{limit} is chosen by the runtime algorithm. Smaller Δ_{limit} enables load-balancing nodes across PEs. However, smaller Δ_{limit} increases the depth of fanin and fanout trees, which increases the graph-step latency, $L_{dataflow}$. So we maximize Δ_{limit} with the constraint that PEs can be load balanced. Perfectly load balanced PEs have $|P_i| = |S_i| = |E|/N_{PEs}$. For each decomposed node, v, we want $\Delta_{in}(v) \leq |E|/N_{PEs}$ and $\Delta_{out}(v) \leq |E|/N_{PEs}$, so we set:

$$\Delta_{limit} = \frac{|E|}{N_{PEs}}$$

For the purpose of maximizing memory load balance, it is fine to have one node take

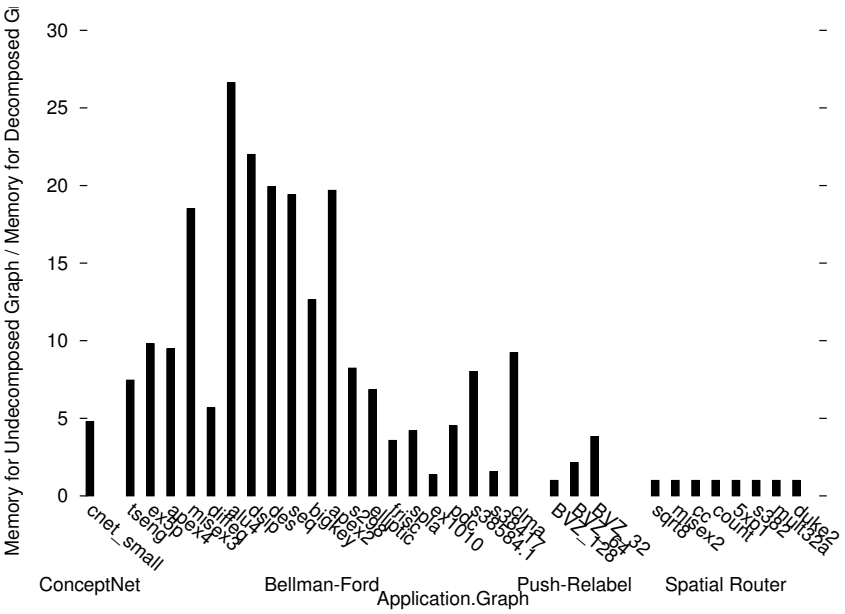


Figure 7.3: The ratio of memory required by the undecomposed graph to memory required for the decomposed graph for each application and graph. RMS is on the right. Uniform PE memory sizes are assumed.

up all or most of a PE. The load balancer can use the many small nodes to pack PEs evenly. Figure 7.3 shows, for each application and graph, the ratio of memory required when decomposition is not performed to memory required with decomposition. This assumes that memory per PE is uniform, as it is in our logic architecture. The average memory reduction due to decomposition is 10 times, and the maximum is 27 times.

Unlike memory requirements, the amount of computation work for a node in a particular graph-step is dynamic. In general only a subset of nodes are active in each graph-step so the observed computation load balance is not the same as the memory load balance and is different for each graph-step. Large decomposed nodes which take up most of the PE could potentially ruin the computation load-balance. Also contributing to T_{step} is the amount of message traffic. The node to PE placement algorithm tries to place fanin and fanout nodes on the same PE as their descendants. Smaller nodes may result in less message traffic due to fewer long-distance messages, resulting in lower decreasing R_{net}. However large nodes mean the depth of fanin and fanout trees in smaller, resulting in a lower $L_{dataflow}$ latency.

Figure 7.4 plots the root mean square of normalized T_{step} across all graphs for a range

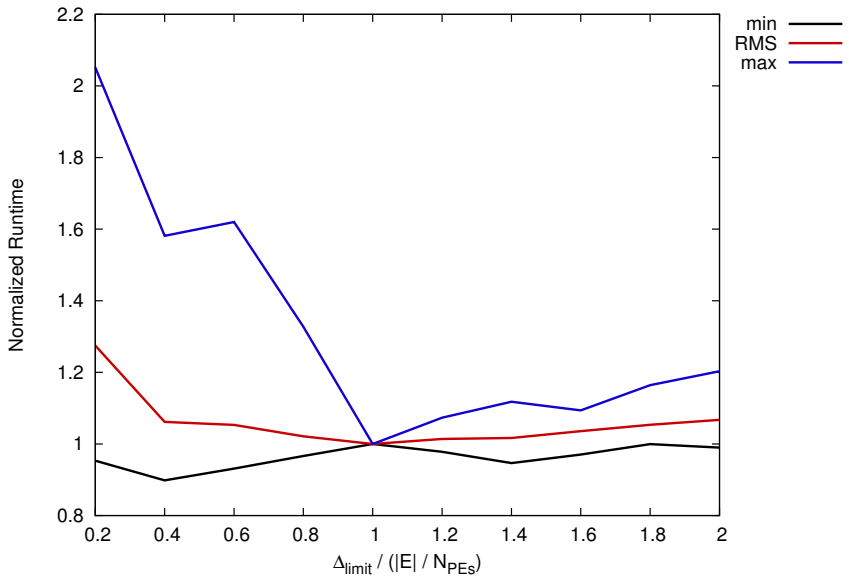


Figure 7.4: Root-mean-square along with min and max across applications and graphs of T_{step} for a range of Δ_{limit}s. Δ_{limit} is normalized to $|E|/N_{PEs}$.

of Δ_{limit}s; Δ_{limit} is normalized to $|E|/N_{PEs}$. The choice with the best RMS is the simplest: $\Delta_{limit} = |E|/N_{PEs}$. Figure 7.5 shows the effect of varying Δ_{limit} on various contributors to graph-step time, where Δ_{limit} is normalized to $|E|/N_{PEs}$. This figure shows that $L_{dataflow}$ is the dominating contributor. For $\Delta_{limit} < |E|/N_{PEs}$ the depth of fanin and fanout trees increases, so $L_{dataflow}$ becomes more severe. For $\Delta_{limit} > |E|/N_{PEs}$ it is impossible to perfectly load-balance nodes across PEs. Figure 7.6 shows the number of fanin and fanout level across decomposed nodes.

7.3 Message Synchronization

For a graph which has no decomposed nodes, each node operates on messages sent from the last graph-step. Normally, we use a global barrier to tell each node that all messages have arrived so it can fire. An alternative to global synchronization is to use nil messages so each edge passes a messages on each step. This way, each node can fire as soon as it receives one message from each of its input edges, so there is no need for a costly global barrier. When we consider decomposed graphs, we have three options:

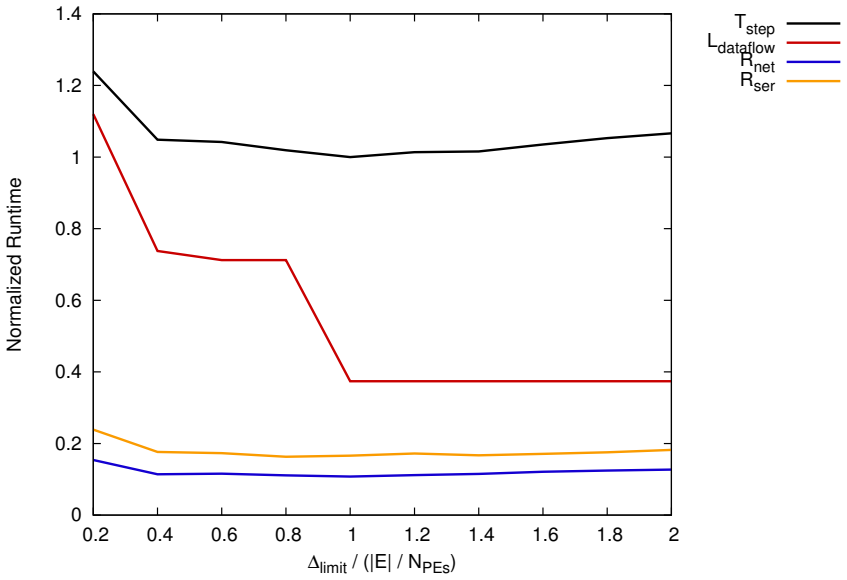


Figure 7.5: Contributors to T_{step}, which are affected by Δ_{limit}, along with T_{step}

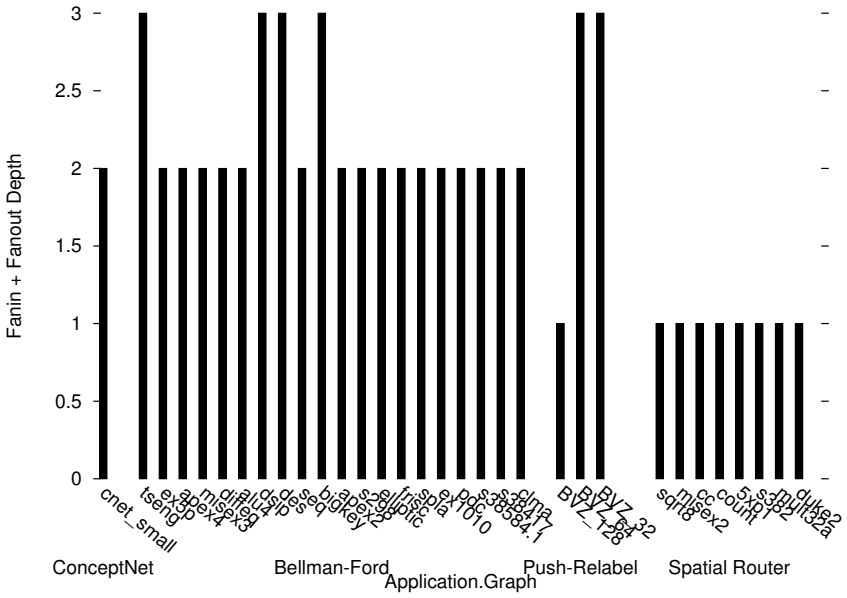


Figure 7.6: The depth of fanin and fanout trees when $\Delta_{limit} = |E|/N_{PEs}$

- *fully-dense*: All edges transmit (possibly nil) messages on each graph-step. Nodes use local dataflow synchronization to fire.
- *fully-sparse*: A sparse subset of edges are active on each graph-step, including fanin edges. Nodes use global barrier synchronization to fire. Since they input one message only, fanout nodes can be dataflow synchronized with no cost in terms of extra message traffic.
- *fan-dense*: Edges internal to fanin trees transmit messages on each graph-step. Likewise, nodes internal to fanin trees are dataflow synchronized. Edges which were in the original undecomposed graph are sparsely active so fanin leaf nodes need to be barrier synchronized. Fanout nodes are dataflow synchronized. This is the style used by the logic architecture, described in Section 5.2.1.1 and Figure 5.11.

Figure 7.7 shows that the rate of edge activation varies from $1/800$ to $1/4$ across applications and graphs with a mean of $1/8$. This low activation means `fully-dense` synchronization has a high cost in terms of network traffic and message processing time. Figure 7.6 shows the maximum number of fanin and fanout levels over decomposed nodes. The number of levels for an undecomposed node is 1. `fully-dense` synchronization will require one global barrier for each level.

Figure 7.8 shows time for the `fully-dense` and `fully-sparse` cases normalized to our default `fan-dense` case. The `fully-dense` case has a mean T_{step} of 4 times the fan-dense case due to higher compute and computation load. Edge activation (Figure 7.7) is too low for `fully-dense` to be useful. The `fully-sparse` case has a mean T_{step} of 1.7 times the fan-dense case since it needs to perform more barrier synchronizations, and the time saved due to sparse activation of fanin-tree nodes is minimal. Figure 7.6 shows that in most cases the latency of all barrier synchronizations in a graph-step increases by 2 to 3 times.

7.4 Placement for Locality

In this section we study the effect of assigning nodes to PEs to optimize for locality. Most graphs for most applications can be partitioned into balanced regions so the probability

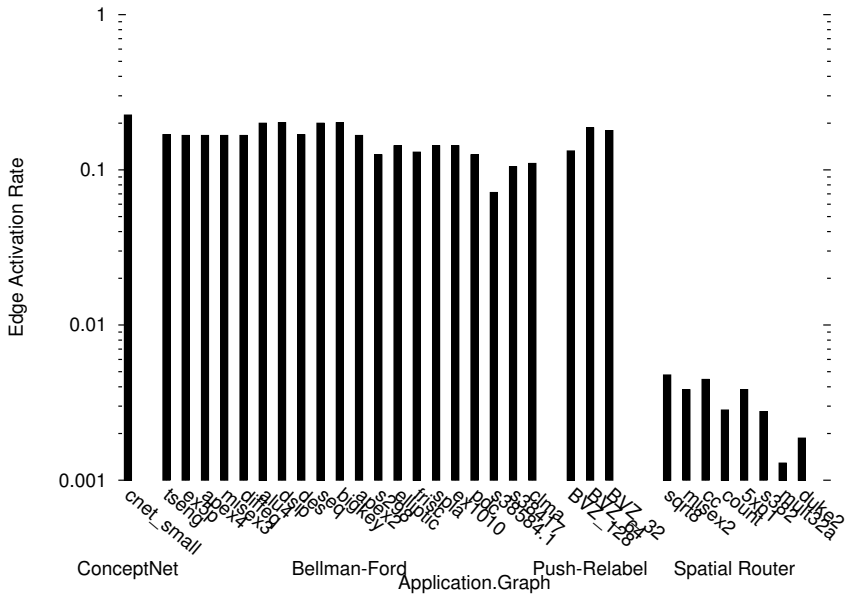


Figure 7.7: Average fraction of edges which are active (i.e. pass messages) over graph-steps

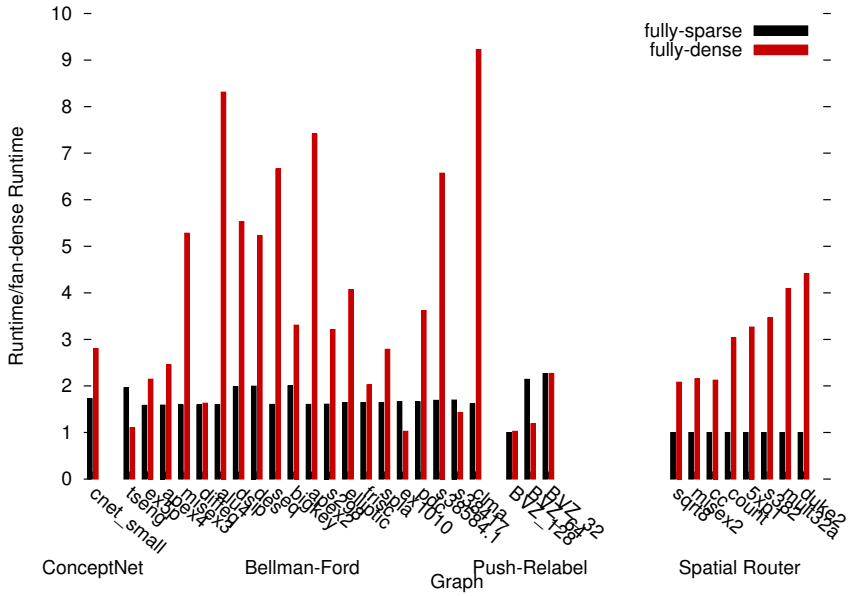


Figure 7.8: Performance of message synchronization options

that a pair of nodes in the same region is connected is higher than the average probability. Assigning local regions to PEs can minimize the message traffic between PEs and minimize message latency on the critical path.

We define $A : V \rightarrow PEs$ to be assignment of nodes to PEs. The number of cut edges in PE i is sum over non-local predecessor and successor nodes:

$$C_i = \sum_{v \in V_i} |\{u \in pred(v) \cup succ(v) : A(u) \neq A(v)\}|$$

The total cut edges is:

$$C = \sum_{i=1}^{N_{PEs}} C_i$$

To assign for locality, C should be minimized while keeping the maximum load, L, (Equation 7.1) close to optimal. To measure the quality of the maximum load we define the load balance B:

$$B = \frac{L}{|E|/N_{PEs}}$$

With an optimal load balance $B = 1$.

Graphs from physics problems such as FEM usually have a two- or three-dimensional structure. A graph with a regular three-dimensional structure (e.g. a 3-D fluid dynamics simulation) can be partitioned into balanced, local regions by slicing the three-dimensional space, so $C_i \propto |V_i|^{2/3}$. Our Spatial Router works on a two-dimensional mesh which can be partitioned into a two-dimensional array of submeshes with $C_i \propto |V_i|^{1/2}$. Circuit netlists (used for benchmarks in Bellman-Ford) often follow Rent's Rule [87]. Rent's Rule states that for a large range over $|V_i|$, $C_i \propto |V_i|^P$ where P is some constant. When $P < 1$ there is local structure in the graph, which is common for circuits.

The placement algorithm partitions the graph hierarchically to match the hierarchical structure of the platform. Total chip-to-chip bandwidth is less than total bandwidth between neighboring PEs. The total bandwidth of high levels of a BFT on-chip network (Figure 5.12) is less than total bandwidth between its neighboring PEs. To match this structure, we first partition the graph into a binary tree of subgraphs: Each subgraph is partitioned into two children with PE-sized subgraphs as the leaves. We then map this binary tree onto

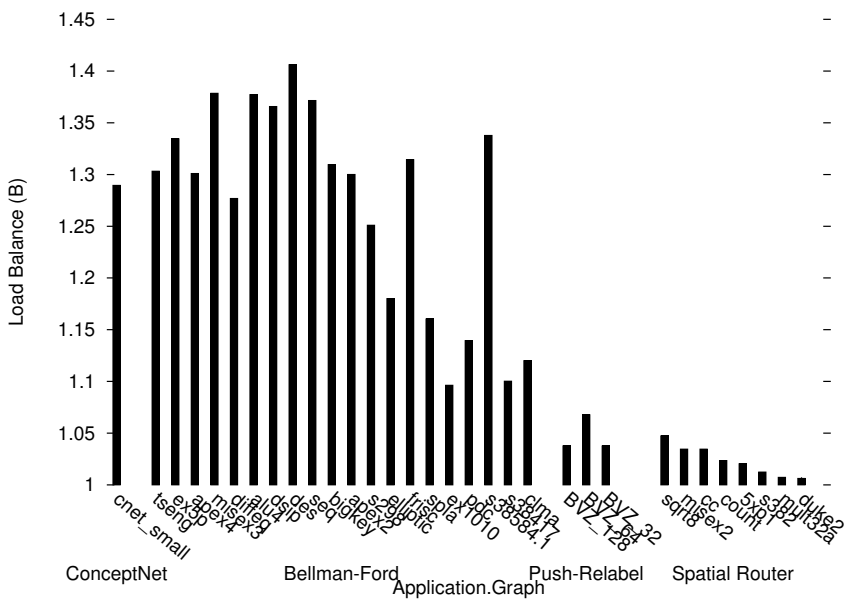


Figure 7.9: Load balance (B) with local assignment for each application and graph

the tree of chips, subnetworks and PEs. Each binary partition is performed by the Mlpart 5.3.6 graph partitioner [88].

Each recursive step of the binary partition algorithm maps the vertices of a subgraph, $V^{(sub)}$, into two parts: $V_1^{(sub)}$ and $V_2^{(sub)}$. The assignment function computed in the binary partition is $A^{(sub)} : V^{(sub)} \rightarrow \{1, 2\}$. The objective function is to minimize the number of cut edges between $V_1^{(sub)}$ and $V_2^{(sub)}$:

$$obj(A^{(sub)}) = |\{e \in E^{(sub)} : A^{(sub)}(pred(e)) \neq A^{(sub)}(succ(e))\}|$$

The load balance $B^{(sub)}$ specifies the target load balance, so that:

$$B^{(sub)} \approx 2\frac{V_i^{(sub)}}{V^{(sub)}}$$

$B^{(sub)}$ is chosen so the final load balance of edges on PEs is $B \approx 1.2$. The height of the binary tree of partitions is $h = \lceil \log_2 N_{PEs} \rceil$ so setting $B^{(sub)} = 1.2^{1/h}$ gives $B \approx 1.2$. The mean load balance across benchmark applications and graphs is 1.2 with a maximum of 1.4 and minimum of 1.0. Figure 7.9 shows the load balance for each application and graph.

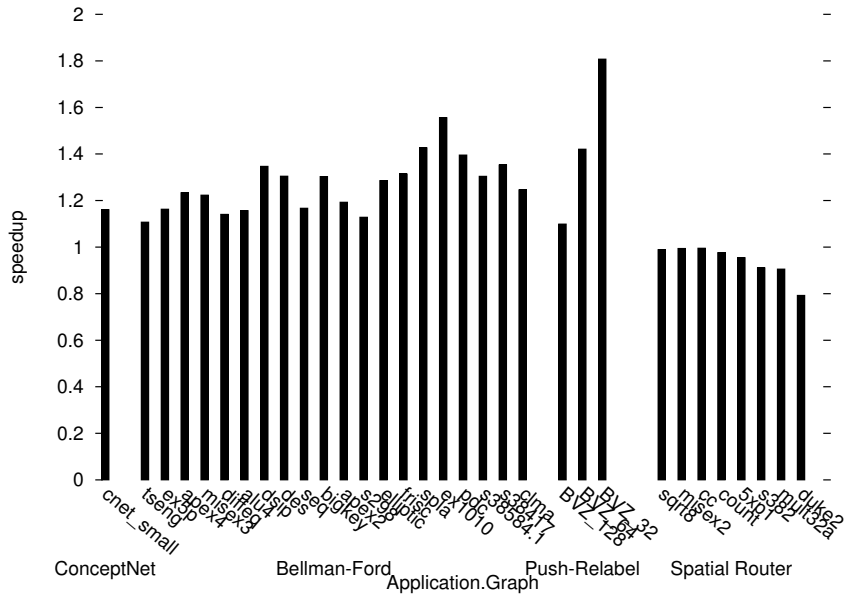


Figure 7.10: Speedup due to local assignment for benchmark graphs and applications

We compare T_{step} for locally assigned graphs to T_{step} for the baseline case. The baseline assignment is oblivious to locality and randomly assigns nodes to PEs with the constraint that the load balance B is minimized. The overall mean speedup by local assignment is 1.3, the minimum is 0.8 and the maximum is 1.8. Figure 7.10 shows the speedup for all application and graphs due to local assignment. Most graphs benefit from local assignment with the exception of Spatial Router graphs.

Local assignment matters most when message traffic (R_{net}) is the main contributor to T_{step}. We use our performance model to estimate the effect of local assignment on message traffic in the context of total runtime. Figure 7.11 shows T_{step} for the baseline case (Oblivious), the locally assigned case (Local), and the case where the cost of message traffic is removed ($R_{net} = 0$) (No Traffic). For our benchmarks, local assignment always achieves the same T_{step} as no network traffic.

In some cases local assignment produces worse results than oblivious assignment. As discussed in Section 7.2, load balancing minimizes the static load, L, as a heuristic for minimizing the dynamic load, R_{ser}. With random assignment there is no correlation between node firings in the same PE so R_{ser} should be proportional to L. When nodes are locally

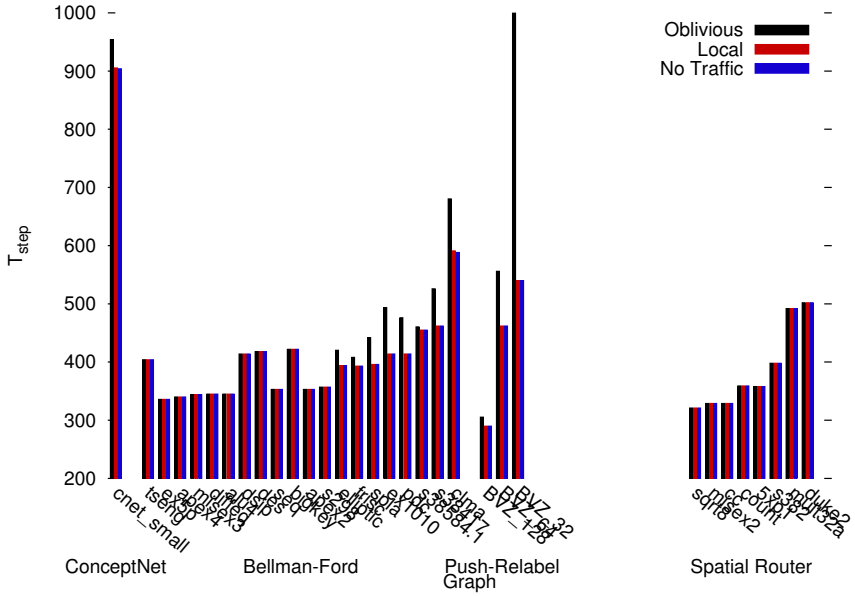


Figure 7.11: The performance model is used to show the effect of local assignment on message traffic (R_{net}) and a fraction of T_{step}.

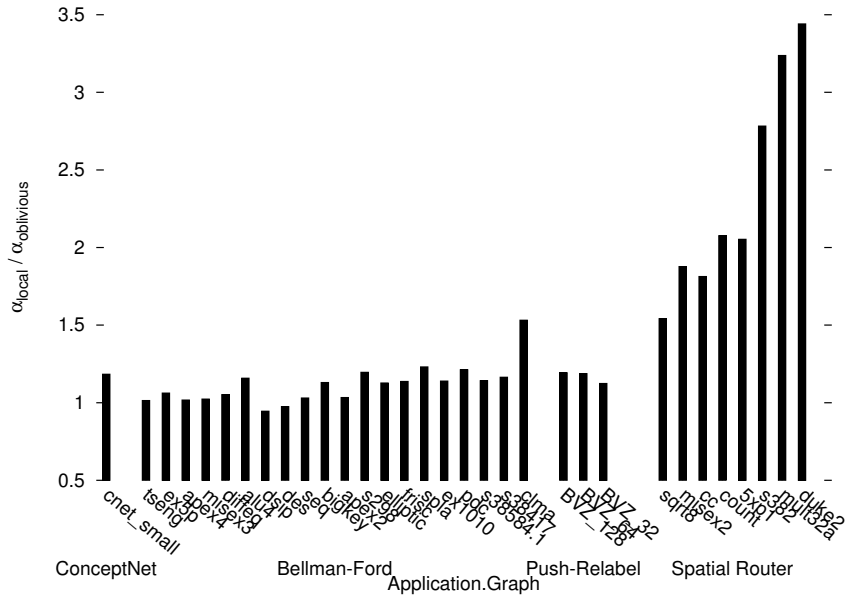


Figure 7.12: $\alpha = R_{ser}/L$ is compared for local assignment and random assignment. This shows how much correlation in activity between nodes in the same PE increases R_{ser}.

assigned PE firings can be correlated, making R_{ser} unexpectedly large. For example, the graph may model a narrow pipe where activation propagates in a wavefront along the pipe. A good local assignment will assign a section of the pipe to each PE, resulting in only one or two PEs containing active nodes at any point in time. Correlated activation causes R_{ser} to increase relative to L, so we define $\alpha = R_{ser}/L$ to measure the effect of correlation on R_{ser}. Figure 7.12 compares α for local assignment (α_{local}) and for random assignment (α_{random}). This shows that correlation increases R_{ser} by up to 3 times. Correlation is worst for Spatial Router graphs, which are the ones for which local assignment decreases performance. This makes sense since the Spatial Router propagates wavefronts on a mesh. The other graph which has a relatively high correlation is Bellman-Ford's clma. However, clma benefits from locality since its network traffic is decreased (Figure 7.11).

Chapter 8

Design Parameter Chooser

The logic architecture generated by the compiler has parameters that determine how many FPGA logic and memory resources are used. These design parameters size components which compete for logic and memory resources so they need to be traded-off judiciously for good performance. Since datapath widths, memory word widths and method contents are application dependent, design parameters need to be chosen for each GRAPAL application. Further, each FPGA model has a different number of logic and memory resources, so the design parameters must be customized to the target FPGA to maximally utilize resource without using more resources than available. For typical FPGA programming models, the programmer must consider the resource availability of the target FPGA for the program to be correct and/or efficient. A typical FPGA programmer must also set the clock frequency so signals have enough time to propagate from one stage of registers to the next. By automatically choosing the clock frequency and good design parameter values the compiler abstracts above the target FPGA, so one program can be compiled to a wide range of FPGAs. This chapter explains how the GRAPAL compiler automatically chooses design parameters to provide good performance and evaluates the quality of the automatic chooser.

The number of PEs, N_{PEs}, and the interconnect flit width, W_{flit}, are the two parameters that compete for logic resources. The depth of node memories, D_{nodes}, and the depth of edge memories, D_{edges}, are the two parameters that compete for BlockRAM memory resources. Logic and memory resources are disjoint so the trade off between N_{PEs} and W_{flit} can be determined independently from the tradeoff between D_{nodes} and D_{edges}.

Application	Full Design	PE		4-2 switch	
	MHz	logic-pairs	MHz	logic-pairs	MHz
ConceptNet	100	2980	150	5224	195
Bellman-Ford	100	3247	174	4736	195
Push-Relabel	100	7493	147	5132	201
Spatial Router	100	4963	172	4958	194

Table 8.1: The resource use model used by `LogicParameterChooser` (Section 8.2) is based on logic-pairs used by each PE and by each 4-2 switch. The full design found by `FullDesignFit` (Section 8.4 for all applications is 100 MHz. Maximum frequencies imposed by individual PE and 4-2 switch components are 50% to 2x greater than the full design.

8.1 Resource Use Measurement

The three types of hardware resources used in the Xilinx Virtex-5 XC5VSX95T target device are 6-LUTs, flip-flops, and BlockRAMs. Primitive FPGA components such as 5-LUTs, SRL32 shift registers and distributed RAM use the same resources as 6-LUTs and can be measured in terms of 6-LUTs. Other primitive devices, such as DSP48 multipliers, are not used by the application logic. Virtex-5 hardware has one flip-flop on the output of each 6-LUT. A 6-LUT and its associated flip-flop may be used together or may be used without the other. The cost of each logic component, such as a PE or network switch, is measured in terms of pairs of one 6-LUT and one flip-flop called *logic-pairs*. Logic-pairs used by a component is the number of physical 6-LUT, flip-flop pairs in which the 6-LUT is used, the flip-flop is used, or both are used. Table 8.1 shows the number of logic-pairs used by a PE and by a 4-2 switch in the BFT for each benchmark application.

Before deciding how many components (e.g. PEs or network switches) of each type will be used, the compiler must have a good model of resources used by each component. The procedure used to measure resources used is called `ComponentResources`. The FPGA tool chain passes a VHDL design through the synthesis, map, place and route stages. These stages contain multiple NP-Hard optimizations so it is difficult to model their outcome in terms of resource use and clock frequency without actually running them. The FPGA tool chain is wrapped into a procedure so it can be used by our parameter-choice algorithms. The standard use model for an FPGA tool chain is for a human programmer to:

- Specify the VHDL (or other HDL) design and the desired clock frequency (target fre-

quency).

- Run the tool chain: synthesis, map, place and route.
- Read log files which report resource usage, whether the desired clock frequency was met, and the achieved clock frequency.

The GRAPAL compiler's lowest-level component measurement procedure inputs target frequency, prints out logic modules as VHDL, calls the tool chain commands, then parses log files to return resource and frequency outcomes. All higher level algorithms should not have to specify the clock frequency; instead the called procedure should return a good clock frequency. The primary component measurement procedure inputs only the component, and outputs resource usage along with achieved clock frequency. It does this by searching over target frequency with multiple lower-level passes. Since FPGA report the achieved frequency, whether it is less than or greater than the target frequency, each iteration of this search can use the previous iteration's achieved frequency. The first iteration uses a target of 100 MHz, which is usually within a factor of 2 of the final frequency. By default two iterations are performed, which is usually enough to bring the final frequency within 5% of optimal.

8.2 Logic Parameters

This section explains the compiler's strategy for choosing the N_{PEs} and W_{flit} parameters, examines the effectiveness of the strategy, and explains the `LogicParameterChooser` algorithm which implements the logic parameter choice.

Ideally the compiler would know the runtime workload on PEs and the load on the interconnect. With knowledge of workloads, the compiler could optimally allocate logic-pairs to PEs and the interconnect to maximize performance. However, the workload depends on the graph supplied at runtime and the pattern of activation on node and edge at runtime. Since the compiler does not have knowledge of the workload it uses a strategy to come within a factor of 2 of the optimal allocation for any given run. It gives half of each FPGA's logic-pairs, A_{all}, to PEs and the other half to network switches, so PEs' computation time is at most 2 times optimal and message routing time is at most 2 times optimal. N_{PEs}, is ap-

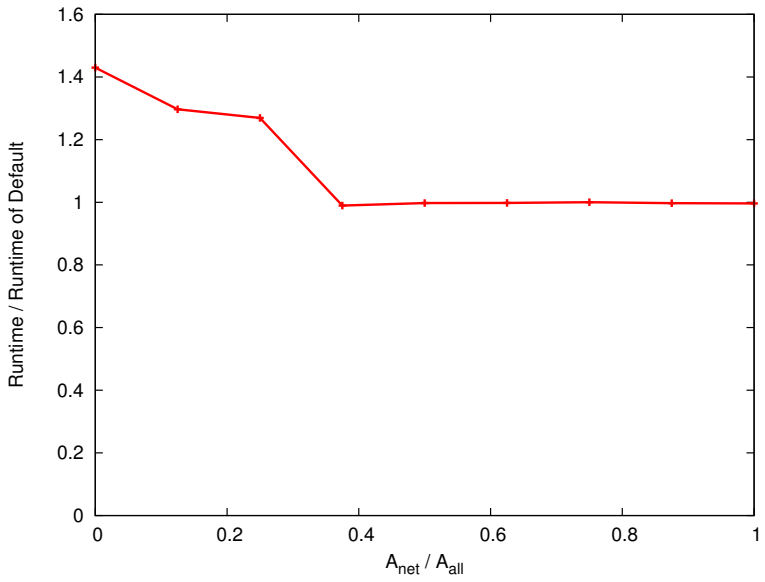


Figure 8.1: The mean normalized runtime of all applications and graphs is plotted for each choice of A_{net}. On the y-axis runtime is normalized to the runtime of the default choice where $A_{net} = A_{all}/2$ and on the x-axis A_{net} is relative to A_{all}.

proximately proportional to area devoted to PEs, A_{PEs}. Time spent computing, assuming a good load balance, is approximately inversely proportional to N_{PEs}. So doubling A_{PEs} speeds up the computing portion of total time by approximately 2. Total switch bandwidth is approximately proportional to area devoted to interconnect, A_{net}. Time spent routing messages is approximately inversely proportional to total switch bandwidth. So doubling A_{net} speeds up the communication portion of total time by approximately 2.

We evaluate the choice of $A_{net} = A_{all}/2$ by running `LogicParameterChooser` on a range of target $F_{net} = A_{net}/A_{all}$ ratios. Figure 8.1 shows the mean normalized runtime of all applications and graphs for the range $F_{net} \in [0, 1]$. For these graphs the choice of $A_{net} = A_{all}/2$ is within 1% of the optimal choice of $A_{net} = 3/8A_{all}$. Runtime for each run is normalized to the runtime of the run with $A_{net} = A_{all}/2$, then the mean is taken over all runs with a particular A_{net} value. Figure 8.1 shows that runtime only varies for F_{net}s in the range $[1/4, 1/2]$. W_{flit} is lower-bounded by a $W_{flit}^{(min)}$ so F_{net} cannot be decreased beyond $1/4$. W_{flit} is upper-bounded by a $W_{flit}^{(max)}$ so there is no point to increasing F_{net} beyond $1/2$.

```
LogicParameterChooser =
  binary search to maximize W_flit
    in the range [W_flit^min, W_flit^max]
    given constraint A_net(W_flit) ≤ A_all / 2
  return W_flit and N_PEs^(i)(W_flit) for each i

// Sum interconnect area over all FPGAs.
// Each FPGA's interconnect area is a function of both W_flit and N_PEs^(i).
A_net(W_flit) = Σ_{i=1}^{4} A_net^(i)(W_flit, N_PEs^(i)(W_flit))

// Compute number of PEs on FPGA i as a function of W_flit.
N_PEs^(i)(W_flit) =
  binary search to maximize N_PEs^(i)
    in the range [1, ∞]
    given constraint A_PE × N_PEs^(i) + A_net^(i)(W_flit, N_PEs^(i)) ≤ A_all^(i)
```

Figure 8.2: `LogicParameterChooser` Algorithm. The outer loop maximizes W_{flit}. With a fixed W_{flit}, the inner loop can maximize the PE count for each FPGA, $N_{PEs}^{(i)}$, with the constraint that PEs and interconnect fit in FPGA resources. Resources used for interconnect, $A_{net}^{(i)}(W_{flit}, N_{PEs}^{(i)})$, can only be calculated when both W_{flit} and $N_{PEs}^{(i)}$ are known.

The master FPGA has fewer available resources than each of the three slave FPGAs since it has the MicroBlaze processor and a communication interface to the host PC (Figure 5.2). In order to handle multiple FPGAs with different resources amounts the parameter choice algorithm sums resources over all chips in the target platform. A_{net} is logic-pairs used for interconnect on all FPGAs, and A_{PEs} is logic-pairs used for all PEs. A_{all} is the total number of logic-pairs available for GRAPAL application logic across all FPGA. The entire interconnect, including each on-chip network, uses a single flit width, so W_{flit} is global across FPGAs. On-chip PE count can be unique to each FPGA so `LogicParameterChooser` must choose $N_{PEs}^{(i)}$ for FPGA i, so $N_{PEs} = \sum_{i=1}^{4} N_{PEs}^{(i)}$.

Figure 8.2 shows the `LogicParameterChooser` algorithm. This algorithm must maximize $N_{PEs}^{(i)}$ for each FPGA along with maximizing W_{flit} so $A_{net} \approx A_{PEs}$. It must also satisfy the constraint that resources on each FPGA are not exceeded: $A_{PEs}^{(i)} + A_{net}^{(i)} \leq A_{all}^{(i)}$. The only variable $A_{PEs}^{(i)}$ depends on is $N_{PEs}^{(i)}$. $A_{net}^{(i)}$ depends on $N_{PEs}^{(i)}$ for the switch count and network topology and W_{flit} for the size of each switch. An outer binary search finds W_{flit} and an inner binary search finds $N_{PEs}^{(i)}$ given W_{flit}. Binary searches are adequate for finding optimal logic parameter values since $A_{PEs}^{(i)}$ and $A_{net}^{(i)}$ are monotonic functions of the

logic parameters. In FPGA i, the maximum $N_{PEs}^{(i)}$ is codependent with the maximum W_{flit} and W_{flit} is common across all FPGA so it is simplest to find W_{flit} first, in the outer loop.

Each binary search first finds an upper bound by starting with a base values and doubling it until constraints are violated. It then performs divide and conquer to find the maximum feasible value. The range of the binary search over W_{flit} is $[W_{flit}^{(min)}, W_{flit}^{(max)}]$. $W_{flit}^{(min)}$ is the message address width, used to route a message to a PE, which must be contained in the first flit of every message. $W_{flit}^{(max)}$ is the minimum of the width of inter-FPGA channels and the width of a message. Each flit must be routable over inter-FPGA channels, and there is no reason to make flits larger than a message.

`LogicParameterChooser` must estimate resources used by a single PE, A_{PE}, and estimate resources used by the interconnect, $A_{net}^{(i)}$ as a function of $N_{PEs}^{(i)}$ and W_{flit}. Each PE inputs and outputs messages, so some of its datapath widths depend on width of the address used to route messages to PEs. A_{PE} depends on N_{PEs}, which we are trying to compute as a function of A_{PE}. `LogicParameterChooser` places a lower bound on PE resources by using an address width of 0, then places an upper bound on PE resources by using an address with large enough to address a design with as many lower-bound PEs as can fit on the FPGAs. To be conservative, the upper bound is used as A_{PE}. The effect of this estimation is minimal since the address width for this upper bound is only 1 or 2 bits greater than actual width for all benchmark applications. Since PE logic depends on the GRAPAL application, resource usage for PE with a concrete address width must be measured by the procedure wrapping the FPGA tool chain, `ComponentResources`. Calls to the FPGA tool chain are expensive, so the two measurements for A_{PE} are performed by `ComponentResources` before the binary search loops.

$A_{net}^{(i)}$ is estimated as the sum of resources used by all switches in the on-chip network. The logic for each switch is a function of both W_{flit} and address width, W_{addr}. The address width used is the same slight overestimate as that used by A_{PE}. Switch resources must be recalculated for each iteration of the outer binary search loop over W_{flit}. To avoid a call to the expensive FPGA tool chain in each iteration, a piecewise-linear model for switch resources is used. A linear model provides a good approximation since all subcomponents of switch logic which depend on W_{addr} or W_{flit} are linear structures. At compiler installation-

time, before compile-time, `ComponentResources` is run for each switch crossed with a range of powers of two of W_{addr} and W_{flit}. All powers of two are included up until the switch exceeds FPGA resources. These `ComponentResources` measurements provide the vertices of the piecewise-linear model. The interpolation performed to approximate resources used by a switch, $A_{sw}(W_{addr}, W_{flit})$, is:

$$A_{sw}(W_{addr}, W_{flit}) = A_{base} + A_{addr} + A_{flit}$$

where:

$$\begin{aligned}
A_{base} &= A_{sw}(\lfloor W_{addr} \rfloor_2, \lfloor W_{flit} \rfloor_2) \\
A_{addr} &= \frac{W_{addr} - \lfloor W_{addr} \rfloor_2}{\lceil W_{addr} \rceil_2 - \lfloor W_{addr} \rfloor_2} \left[A_{sw}(\lceil W_{addr} \rceil_2, \lfloor W_{flit} \rfloor_2) - A_{base} \right] \\
A_{flit} &= \frac{W_{flit} - \lfloor W_{flit} \rfloor_2}{\lceil W_{flit} \rceil_2 - \lfloor W_{flit} \rfloor_2} \left[A_{sw}(\lfloor W_{addr} \rfloor_2, \lceil W_{flit} \rceil_2) - A_{base} \right]
\end{aligned}$$

$\lfloor \mathbf{.} \rfloor_2$ and $\lceil \mathbf{.} \rceil_2$ round down and up, respectively, to the nearest power of 2.

8.3 Memory Parameters

This section explains the compiler's strategy for choosing the D_{edges} and D_{nodes} memory parameters, examines the effectiveness of the strategy, and explains the `MemoryParameterChooser` algorithm which implements memory parameter choice.

There are five memories in each PE which contain state for nodes and edges (Figure 5.7). Node Reduce Memory, Node Reduce Queue, and Node Update Memory each store one word for each node assigned to the PE. The depth of these node memories, D_{nodes}, determines the maximum number of nodes which can be assigned to a PE. Edge Memory stores one word for each edge assigned to the PE, while Send Memory stores one word for each successor edge from a node assigned to the PE. The depth of these edge memories, D_{edges}, determines the maximum number of predecessor or successor edges of nodes assigned to a PE. With a good load balance it is not beneficial to have different depths for Edge Memory and Send Memory.

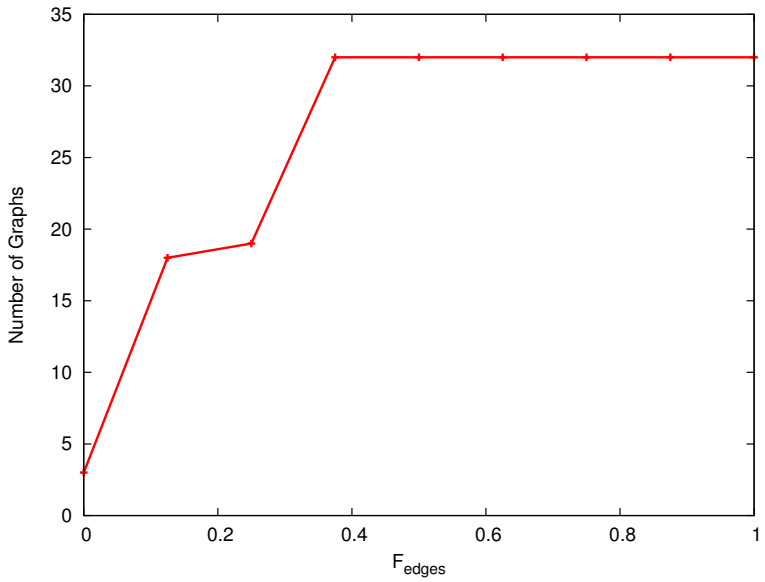


Figure 8.3: Number of benchmark graphs which fit in memory across all applications for each ratio of edge BlockRAMs to total BlockRAMs (F_{edges})

For all PEs the compiler allocates B_{nodes} BlockRAMs to node memories and B_{edges} BlockRAMs to edge memories out of B_{PE} available BlockRAMs in the PE. All available BlockRAMs on the FPGA are divided evenly between PEs, except for those which are used for MicroBlaze memory on the master FPGA. At compile-time, the compiler chooses $F_{edges} = B_{edges}/B_{PE}$, to maximize the chance that a graph loaded at runtime will fit in memory. Typical graphs have many more edges than nodes, and much more edge bits than node bits. Our benchmark graphs have between 1.9 and 4.8 edges per node with an average of 3.6, and between 0.5 and 2.1 times more bits in edge memory than node memory. The compiler's strategy is to choose a robust F_{edges} by devoting approximately $3/4$ of the available BlockRAMs to edge memories and the rest to node memories. Compared to $B_{edges}/B_{PE} = 3/4$ an optimal strategy would allow at most $4/3$ more edges. Even with $B_{nodes}/B_{PE} = 1/4$, node memory capacity is rarely the limiter. Figure 8.3 maps F_{edges} to the number of graphs for all of our benchmark application that fit in memory. For $F_{edges} = 3/4$ all benchmark graphs fit in memory.

`MemoryParameterChooser` must use a model to count the number of BlockRAMs required for each memory to get D_{nodes} and D_{edges} as a function of F_{edges}. $B(depth, width)$

is the number of BlockRAMs used by a memory as a function of depth and word width. Before compilation time, at compiler-install time, `ComponentResources` uses the FPGA tools to measure BlockRAM counts. This populates a table which represents $B(depth, width)$. Not every depth and word width must be measured by ComponentResources since $B(depth, width)$ is constant over a range of depths and range of word widths. Only the boundaries between constant regions need to be found.

8.4 Composition to Full Designs

PEs composed with network switches to get the full GRAPAL application logic. GRAPAL application logic is then composed with inter-FPGA channel logic on all FPGA and with the MicroBlaze processor and host communication logic on the master FPGA. Resources used for the full design may be slightly different than the sum over components. Clock frequency may be different than the minimum clock frequency over components. The `FullDesignFit` algorithm refines logic parameters, memory parameters and frequency until a legal design is produced.

The first iteration of `FullDesignFit` compiles the full design with the parameters chosen by LogicParameterChooser and MemoryParameterChooser. These are the number of PEs on each FPGA, $N_{PEs}^{(i)}$, the flit width of the interconnect, W_{flit}, PE node memory depth, D_{nodes}, and PE edge memory depth D_{edges}. It also uses the minimum clock frequency over the logic components chosen by LogicParameterChooser. Each iteration runs the FPGA tool chain separately for each FPGA with the current parameters. For each FPGA, the FPGA tool chain wrapper may return success or failure with the reason for failure. If success is returned for all FPGAs then `FullDesignFit` is finished and the HARDWARE compilation stage is complete (Figure 5.1). The reason for failure may be that excessive logic-pairs were used, excessive BlockRAMs were used, or the requested clock frequency was too high. If excessive logic-pairs were used on FPGA i then $N_{PEs}^{(i)}$ is multiplied by 0.9. Also, MemoryParameterChooser is run again to get a suitable D_{nodes} and D_{edges} for the new $N_{PEs}^{(i)}$. If the design fits in logic on all FPGAs but used excessive BlockRAMs then both D_{nodes} and D_{edges} are multiplied by 0.9. If the design fit in logic and

BlockRAMs on all FPGAs but the requested clock frequency was not met, then the lower feasible clock frequency found by the FPGA router tool is used for the next iteration. On the other hand, if the design fit and the requested clock frequency was less than 0.9 times the feasible clock frequency then the higher feasible clock frequency is used for the next iteration.

Compilation time of each full design for each FPGA through the FPGA tool chain is typically 1 hour, but can take up to 4 hours on a 3 GHz Intel Xeon. Further, one design must be compiled for each of the four FPGA on the BEE3 platform. So full design compilation dominates the runtime of the compiler and it is critical to minimize the number of iterations of `FullDesignFit`.

For all applications the first iteration of `FullDesignFit` is successful. This means that models used by `LogicParameterChooser` and `MemoryParameterChooser` do not underestimate resource usage. Across applications, the FPGAs' logic-pairs utilized is 95% to 97%. BlockRAMs utilized is 89% to 94%. So our models overestimate resource usage by at most 5% for logic-pairs and 11% for BlockRAMs.

Chapter 9

Conclusion

9.1 Lessons

9.1.1 Importance of Runtime Optimizations

Node decomposition to enable load-balancing is the most important runtime optimization we explored. It allows us to fit larger graphs in our architecture of many, small PEs than would otherwise fit. This allows us to always allocate as many PEs as possible so our compiler does not have to trade off between throughput of parallel operations and memory capacity for large nodes. Further, it provided a mean 2.6x speedup by load-balancing graphs that fit without decomposition.

Placement for locality mattered less than we expected. It only gave a 1.3x speedup for our benchmark applications and graphs. However, we expect placement for locality to provide greater speedups for larger systems with many more than four FPGAs.

9.1.2 Complex Algorithms in GRAPAL

We found that for the simple algorithms Bellman-Ford and ConceptNet, GRAPAL gives large speedups per chip of up to $7x$ and very large energy reductions of 10x to 100x. Our implementation effort in GRAPAL of the complex algorithms Spatial Router and Push-Relabel did not yield per-chip speedups, though Spatial Router does reduce energy cost by 3x to 30x. We have yet to see whether GRAPAL implementations of Spatial Router and Push-Relabel can be further optimized increase performance (Section 9.2.2).

9.1.3 Implementing the Compiler

Our compiler bridges a large gap between GRAPAL source programs and fully functioning FPGA logic. The core of the compiler that checks the source program and transforms it into HDL operators (Section 5.1) is an important part of the design of GRAPAL but is a relatively small part of the implementation effort. Implementing the parameterized logic architecture (Section 5.2) of PEs and switches was more complex. Our customized library for constructing HDL operators helped us manage the complexity of a dataflow-style architecture that is parameterized by the GRAPAL program. Managing BEE3 platform-level details was a big part of implementation work: Implementing inter-chip communication channels, implementing the host PC to MicroBlaze communication channel, and supporting the sequential C program with the C standard library on the MicroBlaze processor are difficult pieces of work that we customized for the specific platform. For the compiler to support a different FPGA platform, many of these low-level customizations need to be changed. Another difficult piece of the compiler is wrapping FPGA tools with a simple interface. An API interface to FPGA tools at the HDL level would save much effort in reformatting and semantics-discovery for anyone who wants to wrap FPGA tools into a automated, closed-loop.

9.2 Future Work

9.2.1 Extensions to GRAPAL

GRAPAL can be extended to make debugging more convenient and to enable simpler programs without sacrificing its efficient mapping to FPGA logic.

Assertion or exceptions would be a simple and very useful extension to GRAPAL. No changes to the logic architecture are required for exceptions. Instead an extra transformation by the compiler would reduce GRAPAL with exceptions to GRAPAL without exceptions: Every global reduce includes an error code that tells the sequential controller what error, if any, occurred in the previous graph-step. Messages are then augmented with error codes to transmit a failure in any operation to the global reduce.

Both Push-Relabel (Section 4.4) and Spatial Router (Section 4.3) define `node send` methods that always receive a single message from a single edge in each graph-step. Although a single message should not need to be reduced with a `node reduce tree` method, GRAPAL enforces the constraint that a `node reduce tree` method must be between the `edge fwd` method and the `node send` method (See Table 3.1). This constraint means the implementation always knows what to do if multiple messages arrive. Currently, the programmer must provide `node reduce tree` methods that do nothing. An extension to GRAPAL would allow an `edge fwd` method to send directly to a `node send` method. This extension would require checking at runtime to ensure that only one message arrived. Only changes to the compiler are required for this runtime check as long as exceptions are already supported.

9.2.2 Improvement of Applications

The GRAPAL applications Spatial Router and Push-Relabel offer little or no speedup per chip over the highly optimized sequential applications to which they were compared (Section 4.5). We spent little time tuning and optimizing the Spatial Router and Push-Relabel GRAPAL programs, so more work tuning heuristics may deliver a performance advantage per chip. To improve Push-Relabel performance, techniques from modern parallel implementations [75, 76, 77, 78] should be evaluated for GRAPAL. The current implementation of Spatial Router routes one source, sink pair at a time, which results in a low degree of parallelism and a low edge-activation rate. Spatial Router can be parallelized to a greater degree by finding routes to multiple sinks in parallel. This may be accomplished by searching in mutually-exclusive regions or by assigning priorities to searches so low-priority searches yield to high-priority searches.

9.2.3 Logic Sharing Between Methods

The HDL operators that GRAPAL methods are compiled to can be optimized further to decrease logic resources used. Since each GRAPAL method is feed-forward, it closely corresponds to HDL, which allows it to be optimized after our compiler outputs HDL

by the logic synthesis stage (Synplify Pro in Figure 5.1). A separate HDL module is allocated for each method, with all methods of the same kind mapped to the same large operator (Figure 5.10). For example, logic generated for all `node send` methods in all classes are multiplexed into the same large Node Update Operator. Only one of these method-operators is active in a cycle, with all other idle method-operators wasting area. The compiler should be extended with an optimization that shares logic between methods. In particular, floating point adders and multipliers take large amounts of logic and are easy units for a logic-sharing optimization to identify. This should be performed by the GRAPAL compiler because ordinary logic synthesis on HDL is not advanced enough to share logic between methods.

9.2.4 Improvements to the Logic Architecture

The clock frequency of most applications we tested is 100 MHz, which is much less than the maximum frequency permitted by our target Virtex-5 FPGA. BlockRAM memories are the fundamental limiting factor, whose maximum frequency is 450 MHz. First, careful analysis of PEs and network switches is required to improve the frequency. Second, method logic may need to be pipelined as well to adapt to methods with many operations on paths between their inputs and outputs. Third, long channels between network switches should be pipelined. To pipeline network switches effectively, the compiler should first place PEs and switches in the two-dimensional FPGA fabric so it knows the distance between each pair of connected switches. The compiler then uses the distance to calculate the number of registers to add to each channel.

9.2.5 Targeting Other Parallel Platforms

Runtimes and backends for the GRAPAL compiler could be developed that target MPI clusters, SMP machines, or GPUs. Like the FPGA implementation, these could take advantage of the GraphStep structure to perform node decomposition (Section 7.2), placement for locality (Section 7.4), and our mixed sparse and dense message synchronization style (Section 7.3).

Bibliography

[1] M. deLorimier, N. Kapre, N. Mehta, D. Rizzo, I. Eslick, R. Rubin, T. E. Uribe, T. F. Knight, Jr., and A. DeHon, "GraphStep: A System Architecture for Sparse-Graph Algorithms," in *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*. IEEE, 2006, pp. 143–151.

[2] M. deLorimier, N. Kapre, N. Mehta, and A. DeHon, "Spatial hardware implementation for sparse graph algorithms in graphstep," *ACM Transactions on Autonomous and Adapative Systems*, vol. 6, no. 3, pp. 17:1–17:20, September 2011.

[3] T. M. Parks, "Bounded scheduling of process networks," UCB/ERL95-105, University of California at Berkeley, 1995.

[4] S. Fortune and J. Wyllie, "Parallelism in random access machines," in *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing*. New York, NY, USA: ACM, 1978, pp. 114–118.

[5] L. G. Valiant, "A bridging model for parallel computation," *Communications of the ACM*, vol. 33, no. 8, pp. 103–111, August 1990.

[6] B. Parhami, *Introduction to Parallel Processing: Algorithms and Architectures*. Kluwer Academic Publishers, 1999.

[7] Xilinx, *Virtex-5 FPGA Data Sheet: DC and Switching Characteristics*, <http://www.xilinx.com/support/documentation/data_sheets/ds202.pdf>.

[8] Intel, *Intel 64 and IA-32 Architectures Optimization Reference Manual*, <http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html>.

[9] Xilinx, *Achieving Higher System Performance with the Virtex-5 Family of FPGAs*, <http://www.xilinx.com/support/documentation/white_papers/wp245.pdf>, 2006.

[10] P. B. Minev and V. S. Kukenska, "The virtex-5 routing and logic architecture," *Annual Journal of Electronics — ET*, vol. 3, pp. 107–110, September 2009.

[11] A. Rodrigues, K. B. Wheeler, P. M. Kogge, and K. D. Underwood, "Fine-grained message pipelining for improved mpi performance," in *CLUSTER*, 2006.

[12] T. Cormen, C. Leiserson, and R. Rivest, *Introduction to Algorithms*. MIT Press, 1990.

[13] K. M. Chandy and J. Misra, "Distributed computation on graphs: shortest path algorithms," *Commun. ACM*, vol. 25, no. 11, pp. 833–837, 1982.

[14] G. A. Kildall, "A unified approach to global program optimization," in *POPL '73: Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. New York, NY, USA: ACM Press, 1973, pp. 194–206.

[15] M. R. Hestenes and E. Stiefel, "Methods of conjugate gradients for solving linear systems," *J. Res. Nat. Bur. Stand.*, vol. 49, no. 6, pp. 409–436, 1952.

[16] L. N. Trefethen and D. Bau, *Numerical Linear Algebra*. SIAM: Society for Industrial and Applied Mathematics, 1997.

[17] A. Ralston, *A First Course in Numerical Analysis (1st ed.)*, ser. International Series in Pure and Applied Mathematics. New York, NY: McGraw-Hill, 1965.

[18] C. C. Paige and M. A. Saunders, "Solution of sparse indefinite systems of linear equations," *SIAM Journal on Numerical Analysis*, vol. 12, no. 4, pp. 617–629, 1975.

[19] A. DeHon, R. Huang, and J. Wawrzynek, "Stochastic Spatial Routing for Reconfigurable Networks," *Journal of Microprocessors and Microsystems*, vol. 30, no. 6, pp. 301–318, September 2006.

[20] M. Wrighton and A. DeHon, "Hardware-Assisted Simulated Annealing with Application for Fast FPGA Placement," in *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, February 2003, pp. 33–42.

[21] S. E. Fahlman, *NETL: A System for Representing and Using Real-World Knowledge*. MIT Press, 1979.

[22] J.-T. Kim and D. I. Moldovan, "Classification and retrieval of knowledge on a parallel marker-passing architecture," *IEEE Transactions on Knowledge and Data Engineering*, vol. 5, no. 5, pp. 753–761, October 1993.

[23] H. Liu and P. Singh, "ConceptNet – A Practical Commonsense Reasoning Tool-Kit," *BT Technical Journal*, vol. 22, no. 4, p. 211, October 2004.

[24] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web." Stanford InfoLab, Technical Report 1999-66, November 1999.

[25] H. Lieberman, *Concurrent object-oriented programming in Act 1*. Cambridge, MA, USA: MIT Press, 1987.

[26] G. Agha, *ACTORS: A Model of Concurrent Computation in Distributed Systems*. Cambridge: MIT Press, 1998.

[27] R. Milner, J. Parrow, and D. Walker, "A calculus of mobile processes, parts i and ii," *Information and Computation*, vol. 100, no. 1, pp. 1–77, September 1992.

[28] G. Kahn, "The semantics of a simple language for parallel programming," in *Proceedings of the IFIP CONGRESS 74*. North-Holland Publishing Company, 1974, pp. 471–475.

[29] E. Caspi, M. Chu, R. Huang, N. Weaver, J. Yeh, J. Wawrzynek, and A. DeHon, "Stream computations organized for reconfigurable execution (SCORE): Extended abstract," in *Proceedings of the International Conference on Field-Programmable*

Logic and Applications, ser. LNCS. Springer-Verlag, August 28–30 2000, pp. 605–614.

[30] E. Lee, "UC Berkley ptolemy project," <http://www.ptolemy.eecs.berkeley.edu/>, 2005.

[31] E. A. Lee and D. G. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, September 1987.

[32] "Brook project web page," <http://brook.sourceforge.net>, 2004.

[33] N. Shah, W. Plishker, K. Ravindran, and K. Keutzer, "NP-Click: A productive software development approach for network processors," *IEEE Micro*, vol. 24, no. 5, pp. 45–54, September 2004.

[34] J. M. D. Hill, B. Mccoll, D. C. Stefanescu, M. W. Goudreau, K. Lang, S. B. Rao, T. Suel, T. Tsantilas, and R. Bisseling, "Bsplib — the bsp programming library," *Parallel Computing*, vol. 24, 1997.

[35] "Bsponmpi," <http://bsponmpi.sourceforge.net/>.

[36] F. Loulergue, F. Gava, and D. Billiet, "Bulk synchronous parallel ml: Modular implementation and performance prediction," in *International Conference on Computational Science*. Springer, 2005, pp. 1046–1054.

[37] E. Lusk, N. Doss, and A. Skjellum, "A high-performance, portable implementation of the mpi message passing interface standard," *Parallel Computing*, vol. 22, pp. 789–828, 1996.

[38] G. E. Blelloch, S. Chatterjee, J. C. Hardwick, J. Sipelstein, and M. Zagha, "Implementation of a portable nested data-parallel language," in *Proceedings 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, San Diego, May 1993, pp. 102–111.

[39] C. H. Koelbel, D. B. Loveman, R. S. Schreiber, J. Guy L. Steele, and M. E. Zosel, *The high performance Fortran handbook*. Cambridge, MA, USA: MIT Press, 1994.

[40] W. D. Hillis, *The Connection Machine*, ser. Distinguished Dissertations. MIT Press, 1985.

[41] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *Proceedings of the Symposium on Operating System Design and Implementation*, 2004, pp. 137–150.

[42] W. D. Hillis and G. L. Steele, "Data parallel algorithms," *Communications of the ACM*, vol. 29, no. 12, pp. 1170–1183, December 1986.

[43] Khronos OpenCL Working Group, *The OpenCL Specification, version 1.0.29*, <http://khronos.org/registry/cl/specs/opencl-1.0.29.pdf>, 2008.

[44] I. B. J Nickolls, "Nvidia cuda software and gpu parallel computing architecture," Microprocessor Forum, May 2007.

[45] L. Buatois, G. Caumon, and B. Lvy, "Concurrent number cruncher: An efficient sparse linear solver on the gpu," in *High Performance Computation Conference (HPCC), Springer Lecture Notes in Computer Sciences*, 2007.

[46] A. Cevahir, A. Nukada, and S. Matsuoka, "High performance conjugate gradient solver on multi-gpu clusters using hypergraph partitioning," *Computer Science - R&D*, vol. 25, no. 1-2, pp. 83–91, 2010.

[47] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, and C. Guestrin, "Graphlab: A distributed framework for machine learning in the cloud," *CoRR*, vol. abs/1107.0922, 2011.

[48] D. Borthakur, *The Hadoop Distributed File System: Architecture and Design*, The Apache Software Foundation, 2007.

[49] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the 2010 International Conference on Management of Data*, ser. SIGMOD '10. ACM, 2010, pp. 135–146.

[50] P. Stutz, A. Bernstein, and W. Cohen, "Signal/collect: graph algorithms for the (semantic) web," in *Proceedings of the 9th International Semantic Web Conference on the Semantic Web — Volume I*, ser. ISWC'10. Springer-Verlag, 2010, pp. 764–780.

[51] A. Buluç and J. R. Gilbert, "The Combinatorial BLAS: Design, implementation, and applications," *The International Journal of High Performance Computing Applications*, 2011.

[52] M. Aubury, I. Page, G. Randall, J. Saul, and R. Watts, "Handel-c language reference guide," 1996.

[53] D. Galloway, "The transmogrifier c hardware description language and compiler for fpgas," in *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, April 1995, pp. 136–144.

[54] M. Gokhale, J. Stone, J. Arnold, and M. Kalinoskwi, "Stream-oriented fpga computing in the streams-c high level lanugage," in *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*. IEEE, April 2000.

[55] D. C. Cronquist, P. Franklin, S. G. Berg, and C. Ebeling, "Specifying and compiling applications for rapid," in *FCCM*, 1998, pp. 116–125.

[56] *System C 2.1 Language Reference Manual*, <http://www.systemc.org>, Open System C Initiative, May 2005.

[57] T. Bollaert, "Catapult Synthesis: A Practical Introduction to Interactive C Synthesis High-Level Synthesis," in *High-Level Synthesis*, P. Coussy and A. Morawiec, Eds. Springer Netherlands, 2008, ch. 3, pp. 29–52.

[58] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. A. Vissers, and Z. Zhang, "High-level synthesis for fpgas: From prototyping to deployment," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 30, no. 4, pp. 473–491, 2011.

[59] C. A. R. Hoare, *Communicating Sequential Processes*, ser. International Series in Computer Science. Prentice-Hall, 1985.

[60] T. Callahan, J. Hauser, and J. Wawrzynek, "The Garp Architecture and C Compiler," *IEEE Computer*, vol. 33, no. 4, pp. 62–69, April 2000.

[61] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann, "Effective compiler support for predicated execution using the hyperblock," in *Proceedings of Micro-25*, 1992, pp. 45–54.

[62] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski, "Legup: high-level synthesis for fpga-based processor/accelerator systems," in *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*. ACM, 2011, pp. 33–36.

[63] D. Lau, O. Pritchard, and P. Molson, "Automated generation of hardware accelerators with direct memory access from ansi/iso standard c functions," in *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2006, pp. 45–56.

[64] C. Leiserson, F. Rose, and J. Saxe, "Optimizing synchronous circuitry by retiming," in *Third Caltech Conference On VLSI*, March 1983.

[65] V. Betz and J. Rose, "FPGA Place-and-Route Challenge," <http://www.eecg.toronto.edu/ vaughn/challenge/challenge.html>, 1999.

[66] A. DeHon, R. Huang, and J. Wawrzynek, "Hardware-Assisted Fast Routing," in *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2002, pp. 205–215.

[67] R. Huang, J. Wawrzynek, and A. DeHon, "Stochastic, Spatial Routing for Hypergraphs, Trees, and Meshes," in *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, February 2003, pp. 78–87.

[68] L. McMurchie and C. Ebeling, "PathFinder: A Negotiation-Based Performance-Driven Router for FPGAs," in *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, 1995, pp. 111–117.

[69] K. McElvain, "LGSynth93 Benchmark Set: Version 4.0," <http://www.cbl.ncsu.edu/pub/Benchmark_dirs/LGSynth93/doc/iwls93.ps>, May 1993.

[70] V. Betz and J. Rose, "VPR: A new packing, placement, and routing tool for FPGA research," in *Proceedings of the International Conference on Field-Programmable Logic and Applications*, ser. LNCS, W. Luk, P. Y. K. Cheung, and M. Glesner, Eds., no. 1304. Springer, August 1997, pp. 213–222.

[71] J. Rose *et al.*, "VPR and T-VPack: Versatile Packing, Placement and Routing for FPGAs," <http://www.eecg.utoronto.ca/vpr/>, 2008.

[72] A. V. Goldberg and R. E. Tarjan, "A new approach to the maximum flow problem," in *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing*, ser. STOC '86. ACM, 1986, pp. 136–146.

[73] A. Goldberg, *A New Max-flow Algorithm*, ser. MIT/LCS/TM-. Laboratory for Computer Science, Massachusetts Institute of Technology, 1985.

[74] B. V. Cherkassy and A. V. Goldberg, "On implementing push-relabel method for the maximum flow problem," in *Proceedings of the 4th International IPCO Conference on Integer Programming and Combinatorial Optimization*. Springer-Verlag, 1995, pp. 157–171.

[75] R. J. Anderson and J. a. C. Setubal, "On the parallel implementation of goldberg's maximum flow algorithm," in *Proceedings of the Fourth Annual ACM Symposium on Parallel Algorithms and Architectures*, ser. SPAA '92. ACM, 1992, pp. 168–177.

[76] D. A. Bader and V. Sachdeva, "A cache-aware parallel implementation of the push-relabel network flow algorithm and experimental evaluation of the gap relabeling heuristic," in *ISCA PDCS*, 2005, pp. 41–48.

[77] Y. Lu, H. Zhou, L. Shang, and X. Zeng, "Multicore parallel min-cost flow algorithm for cad applications," in *Proceedings of the 46th Annual Design Automation Conference*, ser. DAC '09. ACM, 2009, pp. 832–837.

[78] Z. He and B. Hong, "Dynamically tuned push-relabel algorithm for the maximum flow problem on cpu-gpu-hybrid platforms," in *IPDPS*, 2010, pp. 1–10.

[79] Xilinx, *MicroBlaze Processor Reference Guide*, <http://www.xilinx.com/tools/microblaze.htm>.

[80] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Transactions on Programming Languages and Systems*, vol. 13, no. 4, pp. 451–490, Oct. 1991.

[81] R. Wilson, R. French, C. Wilson, S. Amarasinghe, J. Anderson, S. Tjiang, S. Liao, C. Tseng, M. Hall, M. Lam, and J. Hennessy, "The suif compiler system: a parallelizing and optimizing research compiler," Computer Systems Lab, Stanford University, Tech. Rep., 1994.

[82] P. Bellows and B. Hutchings, "JHDL — an HDL for reconfigurable systems," in *IEEE Symposium on FPGAs for Custom Computing Machines*, K. L. Pocek and J. Arnold, Eds. Los Alamitos, CA: IEEE Computer Society Press, 1998, pp. 175–184.

[83] M. Chu, N. Weaver, K. Sulimma, A. DeHon, and J. Wawrzynek, "Object Oriented Circuit-Generators in Java," in *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, April 1998, pp. 158–166.

[84] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh, "Lava: hardware design in haskell," *SIGPLAN Not.*, vol. 34, no. 1, pp. 174–184, Sep. 1998.

[85] A. Gill, T. Bull, G. Kimmell, E. Perrins, E. Komp, and B. Werling, "Introducing kansas lava," in *Implementation and Application of Functional Languages*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2011, vol. 6041, pp. 18–35.

[86] N. Kapre, N. Mehta, M. deLorimier, R. Rubin, H. Barnor, M. J. Wilson, M. Wrighton, and A. DeHon, "Packet-Switched vs. Time-Multiplexed FPGA Overlay Networks,"

in *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*. IEEE, 2006, pp. 205–213.

[87] B. S. Landman and R. L. Russo, "On pin versus block relationship for partitions of logic circuits," *IEEE Transactions on Computers*, vol. 20, pp. 1469–1479, 1971.

[88] A. Caldwell, A. Kahng, and I. Markov, "Improved Algorithms for Hypergraph Bipartitioning," in *Proceedings of the Asia and South Pacific Design Automation Conference*, January 2000, pp. 661–666.

Appendix A

GRAPAL Context-Free Grammar

```
Program ::= (Class | Function)*
Class ::= GlobalClass | NodeClass | EdgeClass
GlobalClass ::= global Symbol { (OutDecl | GlobalMethod)* }
NodeClass ::= node Symbol { (OutDecl | FieldDecl | NodeMethod)* }
EdgeClass ::= node Symbol { (OutDecl | FieldDecl | EdgeMethod)* }
OutDecl ::= out Symbol Symbol ;
FieldDecl := ValueType Symbol ;
GlobalMethod ::= BcastMethod | GlobalReduceMethod
NodeMethod ::= SendMethod | NodeReduceMethod
EdgeMethod ::= FwdMethod
BcastMethod ::= bcast Symbol ParamList Symbol . Symbol
GlobalReduceMethod ::= reduce tree Symbol ParamList Value StatementBlock
SendMethod ::= send Symbol ParamList StatementBlock
NodeReduceMethod ::= reduce tree Symbol ParamList StatementBlock
FwdMethod ::= fwd Symbol ParamList StatementBlock
Function ::= ValueType Symbol ParamList StatementBlock
ParamList ::= ( ) | ( Param (, Param)* )
Param ::= ValueType Symbol
Statement ::= VarDecl | Assignment | Dispatch | Return | IfThenElse
  | StatementBlock
Block ::= { Statement* }
VarDecl ::= ValueType Symbol = Expression ;
Assignment ::= LeftExpression = Expression ;
Dispatch ::= Symbol . Symbol ArgList ;
Return ::= return Value ;
IfThenElse ::= if Value Statement Statement?
LeftExpression ::= LeftTuple | TupleIndex
LeftTuple ::= Symbol | ( ) | ( LeftTuple (, LeftTuple)* )
Expression ::= Symbol | Group | TupleIndex | Tuple | FunctionCall
  | Conditional | UnaryExpr | BinaryExpr
ArgList ::= ( ) | ( Expression (, Expression)* )
Group ::= ( Expression )
Tuple ::= ( ) | ( Expression (, Expression)* )
TupleIndex ::= Symbol [ Natural ]
FunctionCall ::= Symbol ArgList
Conditional ::= Expression ? Expression : Expression
UnaryExpr ::= (! | - | ~) Expression
BinaryExpr ::= Expression BinaryOp Expression
BinaryOp ::= (== | && | || | ^ | & | | | + | - | * | / | % | << | >>)
ValueType ::= boolean | int < Natural > | unsigned < Natural > | TupleType
TupleType ::= ( ) | ( ValueType (, ValueType)* )
Value ::= Boolean | Integer | Decimal | Tuple
TupleValue ::= ( ) | ( Value (, Value)* )
Symbol ::= [_a-z][_a-z0-9]*
Natural ::= [0-9]+
Integer ::= -?[0-9]+
Decimal ::= -?[0-9]+(.[0-9]+)?
```

Figure A.1: Context-free grammar for GRAPAL: Nonterminals are green, literal terminals are red, and terminals expressed as regular expressions are blue.

Appendix B

Push-Relabel in GRAPAL

```
global Glob {
  out Node nodes;

  bcast get_sink_overflow_b() nodes.get_sink_overflow;
  reduce tree get_sink_overflow_r(int<24> x, int<24> y) 0 { return x + y; }

  bcast init_start(int<24>) nodes.init_start;
  bcast global_relabel_start() nodes.global_relabel_start;

  bcast push_start() nodes.push_start;
  bcast relabel_start() nodes.relabel_start;
  reduce tree is_active(boolean x1, boolean x2) false { return x1 || x2; }
}

edge Edge {
  out Node to;
  int<24> capacity, complement_capacity, flow;
  int<24> idx_from, idx_to;

  fwd init_adj() {
    flow = capacity;
    to.init_adj(idx_from, capacity);
  }
  fwd init_back(int<24> idx_s) {
    if (idx_to == idx_s) flow = -complement_capacity;
  }

  fwd global_relabel(int<24> height) {
    if (complement_capacity != -flow) to.global_relabel(height + 1);
  }

  fwd push_request(int<24> height, int<24> overflow) {
    int<24> push_amount = min(capacity - flow, overflow);
    if (0 < push_amount) to.push_request(idx_from, height, push_amount);
  }
  fwd push_ack(int<24> idx_req, int<24> push_amount) {
    if (idx_req == idx_to) {
      to.push_ack(idx_from, push_amount);
    }
  }
  fwd push_do_fwd(int<24> idx_ack, int<24> push_amount) {
    if (idx_ack == idx_to) {
      flow = flow + push_amount;
      to.push_do(idx_from, push_amount);
    }
  }
  fwd push_do_back(int<24> idx_req, int<24> push_amount) {
    if (idx_req == idx_to) flow = flow - push_amount;
  }

  fwd relabel_height(int<24> height) {
      if (flow != -complement_capacity) { // complement edge is in residual network
      to.relabel_height(height);
    }
  }
}
```

Figure B.1: Push_Relabel Part 1 of 3

```
node Node {
  out Glob parent;
  out Edge edges;

  boolean is_source, is_sink;
  int<24> idx, height, overflow, nnodes;

  send get_sink_overflow() { if (is_sink) parent.get_sink_overflow_r(overflow); }

  send init_start(int<24> nnodes_in) {
    nnodes = nnodes_in;
    if (is_source) {
      height = max_int(); // not necessary if global_relabel run after init
      edges.init_adj();
    }
  }
  // only 1 message possible since there is 1 source and it's not a multigraph
  reduce tree init_adj(int<24> idx_s1, int<24> amount1,
                       int<24> idx_s2, int<24> amount2) {
    return (0, 0);
  }
  send init_adj(int<24> idx_s, int<24> amount) {
    overflow = amount;
    edges.init_back(idx_s);
  }

  send global_relabel_start() {
    if (is_sink) {
      height = 0;
      edges.global_relabel(height);
    } else {
      height = max_int();
    }
  }
  reduce tree global_relabel(int<24> height1, int<24> height2) {
    return min(height1, height2);
  }
  send global_relabel(int<24> n_height) {
    if (n_height < height) {
      height = n_height;
      edges.global_relabel(n_height);
    }
  }
```

Figure B.2: Push_Relabel Part 2 of 3

```
  send push_start() {
    if (!is_source && !is_sink && height < nnodes && 0 < overflow) {
      edges.push_request(height, overflow);
    }
  }

  // choose which neighbor to ack
  reduce tree push_request(int<24> idx_req1, int<24> height_req1, int<24> push_amount1,
                           int<24> idx_req2, int<24> height_req2, int<24> push_amount2) {
    if (idx_req1 < idx_req2) return (idx_req1, height_req1, push_amount1);
    else return (idx_req2, height_req2, push_amount2);
  }
  send push_request(int<24> idx_req, int<24> height_req, int<24> push_amount) {
    if (height_req == height + 1) {
      edges.push_ack(idx_req, push_amount);
    }
  }

  // choose which pushable edge to push on -- priority to lower flow,
  //                                          then priority to lower idx
  reduce tree push_ack(int<24> idx_ack1, int<24> push_amount1,
                       int<24> idx_ack2, int<24> push_amount2) {
    if (push_amount1 > push_amount2 ||
        (push_amount1 == push_amount2 && idx_ack1 < idx_ack2)) {
      return (idx_ack1, push_amount1);
    } else return (idx_ack2, push_amount2);
  }
  send push_ack(int<24> idx_ack, int<24> push_amount) {
    overflow = overflow - push_amount;
    edges.push_do_fwd(idx_ack, push_amount);
  }

  reduce tree push_do(int<24> idx_req1, int<24> push_amount1,
                      int<24> idx_req2, int<24> push_amount2) {
    return (0, 0); // there should be only 1 message, so this method should never fire
  }
  send push_do(int<24> idx_req, int<24> push_amount) {
    overflow = overflow + push_amount;
    edges.push_do_back(idx_req, push_amount);
  }

  send relabel_start() {
    edges.relabel_height(height);
  }
  reduce tree relabel_height(int<24> height1, int<24> height2) {
    return min(height1, height2);
  }
  send relabel_height(int<24> neighbor_height) {
    if (!is_source && !is_sink && height < nnodes && 0 < overflow) {
      height = neighbor_height + 1;
      parent.is_active(true);
    }
  }
}

int<24> max_int() { return 8388607; }

int<24> min(int<24> x, int<24> y) {
  return x < y ? x : y;
}
int<24> max(int<24> x, int<24> y) {
  return x > y ? x : y;
}
```

Figure B.3: Push_Relabel Part 3 of 3

Appendix C

Spatial Router in GRAPAL

```
unsigned<5> min(unsigned<5> x, unsigned<5> y) {
  return x < y ? x : y;
}

unsigned<4> inid_to_vlutin(unsigned<5> inid) {
  unsigned<4> one = 1;
  return one << inid;
}
unsigned<8> outid_to_vout(unsigned<5> outid) {
  unsigned<8> one = 1;
  return one << outid;
}
boolean vout_get_outid(unsigned<8> v, unsigned<5> outid) {
  unsigned<8> one = 1;
  return (v & (one << outid)) != 0;
}

global Glob {
  out Lut luts;
  out Mux muxs;

  // sets congestion delay for all muxs
  bcast set_congestion_delay(unsigned<10>) muxs.set_congestion_delay;

  // turn_current_net_routed_on(src)
  bcast turn_current_net_routed_on(unsigned<16>) luts.turn_current_net_routed_on;

  // called at begining of route set_pt2(src, sink, victimize)
  bcast set_pt2(unsigned<16>, unsigned<16>, boolean) luts.set_pt2;

  bcast search_no_victimize() muxs.start_search_no_victimize;
  bcast alloc_no_victimize() luts.alloc_no_victimize;

  bcast search_victimize() muxs.start_search_victimize;
  bcast alloc_victimize() luts.alloc_victimize;

  // gets only 1 msg
  reduce tree found_sink(boolean found1, unsigned<5> inid1, boolean found2,
                         unsigned<5> inid2)
    (false, 0) {
    if (found1) return (found1, inid1);
    else return (found2, inid2);
  }

  bcast query_victimized(boolean, unsigned<16>) luts.query_victimized;
  reduce tree answer_victimized(boolean is_victim1, unsigned<16> lut1,
                                unsigned<4> ins1, boolean is_victim2,
                                unsigned<16> lut2, unsigned<4> ins2)
    (false, 0, 0)
  {
    if (is_victim1 && (!is_victim2 || lut1 < lut2)) {
      return (is_victim1, lut1, ins1);
    } else return (is_victim2, lut2, ins2);
  }

  bcast fin_pt2() muxs.fin_pt2; // does this need to be different for (non)victimize?

  // fin_net(lock, set_congestion_delay, new_congestion_delay)
  bcast fin_net(boolean, boolean, unsigned<10>) muxs.fin_net;
}
```

Figure C.1: Spatial Router Part 1 of 6

```
node Lut {
  out Glob parent;
  out To_mux mouts; // muxs contained in this LUT
  out From_mux mins; // muxs outside this LUT in the IO box

  // constants
  unsigned<16> this_id;
  // variables
  boolean current_sink;
  unsigned<5> current_inid;
  unsigned<4> ripped_ins;

  // for nets which have been victimized and are being routed again
  send turn_current_net_routed_on(unsigned<16> source) {
    if (source == this_id) {
      mouts.turn_current_net_routed_on(true, 0);
    }
  }

  send set_pt2(unsigned<16> source, unsigned<16> sink, boolean victimize) {
    if (source == this_id) mouts.set_src_mux(victimize);
    current_sink = sink == this_id;
  }

  // search happens at a sink
  reduce tree search(unsigned<5> inid1, unsigned<5> inid2) {
    return min(inid1, inid2);
  }
  send search(unsigned<5> inid) {
    if (current_sink) {
      current_inid = inid;
      parent.found_sink(true, inid);
    }
  }

  send alloc_no_victimize() {
    if (current_sink) mins.alloc_no_victimize(current_inid);
  }

  send alloc_victimize() {
    if (current_sink) mins.alloc_victimize(false, 0, true, current_inid);
  }

  reduce tree rip_fwd(unsigned<4> ins1, unsigned<4> ins2) {
    return ins1 | ins2;
  }
  send rip_fwd(unsigned<4> ins) {
    ripped_ins = ripped_ins | ins;
  }

  send query_victimized(boolean exists_last_lut, unsigned<16> last_lut) {
    if (exists_last_lut && last_lut == this_id) ripped_ins = 0;
    else if (ripped_ins != 0) parent.answer_victimized(true, this_id, ripped_ins);
  }
}
```

Figure C.2: Spatial Router Part 2 of 6

```
node Mux {
  out To_mux mouts;
  out To_lut lout; // should appear where mouts appears
  out From_mux mins;
  out Self_mux self;

  unsigned<5> current_inid, routed_inid;
  // current_net_routed => routed
  boolean current_search, current_net_routed, routed, locked;
  unsigned<5> fanout_count;
  unsigned<8> mouts_on_route; // mouts_on_route != 0 <=> routed && (routed to mux)
  unsigned<10> congestion_delay;

  send set_congestion_delay(unsigned<10> congestion_delay_in) {
    congestion_delay = congestion_delay_in;
  }

  reduce tree turn_current_net_routed_on() { return (); } // 1 in msg
  send turn_current_net_routed_on() {
    // this mux is a source => start on routed muxs only;
    //   not source => routed should always be true
    if (routed) {
      current_net_routed = true;
      mouts.turn_current_net_routed_on(false, mouts_on_route);
    }
  }

  reduce tree set_src_mux(boolean v1, boolean v2) { return v1; } // 1 in msg
  send set_src_mux(boolean victimize) {
    if ((victimize && !locked) || current_net_routed || !routed) current_search = true;
  }

  // ************ non victimizing ************

  send start_search_no_victimize() {
    if (current_net_routed || current_search) {
      current_search = true;
      mouts.search_no_victimize();
      lout.search();
    }
  }
  reduce tree search_no_victimize(unsigned<5> inid1, unsigned<5> inid2) {
    return min(inid1, inid2);
  }
  send search_no_victimize(unsigned<5> inid) {
    if (!routed && !current_search) {
      current_search = true;
      routed_inid = inid;
      mouts.search_no_victimize();
      lout.search();
    }
  }

  // only gets 1 message
  reduce tree alloc_no_victimize(unsigned<5> outid1, unsigned<5> outid2) {
    return outid1;
  }
  send alloc_no_victimize(unsigned<5> outid) {
    if (current_net_routed) { // increment fanout
      fanout_count = fanout_count + 1;
      mouts_on_route = mouts_on_route | outid_to_vout(outid);
    } else { // set new route
      routed = true;
      current_net_routed = true;
      fanout_count = 1;
      mins.alloc_no_victimize(routed_inid);
      mouts_on_route = outid_to_vout(outid);
    }
  }
```

Figure C.3: Spatial Router Part 3 of 6

```
// ************ victimizing ************

send start_search_victimize() {
  if (current_net_routed || current_search) {
    current_search = true;
    mouts.search_victimize();
    lout.search();
  }
}

// keep whichever one has been delayed the most;
//   if neither then break ties with min
reduce tree search_victimize(unsigned<10> delay_cnt1, unsigned<5> inid1,
              unsigned<10> delay_cnt2, unsigned<5> inid2) {
  if (delay_cnt1 > 0) return (delay_cnt1, inid1);
  else if (delay_cnt2 > 0) return (delay_cnt2, inid2);
  else return (0, min(inid1, inid2));
}
send search_victimize(unsigned<10> delay_cnt, unsigned<5> inid) {
  if (!locked && (!current_search || delay_cnt > 0)) {
    current_search = true;
    current_inid = inid;
    if (routed && delay_cnt < congestion_delay && routed_inid != inid) {
      self.search_delay(delay_cnt + 1, inid);
    } else {
      mouts.search_victimize();
      lout.search();
    }
  }
}

reduce tree alloc_victimize(boolean recv_rip_fwd1,
              unsigned<5> recv_rip_back_cnt1, unsigned<8> recv_rip_back_vout1,
              boolean recv_alloc1, unsigned<5> alloc_outid1,
              boolean recv_rip_fwd2,
              unsigned<5> recv_rip_back_cnt2, unsigned<8> recv_rip_back_vout2,
              boolean recv_alloc2, unsigned<5> alloc_outid2) {
  boolean recv_rip_fwd = recv_rip_fwd1 || recv_rip_fwd2;
  unsigned<5> recv_rip_back_cnt = recv_rip_back_cnt1 + recv_rip_back_cnt2;
  unsigned<8> recv_rip_back_vout = recv_rip_back_vout1 | recv_rip_back_vout2;
  boolean recv_alloc = recv_alloc1 || recv_alloc2;
  unsigned<5> alloc_outid = recv_alloc1 ? alloc_outid1 : alloc_outid2;
  return (recv_rip_fwd, recv_rip_back_cnt, recv_rip_back_vout,
          recv_alloc, alloc_outid);
}
```

Figure C.4: Spatial Router Part 4 of 6

```
  send alloc_victimize(boolean recv_rip_fwd,
             unsigned<5> recv_rip_back_cnt, unsigned<8> recv_rip_back_vout,
             boolean recv_alloc, unsigned<5> alloc_outid) {
    boolean recv_rip_back = recv_rip_back_cnt > 0;
    boolean old_routed = routed && !current_net_routed;
    unsigned<8> old_mouts_on_route = mouts_on_route;
    unsigned<5> rip_back_remaining_fanout = fanout_count - recv_rip_back_cnt;

    boolean send_rip_back =
      old_routed && !recv_rip_fwd &&
      ((recv_rip_back && rip_back_remaining_fanout == 0) ||
       recv_alloc);
    boolean send_rip_fwd = old_routed && (recv_rip_fwd || recv_alloc);
    boolean send_alloc = !current_net_routed && recv_alloc;

    unsigned<5> rip_back_inid = routed_inid;
    unsigned<5> alloc_inid = current_inid;

    if (recv_alloc) {
      if (current_net_routed) { // increment fanout
        fanout_count = fanout_count + 1;
        mouts_on_route = mouts_on_route | outid_to_vout(alloc_outid);
      } else { // set new route
        routed = true;
        current_net_routed = true;
        fanout_count = 1;
        mouts_on_route = outid_to_vout(alloc_outid);
        routed_inid = current_inid;
      }
    } else if (old_routed && recv_rip_fwd) { // clear old route
      routed = false;
      fanout_count = 0;
      mouts_on_route = 0;
    } else if (old_routed && recv_rip_back) {
      // decrement fanout and possibly clear old route
      if (rip_back_remaining_fanout == 0) routed = false;
      fanout_count = rip_back_remaining_fanout;
      mouts_on_route = mouts_on_route & ~recv_rip_back_vout;
    }

    if (send_rip_back || send_alloc) { // rip_back and/or alloc
      mins.alloc_victimize(send_rip_back, rip_back_inid, send_alloc, alloc_inid);
    }

    if (send_rip_fwd) {
      lout.rip_fwd();
      mouts.rip_fwd(old_mouts_on_route);
    }
  }

  // ************ cleanup ************

  send fin_pt2() {
    current_search = false;
  }

  send fin_net(boolean lock, boolean set_new_congestion_delay,
               unsigned<10> congestion_delay_in) {
    if (current_net_routed) {
      locked = lock;
      if (set_new_congestion_delay) congestion_delay = congestion_delay_in;
    }
    current_net_routed = false;
  }
}
```

Figure C.5: Spatial Router Part 5 of 6

```
edge To_mux {
  out Mux to;
  unsigned<5> outid, inid;

  fwd turn_current_net_routed_on(boolean to_source, unsigned<8> mouts_on_route) {
    if (to_source || vout_get_outid(mouts_on_route, outid))
      to.turn_current_net_routed_on();
  }

  fwd set_src_mux(boolean victimize) { to.set_src_mux(victimize); }

  fwd search_no_victimize() { to.search_no_victimize(inid); }

  fwd search_victimize() { to.search_victimize(0, inid); }

  fwd rip_fwd(unsigned<8> mouts_on_route) {
    if (vout_get_outid(mouts_on_route, outid))
      to.alloc_victimize(true, 0, 0, false, 0);
  }
}

edge From_mux {
  out Mux from;
  unsigned<5> outid, inid; // same as its complement

  fwd alloc_no_victimize(unsigned<5> recv_inid) {
    if (inid == recv_inid) from.alloc_no_victimize(outid);
  }
  fwd alloc_victimize(boolean rip_back, unsigned<5> rip_back_inid, boolean alloc,
                      unsigned<5> alloc_inid) {
    boolean send_rip_back = rip_back && inid == rip_back_inid;
    unsigned<5> rip_back_cnt = send_rip_back ? 1 : 0;
    unsigned<8> rip_back_vout = outid_to_vout(outid);
    boolean send_alloc = alloc && inid == alloc_inid;
    if (send_rip_back || send_alloc) {
      from.alloc_victimize(false, rip_back_cnt, rip_back_vout, send_alloc, outid);
    }
  }
}

edge Self_mux {
  out Mux to;

  fwd search_delay(unsigned<10> delay_cnt, unsigned<5> inid) {
    to.search_victimize(delay_cnt, inid); }
}

edge To_lut {
  out Lut to;
  unsigned<5> inid;

  fwd search() { to.search(inid); }
  fwd rip_fwd() { to.rip_fwd(inid_to_vlutin(inid)); }
}
```

Figure C.6: Spatial Router Part 6 of 6