Tadpole: An Off-line Router for the NuMesh System

by

Patrick Joseph LoPresti

Submitted to the Department of Electrical Engineering
and Computer Science in Partial Fulfillment of the
Requirements for the Degree of

Master of Science in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 1997

Copyright 1997 Massachusetts Institute of Technology. All rights reserved.

Author ....
Department of Electrical Engineering and Computer Science
February 20, 1997

Certified by ....
Stephen A. Ward
Thesis Supervisor

Accepted by ....
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Tadpole: An Off-line Router for the NuMesh System

by

Patrick Joseph LoPresti

Submitted to the Department of Electrical Engineering and Computer Science on February 20, 1997, in partial fulfillment of the requirements for the degree of Master of Science

Abstract

A framework for analyzing off-line scheduling and routing of fixed-period recurring communication requirements in a multicomputer interconnect is presented. This framework, which is based on multicommodity flow techniques, has been used to develop an off-line scheduling and routing system for the current generation of the NuMesh interconnect technology. Results from the system are presented.

Thesis Supervisor: Stephen A. Ward
Title: Professor of Electrical Engineering and Computer Science

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

The NuMesh project is an exploration of the potential for off-line routing to provide superior throughput for communication-bound multicomputer applications. Tadpole is a software system which takes a compile-time description of the communication needs of an application and produces a global schedule for the NuMesh hardware.

1.1 Off-line vs. On-line Routing

Off-line routing (also known as *static routing*) refers to the strategy of programming a multicomputer interconnect with communication patterns determined at compile time.

Many current multicomputer interconnects use on-line routing; that is, they adapt their behavior to the needs of the application at run time. Although they have the benefit of run-time information which might not be available at compile time, their algorithms are limited to being causal (they cannot know the future) and local (they cannot efficiently incorporate information from distant places in the interconnect). In contrast, although an off-line strategy may have limited information about the dynamic communication requirements of the application, what information it has can be applied globally, both in space and in time.

On-line techniques must be carefully crafted to avoid *deadlock*, where network traffic grinds to a halt because of a cyclic contention for network resources. The well-known *dimensional routing* strategy [Ta81] is a simple way to prevent deadlock in Euclidean meshes; more complex approaches are typically used in practice [DS87]. Off-line techniques, however, can always avoid deadlock merely by never allowing contention at all, giving them more flexibility in their routing choices. (Section 7.3 provides an example where dimensional routing in particular would produce poor results.)

Finally, off-line techniques allow for simpler (and therefore potentially faster) hardware implementations, since they do not incur the overhead of making routing decisions at run time [Sh97].

For a complete comparison of on-line and off-line routing, see [Me97].

1.2 Road Map of Thesis

Chapter 2 summarizes the NuMesh, the hardware for which Tadpole generates output.

Chapter 3 presents a framework for attacking the off-line routing problem in systems similar to the NuMesh. It is this framework which is most likely to be adaptable to other systems.

Chapter 4 fills in the framework with details necessary to cope with the hardware limitations of the current NuMesh implementation.

Chapter 5 and Chapter 6 discuss augmentations of the framework for supporting the additional NuMesh features of “high-bandwidth streams” and “multicast”.

Chapter 7 presents a collection of simple example routing problems and Tadpole’s performance on them.

Chapter 2

The NuMesh

A complete description of the NuMesh architecture is given in [Sh97]. The following is a minimal summary adequate for an understanding of Tadpole.

2.1 Overview

The NuMesh is a programmable *communication substrate* providing the intercommunication facility for a collection of *processing elements* (e.g. Sparc CPUs). The substrate provides virtual channels, which we call *streams*, between nodes. Each stream provides some fixed bandwidth of communication between a source and a destination.

The programming of the communication substrate determines the set of streams. Tadpole's job is to produce a program for the substrate given a set of communication requirements. Tadpole assumes that the substrate's program will be changed only by replacing it with a completely new program.[1]

2.2 Physical Structure

The NuMesh communication substrate consists of a mesh of nearest-neighbor interconnected nodes. The current-generation NuMesh has a four-neighbor three-dimensional "diamond lattice" structure, but Tadpole does not depend on this topology. In principle, Tadpole could be adapted to work on arbitrary graphs; the current implementation supports Euclidean meshes of up to three dimensions as well as the diamond lattice.

2.3 Nodes

Each node consists of a processing element and a *Communication Finite State Machine*

1. It is possible to make incremental run-time changes to the programming of the substrate, but it is not simple to do so; consequently, Tadpole does not support this feature. Of course, independent substrate programs may be generated by independent invocations of Tadpole.

(CFSM). The CFSMs are programmed independently and manage all communication between nodes (Figure 2.1).

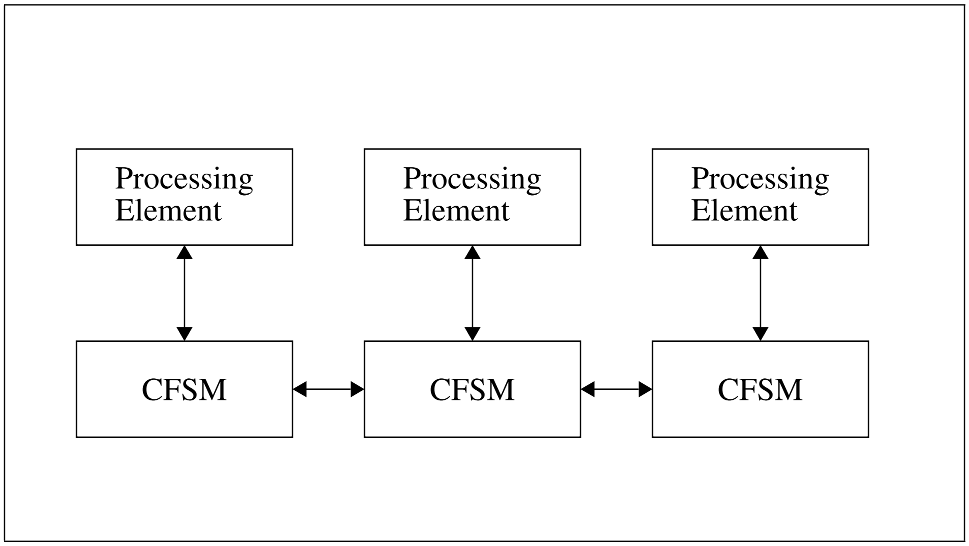

Figure 2.1 Block diagram of a three-node NuMesh system

A CFSM communicates with its neighbors by reading from and writing to *ports*; there is a single such port for each neighbor. A CFSM communicates with its processing element by reading from and writing to *processor registers*; there are several such registers (16 in the current implementation).

All CFSMs in the entire mesh share a common clock which can be independent of the clocks used for the processing elements. Each CFSM performs a fixed, periodic sequence of actions described by its *schedule*, which is part of its program.

2.4 Threads

A CFSM *thread* is a directive to perform a single-word transfer.[2] The source and destination of the transfer may be either a port or a processor register (or a *buffer register*; see section 2.10). Programming a CFSM consists of determining its threads and the schedule

2. Note that this terminology is somewhat unconventional. It comes from viewing each thread as being associated with a stream (see below), so that the serial execution of independent threads corresponds to the serial interleaving of independent streams.

which governs their execution.

Tadpole produces a different set of threads and a different schedule for each CFSM, but it constrains the period of every schedule to be the same. Thus the entire communication substrate has a single global period, which we denote by T henceforth.

2.5 Pipelines

Each CFSM has two independent *pipelines* which execute threads. The CFSM schedule specifies which thread enters each pipeline on each cycle. A thread's execution within a pipeline consists of several single cycle stages. For Tadpole, the relevant stages are the *read stage* and the *write stage*, which are consecutive. That is, while one thread is reading a value from its source, its immediate predecessor in the same pipeline is writing a value to its destination. This relative timing between the read and write stages determines how data transfers work (see section 2.6). Since there are two pipelines with independent read and write stages, each CFSM can perform up to two reads and two writes on each cycle. A read followed by a write is a *data transfer*; thus, each CFSM may perform up to two independent data transfers per cycle. (Put another way, each CFSM may transfer a word from each of two streams on every cycle; see below for a description of streams.)

Each thread in the CFSM is associated with a single pipeline, and may execute only in that pipeline.

2.6 Data Transfers and Streams

To program the transfer of a word between two nodes, we must schedule two threads on neighboring nodes to cooperate. Specifically, the sending node's thread must be in the write stage of its pipeline during the same cycle that the receiving node's thread is in the read stage of its pipeline. This means we must schedule the receiving thread one cycle later than the sending thread. (Recall that all nodes are clocked synchronously, which makes “one cycle later” a meaningful global concept.)

Thus, to transfer a word across multiple nodes, we must schedule cooperating threads on consecutive cycles of successive nodes. We also must choose a pipeline for each thread. We call a (node, cycle, pipeline) triple a *schedule slot*, since each such triple represents a potential scheduling of at most one thread. In other words, to transfer a word across multiple nodes, we must reserve a schedule slot on consecutive cycles of successive nodes.

The ultimate source and final destination of any word are processor registers.[3] Since the global schedule is periodic with period T, reserving a single schedule slot for a thread means executing that thread every T cycles. So by arranging to transfer a word across several nodes from one processor register to another, we create a stream between processing elements with bandwidth $\frac{1}{T}$ words/cycle. (For a method to create streams of higher bandwidth, see section 2.8.)

Note that every thread is associated with a single stream, and that every stream is independent of every other stream.

2.7 Flow Control

In general, the NuMesh communication substrate runs independently of the processing elements. Consequently, the precise timings of reads and writes to a processor register by a processing element are not necessarily predictable.

A processing element can detect when a processor register is full or empty. So it can always avoid writing to a stream and reading from a stream too quickly (i.e., faster than the bandwidth of the stream permits). However, if it is busy performing computation or is

3. Strictly speaking, this is not always true; the CFSM itself is programmed by writing to certain “magic” destinations. As mentioned in section 2.1, Tadpole does not use this facility in the schedules it generates.

otherwise slow, it may read from or write to a stream too slowly, and the communication substrate must be prepared for this eventuality.

If a processor writes to a stream too slowly, the thread at the head of the stream will attempt to read an empty processor register. The NuMesh transfers an *empty/full bit* with each word, guaranteeing that empty words will not be delivered to any destination processor register. So this case is not directly relevant to Tadpole.

If a processor reads from a stream too slowly, however, the communication substrate runs the risk of overwriting valid data in a destination processor register. This is prevented by the NuMesh flow control mechanism, which we will describe now.

In addition to a source and a destination, each thread specifies a *transfer type*. The transfer type is one of *blind*, *flow-controlled*, or *conditional*. Usually, Tadpole uses flow-controlled transfers (but see section 2.9).

Each thread has an associated *buffer register* capable of storing a single word of data. When a flow-controlled transfer fails, the word is stored in the thread's buffer register. Such a failure happens when the destination is not prepared to receive data (e.g., when it is a full processor register).

Eventually, the thread which encountered the failure is scheduled again. When the thread reaches the read stage of its pipeline with a full buffer register, it will read from that buffer register instead of from its designated source. If the source is a neighboring node, the thread on that node will then encounter a failure, placing a word into its own buffer register. (If the source is a processor register, it will simply not be emptied, a condition which the source processing element can detect.) The failure to write to a destination processor register will cause a stream to "back up" in the mesh. When the destination processor register is emptied, the stream will start moving again.

This means that a processing element can be "slow on the uptake" without causing data loss. The NuMesh flow control mechanism has the notable advantages that it is transparent to the CFSM programmer, and that it incurs no performance overhead [SHM96,Sh97].

2.8 High-Bandwidth Streams

The streams we have discussed so far are *minimum bandwidth*; that is, their bandwidth is $\frac{1}{T}$ words/cycle. To create a stream of higher bandwidth we must invoke the same thread more than once in the periodic schedule. If we invoke a single thread n times during one period, the associated stream will have a bandwidth of $\frac{n}{T}$ words/cycle.

Note that it will not work to provide several independent streams which happen to have the same source and destination, since the flow control mechanism would allow words to arrive in a different order than they were sent. For example, consider two streams s_0 and s_1 which have the same source and destination processor registers but are implemented by different sequences of threads. If s_0 were to encounter a failure because the destination processor register was full, it would "back up". If the destination processor register were emptied by the destination processor shortly thereafter, s_1 could then successfully deliver a word which was originally read from the source processor register after the word which failed.

So a high-bandwidth flow-controlled stream can be implemented only by a single sequence of threads invoked at multiple points in the periodic schedule.

2.9 Packet Sizes

The NuMesh does not allow the same thread to be scheduled twice in a row. Even if it did, a processing element would have difficulty timing a pair of writes to a single processor register to exactly match up to the two consecutive cycles on which some thread was scheduled.

However, some communication patterns do send data in short bursts, or *packets*. The NuMesh architecture supports a *conditional* style of thread transfer to handle this case. A conditional transfer inherits all of its flow-control data (i.e., success or failure) as well as its empty/full bit from the immediately preceding thread in the same pipeline. A packet can thus be sent by scheduling several different streams consecutively which are identical save for the processor registers which they access at the source and destination. All streams after the first in the sequence (which transfers the *head* of the packet) use conditional transfers instead of the usual flow-controlled. So the disposition of the head of a packet (i.e., whether it succeeds, “backs up”, or carries no data) determines the disposition of the following words in the packet.

The source processing element must take care to write the head of the packet last to insure that the inherited empty/full bits are safe. Similarly, the destination processing element must read the head of the packet last to insure that the inherited flow-control data are safe.

2.10 Delays

As described so far, streams require reserving threads on consecutive cycles of successive nodes along the path from source to destination. In the presence of high congestion, this requirement may be difficult to meet. To relax this requirement, the NuMesh allows any thread buffer register to be specified as a source or destination for any thread in the same pipeline. This allows one thread to transfer a word to another which is scheduled later. The earlier thread specifies its own buffer register as a destination, while the later thread specifies the earlier’s as a source. The result is to *delay* a stream at a node for up to $T - 1$ cycles.

2.11 Forks

The NuMesh CFSM supports a *fork* operation, which allows the data on a single stream to be sent in two directions. A fork requires cooperation between two threads scheduled on consecutive cycles in the same pipeline. If the second thread's source is the same as the first thread's destination, the CFSM will attempt to send the word **both** to that destination and to the second thread. This works only when the destination is a port; it does not work if the destination is a processor register or a thread buffer register.

This idiom (one thread's destination equal to following thread's source) cannot occur outside the context of a fork because it would represent an attempt to both send and receive a word on an external link at the same time. The NuMesh does not support full duplex transfers between nodes, so the meaning of this idiom is unambiguous.

Note that flow control on fork operations works correctly and transparently to the CFSM programmer. That is, the first thread "succeeds" only if both the destination port and the second thread are ready to accept a word; otherwise, both transfers "fail" (see section 2.7).

Through forks, the communication substrate provides a multicast facility, where a stream carries data from a single source to multiple destinations. A stream which forks n times can reach $n + 1$ destinations.

2.12 Summary of Hardware Limitations

This is a summary of the restrictions which the current NuMesh implementation imposes.

- The schedule size (and thus the period T) is at most 128.
- The number of threads per pipeline is at most 32.
- There are two pipelines.
- There are 16 processor registers.
- Full-duplex (i.e., simultaneous and bidirectional) transfers through a single port are prohibited.
- The same thread may not be scheduled on two consecutive cycles.
- The processor registers permit only one CFSM access (read or write) per cycle per pipeline.

2.13 Tadpole's Job

Tadpole's job is to take a number of *node descriptions* and *stream descriptions* as input, and to produce a *feasible schedule* as output. A node description consists of a name and a set of coordinates. A stream description consists of a name, a source node, one or more destination nodes, a *packet size*, and a *bandwidth*. A feasible schedule is a listing of the threads and the schedule for each CFSM, such that (1) each stream has the designated source, destination(s), and packet size, (2) each stream has at least the specified bandwidth, and (3) all of the limitations of the NuMesh hardware are respected.

Chapter 3

Off-line Routing

3.1 Definitions

A *task communication graph* is a directed graph whose vertices are tasks and whose edges are the required communication bandwidth between tasks.

Placement is the job of deciding which nodes of a physical network will house which tasks.

Routing is deciding which path or paths in a physical network will carry the traffic corresponding to each edge of the communication graph.

Scheduling is determining the exact cycle-by-cycle behavior of network traffic; i.e., which messages traverse which links at which times.

Given these definitions, we may describe Tadpole's job as follows. Tadpole takes a task communication graph for which placement has already been performed; the vertices are given by node definitions, and the edges are given by stream definitions. Tadpole then performs off-line routing and scheduling of the communication graph for the NuMesh architecture.

3.2 On-line Scheduling

Much of the existing research on the off-line routing problem assumes on-line scheduling. That is, it assumes the on-line system will schedule whatever network traffic happens to show up at run time, so that the off-line system's job is to place tasks and choose routes for inter-task communication to make the on-line system's job easier. More precisely, the goal is to find a placement and a set of routes which maximize the expected throughput of the on-line system. Off-line placement algorithms generally try to maximize spatial locality, while off-line routing algorithms generally try to reduce link congestion.[1] We will henceforth largely ignore the placement problem, since it does not concern Tadpole.

The off-line routing problem may be formulated as an attempt to find a set of routes which minimizes some *objective function*. The objective function is some measure of link congestion whose minimization corresponds to "optimal" expected on-line performance. Many algorithms for off-line routing use this formulation. The proposed objective function varies, and the proposed method for optimizing it ranges from fluid flow models [BS86] to simulated annealing [BM88] to ad hoc routing using shortest-path computations [KS92]. The general approach, however, is the same.

Tadpole's approach is similar to these in spirit, but with alterations to deal with off-line scheduling.

3.3 Off-line Scheduling

Most research on off-line scheduling has been restricted to particular hardware architectures. This is to be expected, since producing a cycle-by-cycle description of hardware behavior requires extensive knowledge of the hardware.

An architecture similar to the NuMesh is assumed by Shukla and Agrawal's *scheduled routing* technique [SA91], henceforth abbreviated SASR. Designed for video processing or other pipelined tasks, SASR takes an acyclic directed graph of tasks (a *task pipeline*) with edges representing messages. Each message has an associated *length*, and each task has an associated *execution time*. Tasks without parents in the graph are *input tasks*; those without children are *output tasks*. It is assumed that a task may not begin execution until its incoming messages have arrived, and that a task generates its output messages only after completing execution. The input tasks are assumed to be invoked periodically by some external signal whose period is long enough not to overwhelm the slowest element

1. "Link congestion" is defined as the sum of the bandwidths of all streams which use a link, where bandwidth is measured as a fraction of the total capacity of the link.

in the pipeline. With this constraint, all tasks may be invoked with the same period and the pipeline will function properly.

The SASR algorithm goes through several phases, first deriving time bounds for message injection and delivery, then routing messages in the network, then fragmenting messages across time intervals, then scheduling the fragments in time. The output is a detailed node switching schedule.

The SASR algorithm assumes that network latencies are negligible compared to message lengths, so it schedules a clear path from source to destination for the entire duration of each message transmission.

Although SASR was adapted to an earlier generation of the NuMesh [Mi93], it is not appropriate for the current generation. SASR messages are similar to NuMesh packets, except that NuMesh packet sizes (at most a handful of words) hardly make network latencies negligible by comparison. Allocating a clear path from source to destination for the duration of each packet transmission (sometimes as small as a single word) would be nonsensical.

SASR targets acyclic task pipelines which have large messages and a long period; one long enough, that is, to encompass the execution time of the slowest task, which is assumed to be a known quantity. The NuMesh provides relatively tiny packets but sends them repeatedly at a relatively enormous frequency, making it more suitable for providing high-bandwidth streams between tasks of arbitrary execution time.

3.4 Tadpole Formulation

One concept which we might borrow from SASR is the multiple-phase approach; that is, using separate phases for routing and for scheduling. The first phase could use any standard off-line routing approach to minimize some objective function, while the second phase could attempt to schedule network traffic along the chosen routes. Unfortunately, the minimization of some simple objective function during routing provides no guarantee that a feasible schedule will exist. Thus we would probably need to run both phases repeatedly with feedback (as in [SA91] and [Mi93]).

Fortunately, we can avoid the multiple-phase approach altogether by the following observation: Routing in space together with scheduling in time is equivalent to routing in space-time [Li89].

To visualize this concept, let us take a mesh of nodes and make an image of each node at every point in time (i.e. every clock cycle). Let these images of nodes form the vertices of a graph. From the image of each node at each time t, we draw a directed edge to every image of every neighbor of that node at time $t+1$. Each edge represents a link from one node to another between some cycle and the next; these may be thought of as potential single-cycle data transfers between nodes.[2] The edges are directed because time always flows forward.

2. In this chapter we ignore transfers which do not cross nodes (i.e. delays).

For example, consider the one-dimensional mesh in Figure 3.1.

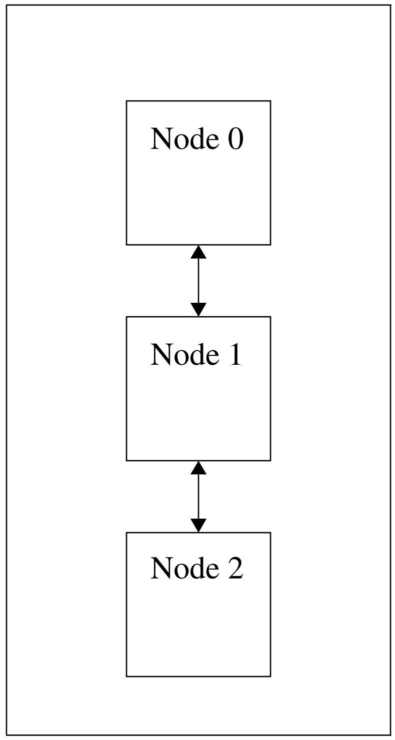

Figure 3.1 A one-dimensional mesh

The corresponding space-time graph is shown in Figure 3.2, where cycles 0, 1, and 2 have been drawn.

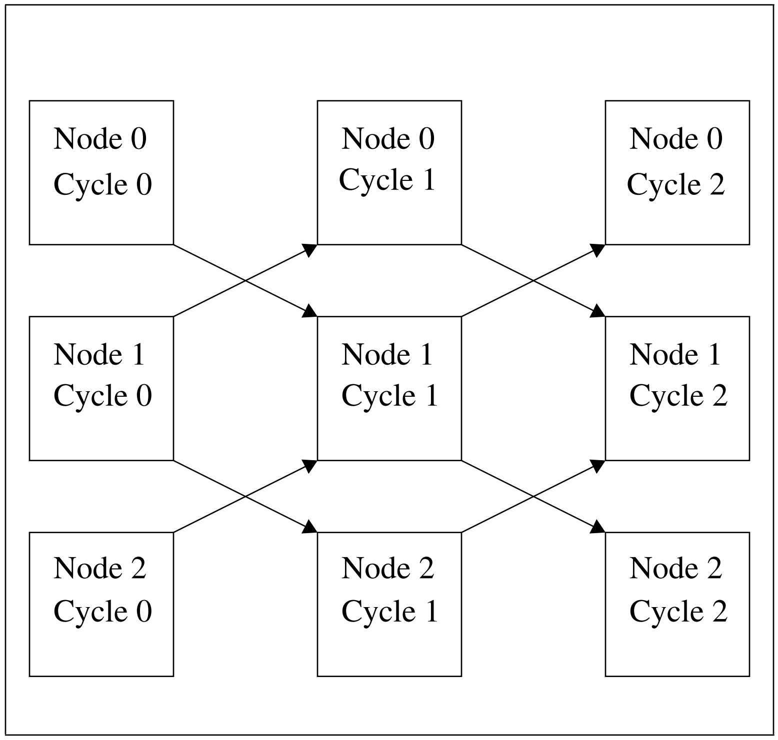

Figure 3.2 A two-dimensional space-time graph

By allocating resources in this space-time graph, we allocate them at particular places and times. For example, if we allocate a route which starts at Node 2, Cycle 0, goes through Node 1, Cycle 1, and ends at Node 0, Cycle 2, we are representing the action of Node 2 communicating to Node 1 on cycle 0, and then Node 1 communicating with Node 0 on the following cycle. Thus we have represented both the spatial routing of a word from Node 2 through Node 1 to Node 0, as well as the temporal scheduling of that route.

We can add additional routes depending on the capabilities of the underlying hardware. For example, if a word needs to be transferred from Node 0 to Node 2, we can do it from cycle 0 to cycle 2 provided Node 1 is capable of handling two transfers simultaneously.

On a space-time graph, algorithms for routing alone may be directly applied to the combined problem of routing and scheduling. Alternatively, information from the space-time graph may be incorporated into the objective function of a standard off-line routing algorithm, helping to insure that a feasible schedule exists when the objective function is minimized.

Note that this formulation is not always useful, since in general an application may require very many cycles to run, which would make the corresponding space-time graph intractably large. But if all activity is periodic, time itself is effectively bounded by the period, and the space-time graph is cyclic. If the period is sufficiently small, the space-time graph will also be small, and so may be explicitly constructed.

The present NuMesh implementation has an overall schedule size of at most 128 cycles, making this formulation applicable.

3.5 Multicommodity Flow

Looking again at the routing problem alone, we see that it is precisely an instance of the *integer multicommodity flow* problem. Briefly, the multicommodity flow problem takes as input a graph, a set of edge capacities, and a set of commodities; also, for each commodity, a collection of source vertices, sink vertices, and demands. The output is a *flow* for each commodity such that all demands are met and the maximum edge congestion is minimized.[3] A flow is a specification of the amount of each commodity which is to traverse each edge.

Integer multicommodity flow adds the restriction that a flow may not be split across different paths.

3. "Edge congestion" is defined as the total flow through an edge divided by its capacity. "Maximum edge congestion" is the highest congestion of any edge.

If we identify the input graph with the physical structure of a mesh, the commodities with streams, the demand for a commodity with the requested bandwidth of the stream, and the edge capacities with the maximum bandwidth allowed by the hardware, we see that off-line routing (with an objective function of maximum link congestion) is really just integer multicommodity flow.

Multicommodity flow is a well-studied problem [KP95,KP95a,KPP95].

The basic multicommodity flow problem may be solved in polynomial time with linear programming methods [KV86], but the integer case is NP-complete. However, there is a fast approximation algorithm due to Stein et. al. [St92]. Tadpole's heuristic approach is heavily based on this approximation algorithm, but augmented with information from the space-time graph.

3.6 Reduced Graphs, Shortest Paths, and Cost Functions

The Stein algorithm begins by routing commodities independently of each other, and then iteratively rerouting them in an attempt to decrease congestion. The rerouting phase uses minimum-cost path computations on a *reduced graph*; that is, one whose edge costs are related to the current edge congestion. A commodity is rerouted to use the "cheapest" path, where the cost of a path depends on how much congestion is present along it.[4]

Tadpole augments this cost function with information from the space-time graph as follows: the cost of a path in the reduced graph depends not only on the congestion of the edges along it, but also on whether a feasible scheduling of that path exists. Thus, routes which do not result in feasible schedules will not be considered, even if they would reduce overall edge congestion.

4. Obviously, the cost of an edge increases with its congestion. The specific cost function used by Stein was chosen to give provable bounds on the performance of the algorithm. The details are not important for Tadpole, whose cost function is a complex morass of heuristics.

In subsequent chapters, we will see a variety of ways in which we further augment the cost function.

Chapter 4

Ugly Details

Note: In this chapter, we consider only the case of minimum-bandwidth streams with a single destination.

4.1 Overview

As described in Chapter 3, Tadpole's general strategy is to build a space-time graph and to use it to augment the cost function of a multicommodity flow computation. The details of the NuMesh architecture, however, suggest both a slightly different formulation of the space-time graph and additional augmentation of the cost function.[1]

4.2 Nodelets

We have seen how to create a space-time graph which associates a vertex with each node at each possible time, and an edge with each potential communication between a pair of nodes. A more directly useful formulation for the NuMesh is to associate a vertex with each potential scheduling of a thread; that is, with each (cycle, pipeline) pair on each node.

We call the vertices in Tadpole's space-time graph *nodelets*. Each nodelet may be identified by its 3-tuple (N, t, k), where N is a node, t is a time (cycle number), and k is a pipeline. Since the schedule is periodic with period T, the nodelet (N, t, k) is the same as the nodelet $(N, t+T, k)$ for any N, t, and k.

The edges in Tadpole's space-time graph correspond to potential transfers of a word between threads. The edges leaving each nodelet (N, t, k) fall into two classes. The first class consists of edges to every nodelet of the form $(N', t+1, k')$, where N' is some neighbor of N and k' is some pipeline. The second class consists of edges to every nodelet of the

1. This is where we take our clean, abstract formulation and sully it somewhat to make it apply to real hardware. Such is engineering.

form $(N, t + \Delta t, k)$, where $0 < \Delta t < T$. The first class of edges corresponds to inter-node transfers, in exact analogy to the formulation of Chapter 3. The second class represents intra-node delays. Note that it is a restriction of the current NuMesh implementation that these intra-node delays may not cross pipelines.

4.3 Tadpole's Job is NP-hard

Given the Tadpole space-time graph, the task of scheduling and routing a stream from some node A to some other node B is easy to describe: Find a path in the graph from any nodelet of the form (A, t, k) to any other nodelet of the form (B, t', k'). Scheduling and routing a collection of streams thus requires finding a collection of paths which do not overlap.

A slight addition to the space-time graph reduces this problem into one which is well known. Augment the vertices of the graph with a collection of "virtual sources" and "virtual sinks" for each node. The "virtual sources" for node N have outgoing directed edges to every nodelet of the form (N, t, k) for all t and k. The "virtual sinks" for node N have incoming directed edges from every such nodelet. Scheduling and routing a stream from A to B now corresponds to finding a route from one of A's virtual sources to one of B's virtual sinks. As before, scheduling and routing a collection of streams consists of finding a collection of such routes which do not overlap. Now, however, we can associate a particular virtual source and virtual sink with each stream, which is what we need to complete our reduction.

This problem (i.e., given a collection of pairs of vertices, find a collection of pairwise vertex-disjoint paths which connect them) is known as the *disjoint connecting paths* problem, and is NP-complete in the general case [Ka72]. On graphs with certain special structure (e.g., those which are "densely embedded" [KT95]), polynomial approximation algorithms exist; on others (e.g., those with bounded tree-width [RS86]), polynomial-time (though impractical) exact algorithms exist.[2]

A polynomial-time algorithm is unlikely to exist for the Tadpole space-time graph, however. The disjoint connecting paths problem is also NP-complete in Euclidean graphs of as few as two dimensions [KL84]. An instance of the problem in a two-dimensional Euclidean graph may be reduced to a Tadpole problem by embedding the graph in the diamond lattice, letting the period T equal 1 (or, equivalently, giving all streams a bandwidth of 1.0 words/cycle), and requesting a stream from each source to each sink. So Tadpole's routing job is at least as hard as the disjoint connecting paths problem, and is therefore NP-hard.

4.4 Dealing with Hardware Limitations

So far, we have been ignoring the various hardware limitations of the current NuMesh implementation. We will rectify that now.

Fortunately, almost all hardware restrictions can be handled by further augmenting the cost function for the reduced graphs. Specifically, we can account for the consumption of resources other than mere bandwidth (such as threads per pipeline) by attaching a cost to the use of those resources. We can also insure that routes which require "impossible" behavior (such as bidirectional transfers) are never found by attaching an infinite cost to the edges associated with such events.

The following sections describe how the various hardware restrictions are handled by Tadpole. Most of these involve increasing the cost of an edge based on the resources it consumes.

2. As already mentioned, Tadpole is based on multicommodity flow methods. It might be possible, however, to adapt algorithms for disjoint connecting paths directly to off-line routing and scheduling with a space-time formulation. Whether this is so, and to what extent, is an interesting area for future research.

4.4.1 Schedule Size of 128 or less

The value of T is a fundamental parameter to most of Tadpole's code. So Tadpole requires either that the user supply it, or that the user give an upper bound on the desired value. If a bound is given, Tadpole merely runs the underlying algorithms for increasing values of T until a feasible schedule is found.[3]

4.4.2 Thirty-Two or Fewer Threads per Pipeline

Each pipeline supports a limited number of threads, currently 32. We incorporate this into the cost function by keeping track of how many threads have been allocated in each pipeline, increasing the cost of an edge to a nodelet as the number of threads in its pipeline increases. The details of this increase are ad hoc, and are determined by trial and error to give good empirical results. Of course, a super-saturated pipeline incurs infinite cost, since it cannot be used at all.

4.4.3 Only Two Pipelines

The current NuMesh architecture supports two pipelines; this is handled naturally by the Tadpole space-time graph by associating a separate nodelet with each pipeline, and allowing only a single transfer through each nodelet. Thus, edges to an already-used nodelet incur infinite cost.

4.4.4 Only Sixteen Processor Registers

This restriction is not relevant to Tadpole, since the number of processor registers used on a node depends entirely upon how many streams originate and terminate at that node. Thus, whether this particular restriction is met is a trivial property of the problem's stated requirements, and not a property of how Tadpole performs. It is the job of the person (or program) generating the requirements to insure this restriction is met. Of course, Tadpole will generate a fatal error if the number of allowed processor registers is exceeded.

3. A larger period usually makes it easier for Tadpole to find a feasible route. Unfortunately, this is not always the case; for example, a stream which requires a bandwidth of at least 0.5 words/cycle would actually produce more congestion with a period of 3 than with a period of 2. Thus a simple binary search for the optimal period will not work, so we resort to brute force.

4.4.5 Full-Duplex Transfers Prohibited

For each inter-node edge in the space-time graph, there is a corresponding edge which represents a potentially simultaneous transfer in the opposite direction. To prevent bidirectional transfers, the cost function checks that corresponding edge to see if it is already allocated to some stream. If so, the edge is given infinite cost.

Chapter 5

High-Bandwidth Streams

Note: In this chapter, we consider only the case of streams with a single destination.

5.1 Simplifying the Problem

By *high-bandwidth stream* we mean specifically a stream implemented by scheduling its threads more than once during each period (see section 2.8). Unfortunately, dealing with such streams is not nearly as easy as dealing with the various hardware restrictions.

What does such a stream, with bandwidth $\frac{n}{T}$, look like in the space-time graph? It is a collection of n minimum-bandwidth streams, each starting and ending at the same node, and taking the **same spatial route**. Put another way, the paths in the space-time graph are identical save for a set of offsets in time.

Finding a minimum-cost collection of n routes which are the same except for arbitrary temporal offsets appears to be a hard problem. The advantage to using minimum-cost path computations is that they can be done quickly, but this new problem does not fit directly into our minimum-cost path framework. Our solution is to simplify the problem artificially.

Instead of trying to find n routes from source to destination with unknown temporal relationships, we fix the temporal relationships before beginning the minimum-cost path computations. That is, for a stream with bandwidth $\frac{n}{T}$, we come up with n *stream offsets* between 0 and $T-1$ such that the offsets are roughly evenly spaced. We then mandate that the successive schedulings of the stream be at precisely those relative offsets, and no others.

For example, suppose we are attempting to route a stream of bandwidth $\frac{2}{T}$ in a mesh with a global period of 60 cycles. Then our stream offsets will be 0 and 30, meaning that whatever route we find in the space-time graph, we will allocate one just like it but 30 cycles later. Thus the two schedulings of the stream will always be exactly 30 cycles apart.[1]

This simplified problem can be solved within the minimum-cost path framework in the following way. When computing the cost of an edge in the space-time graph, we look at the edges which are offset from it by the stream offsets. The cost of the edge is then its own cost plus that of all of its offset images. In this way, we find a route for the "first" scheduling of the stream, and the stream offsets tell us the relative later schedulings of that same stream.

5.2 Self Interference

The simplification just described works, but it introduces a new problem, which we call *self interference*. Recall that our cost function incorporates congestion information for streams which have already been routed, but not for the stream for which the current minimum-cost path computation is being performed. For a stream of minimum bandwidth, this does not matter, since no minimum-cost path can ever cross itself (minimum-cost paths are always acyclic).

With high-bandwidth streams, however, a problem arises when we consider an edge corresponding to an intra-node delay. We must not allow a stream to be delayed by an amount which would cause it to *self-collide*; that is, we must not delay the stream by the difference between any pair of stream offsets. But the minimum-cost path algorithm has no way to know this, since the cost function described so far has knowledge only of already-routed streams. The solution is simple: We must give our cost function new knowledge by introducing a *self-interference cost* for any edge which corresponds to an

1. As mentioned, allowing for "jitter" in the stream offsets (e.g., 0-2 instead of 0, 29-31 instead of 30) seems to complicate the problem tremendously, so we avoid it.

intra-node delay. The cost of an edge which causes the stream to self-collide must be infinite.

Unfortunately, this modified cost function produces another (somewhat subtle) problem. Although our self-interference cost prevents collisions from simple intra-node delays, it does nothing to avoid routes which first leave a node and then come back to self-collide! Before we modified the cost function, it was impossible for this to happen, since for any two nodelets on the same node, the least costly route between them was always a simple intra-node delay. Now that we have added the self-interference cost, however, we have made certain simple delays very costly indeed, and our minimum-cost path computation could well find a path which self-collides. In other words, by handling self interference on a single node as a special case, we have exposed ourselves to the possibility of other kinds of self interference which were formerly impossible.

The structure of the minimum-cost path computation makes this new problem somewhat tricky to solve. Whether an inter-node edge corresponds to a self-collision depends on which nodes the path goes through to reach that edge. How do we compute this efficiently?

5.3 Path Histories

We will remember, for each node, all of the nodes which precede it along the shortest-path route. The order of the nodes is not important, so we can use a simple bitmask to store this information. These *path histories* are possible thanks to the structure of our particular minimum-cost path algorithm (Dijkstra's relaxation algorithm [CLR90]) which guarantees that nodes are visited in order of their distance from the source. Before relaxing a nodelet, Tadpole consults the path history, to insure that the same node is not visited twice along the same path. If so, we count the edge's cost as infinite.

Regrettably, this method carries a price: we now fail to find the actual shortest path under some circumstances. The correctness of Dijkstra's algorithm depends on the self-reducibility of the minimum-cost path problem. That is, any minimum-cost path from nodelet a to nodelet c, if it goes through nodelet b, always consists of a minimum-cost path from a to b joined with a minimum-cost path from b to c. Since the costs of our edges now depend on the path taken to them, this self-reducibility property no longer holds.

We can picture a specific failure as follows. Suppose we discover (through our path history), while relaxing along an inter-node edge, that it causes a self-collision and is therefore infinitely expensive. It is possible that some more expensive route to this point (which we never explored) would make the edge no longer cause a self-collision.

Unfortunately, there appears to be no way to efficiently compute shortest paths in the absence of the self-reducibility property. Thus, we choose to live with this failure. We take some consolation in observing that it occurs only for high-bandwidth streams.

This difficulty is, in some ways, similar to the first problem we encountered with high-bandwidth streams. We are using minimum-cost path computations because the structure of the problem allows an efficient solution, even though the number of potential paths between any pair of nodelets is exponentially large. We have now seen two aspects of dealing with high-bandwidth streams which violate that nice structure. We dealt with the first (unknown temporal offsets) by fixing the collection of stream offsets. We dealt with the second by introducing a path-dependent cost function. In both cases, we have restricted the search space artificially to achieve the necessary performance.

5.4 Packet Sizes

Dealing with packets of size greater than one is very similar to dealing with high-bandwidth streams. Once again, we need to find a multitude of routes which use the same spatial path but differ only in the time at which they are scheduled.[2] Now, however, we seek a collection of routes which occupy a collection of consecutive cycles. All routes except the first will be implemented with conditional moves (see section 2.7).

We incorporate this into our stream offsets. So now, for each stream, the offsets are used to represent both packet size requirements and bandwidth requirements. For example, a stream with packet size of 2 and bandwidth requirement of $\frac{4}{T}$ words/cycle in a schedule with $T = 60$ would use stream offsets of 0, 1, 30, and 31.

Note that there is one substantive difference between packet sizes and high-bandwidth streams: each word of a packet requires a separate thread, whereas each scheduling of a stream uses the same thread. This matters to the cost function only, since it needs to know how many threads are actually used by each stream in order to make sure we do not exceed the allowed number of threads per pipeline.

2. Strictly speaking, a single packet could use different spatial routes if they all took the same number of cycles to reach the destination. This observation was made too late to incorporate it into the current version of Tadpole; we henceforth ignore it for this thesis.

Chapter 6

Multicast

6.1 Introduction

So far, we have considered only streams with a single destination. One of the more interesting applications for off-line routing, though, is handling *multicast* streams. Multicast streams are those which have multiple destinations, and are supported in the NuMesh through fork operations (see section 2.11). Obviously, the decision of where to fork a stream can benefit greatly from global knowledge of network traffic, since the resources required to handle the stream are effectively doubled at all points following the fork. A stream which forks repeatedly (i.e. has more than two destinations) makes the decision even more crucial.

Recall that our approach for single-destination streams was to use minimum-cost path computations on a reduced graph in an effort to find a feasible scheduling and routing for all streams. Let us now consider the case of a stream with multiple destinations. In the space-time graph, routing such a stream means finding a tree from the source to all of the destinations. (There are some restrictions on the structure of the tree, since forks on the NuMesh have their own particular idiom; we will ignore this for now.)

Just as our algorithm for the single-destination case relied on finding minimum-cost paths, our algorithm for the multicast case will rely on finding minimum cost trees in the reduced graph. Finding such a tree is a well-known[1] problem.

6.2 The Steiner Tree Problem

Given an undirected graph, a set of edge costs, and a set of target vertices, the problem of finding the minimum-cost tree in the graph which spans the target vertices is known as

1. and very hard

the *Steiner Tree Problem*. It has long been known to be NP-complete [Ka72], and remains NP-complete even in Euclidean graphs of as few as two dimensions [HRW92].[2]

The Steiner Tree Problem in directed graphs is sometimes called the “Steiner Arborescence Problem”. For acyclic graphs, a branch-and-bound algorithm was proposed by Nastansky et. al. [NSS74]. Although it is conceivable that this approach could be modified to work with the Tadpole space-time graph, it would require exponential time (in general) to run.

In short, sufficiently little progress has been made on the Steiner Tree Problem in general graphs that heuristic approaches are commonly used in practice. Tadpole is no exception.

6.3 Tadpole Multicast Trees

A multicast stream, viewed as a tree in the space-time graph, has the following properties:

- It has a unique source nodelet, forming the root of the tree.
- It has some number of *intermediate nodelets*, having a single source and single destination, comprising vertices of the tree with a single child.
- It has some number of *branch points*, having a single source and exactly two destinations, comprising vertices of the tree with two children. Furthermore, one of these children must be the nodelet which is one cycle later in the same pipeline, and the other must be a nodelet on a different node. (The restrictions on the number and nature of the children are artifacts of the NuMesh hardware; see section 2.11.)
- It has some number of destination nodelets, forming the leaves of the tree.

6.4 Heuristic for Steiner Trees

Our heuristic for minimum-cost Steiner Trees is based on the following observation. A minimum-cost multicast tree has the property that if we take any two nodelets a and b in the tree which are connected solely by intermediate nodelets, that chain of intermediate nodelets must form a minimum-cost path from a to b in the reduced graph.

---

2. Interestingly, for graphs with bounded tree-width, a linear-time algorithm is known [KS90]. This algorithm is only of theoretical interest, however, since the constants involved are enormous.

Consequently, our heuristic is as follows. Begin with any multicast tree which connects the source to all the destinations. Choose any two nodelets which are connected solely by intermediate nodelets, disconnect them from each other, then reconnect them using a minimum-cost path. Repeat until the cost of the tree stops decreasing.[3]

We improve this simple algorithm as follows. Let the two chosen nodelets (connected solely by intermediate nodelets) be a and b. One of these is an ancestor of the other; say it is a. Then deleting the intermediate nodelets disconnects not only a from b, but also the two components of the tree in which a and b reside. Instead of finding a minimum-cost path from a to b, we find a minimum-cost path from b back to any nodelet in the component where a resides.

Of course, we can connect to that component only in certain ways; namely, those which respect the idiom of the NuMesh fork operation. We incorporate this information into the cost function by insuring that edges which correspond to "impossible" forks have infinite cost.

6.5 Our Heuristic and High Bandwidth

This heuristic for finding low-cost Steiner trees has the advantage that it is based on minimum-cost path computations which we have already analyzed and implemented code to perform. Consequently, we automatically handle the case of high-bandwidth streams and greater-than-one packet sizes, because we already dealt with them in the single-destination case (Chapter 5). Indeed, it is the nature of our heuristic that any improvement to the code for the single-destination case will immediately apply to the multicast case as well.

This property of code reuse together with its fair performance in practice are the main arguments in favor of our heuristic.

3. Or until we get tired of iterating. Hey, it's a heuristic.

Chapter 7

Results

7.1 Overview

This chapter presents examples of Tadpole's performance. Although Tadpole is intended for use with the three-dimensional four-neighbor diamond lattice topology, we restrict ourselves here to one- and two-dimensional Euclidean meshes for ease of presentation.

7.2 A Simple Line

Our first example is named *simple_line*. Its purpose is to illustrate Tadpole's scheduling capabilities. The network topology is the one-dimensional mesh with five nodes shown in Figure 7.1.

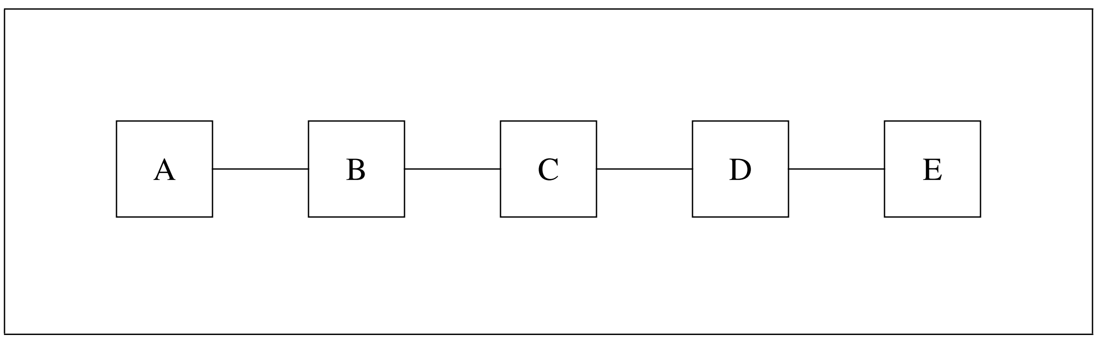

Figure 7.1 Topology of *simple_line* example

The input file for *simple_line* is given in Figure 7.2.[1] This defines the nodes A through

```
(node A (addr 0))
(node B (addr 1))
(node C (addr 2))
(node D (addr 3))
(node E (addr 4))

(stream S1 (src A) (dest E))
(stream S2 (src B) (dest E))
(stream S3 (src C) (dest E))
(stream S4 (src D) (dest E))
```

Figure 7.2 Input file for *simple_line* example

E with one-dimensional coordinates 0 through 4. It also defines four minimum-bandwidth streams named S1 through S4. Each stream's source is one of the nodes A, B, C, or D; all four streams have node E as destination.

When we give this input to Tadpole, we get back the schedule shown in Table 7.1.

Node	Cycle			
	0	1	2	3
A			(S1) preg->B	
B	(S2) preg->C			(S1) A->C
C	(S1) B->D	(S2) B->D	(S3) preg->D	
D	(S4) preg->E	(S1) C->E	(S2) C->E	(S3) D->E
E	(S3) D->preg	(S4) D->preg	(S1) D->preg	(S2) D->preg

Table 7.1 Generated schedule for *simple_line* example

Each entry in the table represents a thread, with ports indicated by the node with which they communicate. Processor registers are denoted simply by "preg" (since distinguishing processor registers is unnecessary when illustrating Tadpole's performance). For example, stream S1 begins with a read from a processor register on node A during cycle 2; proceeds

1. The syntax for Tadpole input files is described in the Appendix.

through nodes B, C, and D on cycles 3, 0, and 1; and arrives at a destination processor register on node E at cycle 2 again.

For this example, we have restricted ourselves to a single pipeline and have set the global period to four cycles. Even with these restrictions, which require that the link to node E be used at 100% capacity, Tadpole has scheduled the read of each source word so that it will reach its destination without being delayed at any node.

7.3 Routing Around Congestion

Our next example is called *around*. It demonstrates Tadpole's routing capabilities. The topology for this example is the two-dimensional structure shown in Figure 7.3.

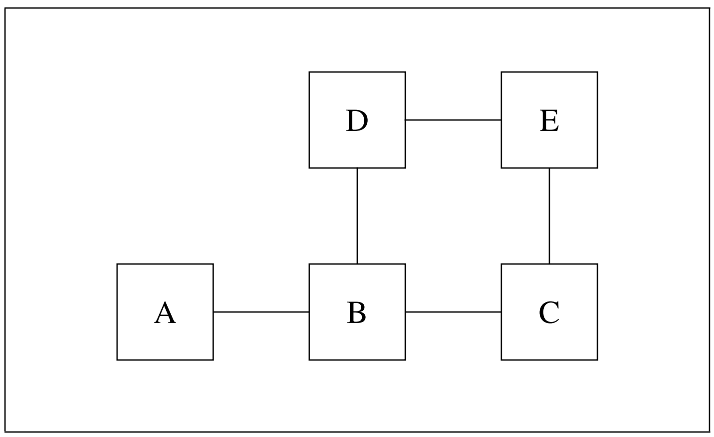

Figure 7.3 Topology of *around* example

The input file is shown in Figure 7.4. Streams S1 and S2 go to node C from nodes A and

```
(node A (addr 0 0))
(node B (addr 1 0))
(node C (addr 2 0))
(node D (addr 1 1))
(node E (addr 2 1))

(stream S1 (src A) (dest C))
(stream S2 (src B) (dest C))
(stream S3 (src A) (dest E))
```

Figure 7.4 Input file for *around* example

B, respectively. Stream S3 goes from node A to node E. We will set the global period to three cycles, meaning each of these streams consumes $\frac{1}{3}$ of the bandwidth capacity of the mesh.

Tadpole's output for this case is shown in Table 7.2, where again we have restricted

Node	Cycle		
	0	1	2
A		(S1) preg->B	(S3) preg->B
B	(S3) A->D	(S2) preg->C	(S1) A->C
C	(S1) B->preg		(S2) B->preg
D		(S3) B->E	
E			(S3) D->preg

Table 7.2 Generated schedule for *around* example

ourselves to one pipeline. Note the routing of stream S3 through node D to avoid the congestion on the link between node B and node C. Also note how dimensional routing (with the horizontal dimension ordered before the vertical) would have saturated that link.

7.4 Multicast

Our multicast example is named *fork*. Its network topology is a 100-node two-dimensional complete Euclidean mesh. The input file is shown in Figure 7.5 (abbreviated for clarity). It

```
(node x0y0 (addr 0 0))
(node x0y1 (addr 0 1))
(node x0y2 (addr 0 2))
<...>
(node x9y8 (addr 9 8))
(node x9y9 (addr 9 9))

(stream Sbig  (src x2y7) (dest x9y0) (size 2) (bw 1.0))
(stream Sfork (src x0y0) (dest x9y8 x8y9))
```

Figure 7.5 Input file for *fork* example defines nodes named x0y0 through x9y9, with Euclidean coordinates (0,0) through (9,9). It defines one stream named Sbig with source x2y7, destination x9y0, bandwidth 1.0 (the maximum), and packet size two words.[2] It also defines one minimum-bandwidth multicast stream named Sfork with source x0y0 and destinations x9y8 and x8y9.

Tadpole's output for this example is shown graphically in Figure 7.6. The heavy gray

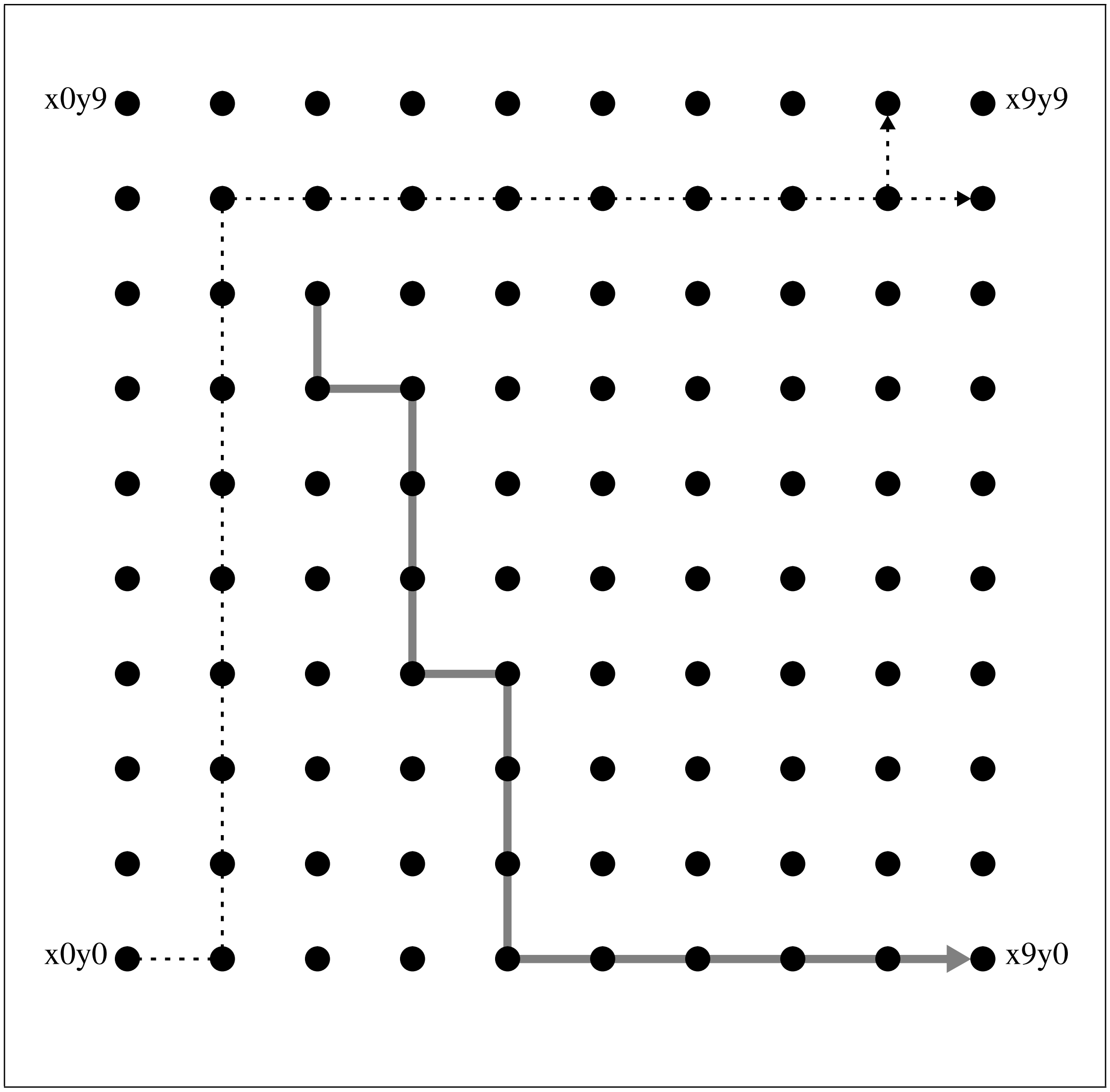

Figure 7.6 Tadpole's routes for *fork* example

line shows the route of Sbig, while the thin dotted line shows the route of Sfork. Tadpole

2. Sbig uses a packet size of 2 because the same thread may not be scheduled twice in a row (meaning a stream with packet size 1 may be scheduled at most every other cycle).

chose to place the actual fork on node x8y8 and to route around the Sbig stream.[3] Although it cannot be seen from the figure, the fork on node x8y8 occurs on consecutive cycles within a single pipeline, as required by the hardware.

7.5 A Silly Example

This example is called *maze*. Its network topology is a 49-node 7x7 complete Euclidean mesh. It uses 12 maximum-bandwidth streams and one minimum-bandwidth. It is probably best understood by examining the output, whose routes are shown in Figure 7.7.

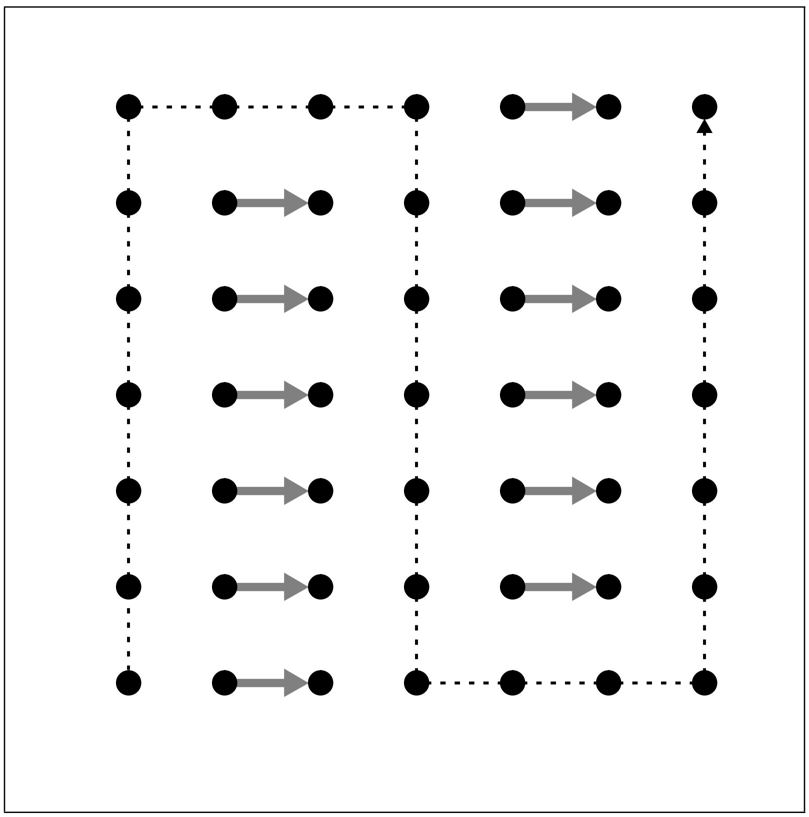

Figure 7.7 Tadpole's routes for *maze* example

The high-bandwidth streams are denoted by thick gray lines; the minimum-bandwidth by a thin dotted line. By totally saturating certain links, we ensure that only one route from

3. Routing around the high-bandwidth stream was not strictly necessary, since the second pipeline on node x4y1 (for example) could have been used to let the streams cross. Tadpole chose the route it did because letting the streams cross would not improve the latency, and using both pipelines is slightly more "costly" than using one (see section 4.4).

the lower-left to the upper-right is feasible.[4] With this trick, we can use Tadpole to solve arbitrary two-dimensional Euclidean mazes.

7.6 Toroidal Nearest-Neighbor

This example is named *near8*. Its network topology is a 100-node, 10x10 complete Euclidean mesh. Each node is the source of a minimum-bandwidth multicast stream whose destinations are the eight surrounding nodes. Such a communication pattern is characteristic of some cellular automaton applications.

For this example, nodes on opposite sides of the mesh are considered adjacent. That is, the logical topology is a torus even though the physical topology is a simple grid. To handle this, Tadpole must allocate routes all the way across the grid for nodes on opposite sides.

Tadpole successfully routes and schedules *near8* for periods as low as 17 cycles, although the results are difficult to display graphically.

7.7 Conclusion

Tadpole performs well on these simple benchmarks, producing working results in a reasonable amount of time. Even *near8*, a fairly large benchmark, takes less than 45 seconds of CPU time to run on a typical workstation.

Unfortunately, Tadpole's heuristic approach makes it nearly impossible to make provable statements about the quality of its results or its total running time. The basic operation of finding a minimum-cost path using Dijkstra's algorithm with a binary heap is $O(E \log V)$ [CLR90], which for the Tadpole space-time graph for a n-node mesh and period T becomes $O(T^2 n \log Tn)$.[5] However, the number of minimum-cost paths which we compute

4. Some would call this "bad placement".
5. This is obvious since there are Tn nodelets and T edges per nodelet corresponding to delays. Purists might suggest the use of Fibonacci heaps (again, [CLR90]), but that would only help if T were large, which it isn't.

depends on how much re-routing we have to do to find a feasible schedule, and there is no obvious way to bound this. Our heuristic approach to finding Steiner trees (section 6.4) further compounds the problem.

However, Tadpole's running time and results are good enough in practice that it is being used as part of a more complex automatic compilation system for the NuMesh [Me97].

Appendix

Configuration File Syntax

A.1 Overview

Tadpole's configuration file uses a Lisp-like s-expression syntax.

A.2 *node* Directive

The *node* directive defines a node. Figure A.1 summarizes its syntax.

```
(node name (addr x y z t))
```

Figure A.1 Syntax of *node* directive

This defines a node named *name* with up to four coordinates *x*, *y*, *z*, and *t*. If fewer than four coordinates are specified, those omitted default to zero. Thus a three-dimensional mesh has *t* equal to zero; a two-dimensional mesh has *z* and *t* equal to zero; etc. Two nodes are adjacent if the *Manhattan distance* between them (the sum of the absolute values of the differences of corresponding coordinates) is one.

A.3 *stream* Directive

The *stream* directive instructs Tadpole to allocate a stream between nodes. Figure A.2 summarizes its syntax.

```
(stream name (src node0) (dest node1 node2 ...)
  (bw bandwidth) (size size))
```

Figure A.2 Syntax of *stream* directive

This defines a stream named *name* with source *node0* and destinations *node1*, *node2*, and so on. *node0*, *node1*, etc. must be names of nodes defined earlier in the configuration file. The stream will have a packet size of *size* and a bandwidth not less than *bandwidth* words/cycle. *node2* and higher are optional (of course), size defaults to 1, and *bandwidth* defaults to the minimum ($\frac{1}{T}$).

The value of T is determined by command-line arguments to Tadpole.

Bibliography

[BM88] S. Wayne Bollinger and Scott F. Midkiff. Processor and link assignment in multicomputers using simulated annealing. *International Conference on Parallel Processing*, pages 1-7, 1988.

[BS86] Ronald P. Bianchini, Jr. and John Paul Shen. Automated compilation of interprocessor communication for multiple processor systems. In *IEEE International Conference on Computer Design*, October 1986.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*, pages 527-532. Cambridge, MA; New York: MIT Press; McGraw-Hill, 1990.

[DS87] W. J. Dally and C. L. Seitz. Deadlock-free message routing in multiprocessor interconnection networks. *IEEE Transactions on Computers*, Vol. C-36, No. 5, pages 547-553, May 1987.

[HRW92] Frank K. Hwang, Dana S. Richards, and Pawel Winter. *The Steiner Tree Problem*. Amsterdam; New York: North-Holland, 1992.

[Ka72] R. M. Karp. Reducibility among combinatorial problems. *Complexity of Computer Computations*, R. E. Miller and J.W. Thatcher, Eds., New York: Plenum Press, pages 85-103, 1972.

[KP95] David Karger and Serge Plotkin. Adding multiple cost constraints to combinatorial optimization problems, with applications to multicommodity flows. In *Proc. 18th Annual ACM Symposium on the Theory of Computing*, May 1995.

[KP95a] Anil Kamath and Omri Palmon. Improved interior point algorithms for exact and approximate solution of multicommodity flow problems. In *Proc. 6th ACM-SIAM Symposium on Discrete Algorithms*, 1995.

[KPP95] Anil Kamath, Omri Palmon and Serge Plotkin. Fast approximation algorithm for minimum cost multicommodity flow. CS-TN-95-19, Dept. of Computer Science, Stanford University, Stanford, CA 94305, 1995.

[KL84] M. E. Kramer and J. van Leewen. The complexity of wire routing and finding the minimum area layouts for arbitrary VLSI circuits. Advances in Computing Research 2: VLSI Theory, F. P. Preparata, Ed., London: JAI Press, pages 129-146, 1984.

[KS90] Ephraim Korach and Nir Solel. Linear time algorithm for minimum weight Steiner tree in graphs with bounded tree-width. Technical Report #632, Dept. of Computer Science, Technion - Israel Institute of Technology, Haifa, Israel, June 1990.

[KS92] Dilip D. Kandlur and Kang G. Shin. Traffic routing for multicomputer networks with virtual cut-through capability. *IEEE Transactions on Computers*, 41(10):1257-1270, October 1992.

[KT95] Jon Kleinberg and Éva Tardos. Disjoint paths in densely embedded graphs. In *36th Annual Symposium on Foundations of Computer Science*, pages 52-61, Milwaukee, Wisconsin, October 1995.

[KV86] S. Kapoor and P. M. Vaidya. Fast algorithms for convex quadratic programming and multicommodity flows. In *Proceedings of the 18th Annual ACM Symposium on Theory of Computing*, pages 147-159, 1986.

[Li89] Bjorn Lisper. *Synthesizing Synchronous Systems by Static Scheduling in Space-time*. New York: Springer-Verlag, 1989.

[Me97] Chris Metcalf. A comparative evaluation of online and offline routing in multiprocessors. Ph.D. thesis, MIT, 545 Technology Square, Cambridge, MA 02139, 1997.

[Mi93] Milan Singh Minsky. Scheduled routing for the NuMesh. Master's thesis, MIT, 545 Technology Square, Cambridge, MA 02139, 1993.

[NSS74] L. Nastansky, S. M. Selkow, and N.F. Stewart. Cost-minimal trees in directed acyclic graphs. *Zeitschrift für Operations Research*, Band 18, pages 59-67, 1972.

[RS86] Neil Robertson and P. D. Seymour. Algorithmic aspects of tree-width. *Journal of Algorithms*, 7(3):309-322, September 1986.

[SA91] Shridhar B. Shukla and Dharma P. Agrawal. Scheduling pipelined communication in distributed memory multiprocessors for real-time applications. In Annual International Symposium on Computer Architecture, pages 222-231, 1991.

[Sh97] David Shoemaker. An optimized hardware architecture and communication protocol for scheduled communication. Ph.D. thesis, MIT, 545 Technology Square, Cambridge, MA 02139, 1997.

[SHM96] David Shoemaker, Frank Honoré, Chris Metcalf, and Steve Ward. NuMesh: An architecture optimized for scheduled communication. *The Journal of Supercomputing*, 10:285-302, 1996.

[St92] Clifford Stein. Approximation algorithms for multicommodity flow and shop scheduling problems. Ph.D. Thesis; also Tech Report MIT/LCS/TR-550, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, 1992.

[Ta81] Andrew S. Tanenbaum. *Computer Networks*. Englewood Cliffs, N.J.: Prentice-Hall, 1981.