Managing Scheduled Routing With A High-Level Communications Language

by

Christopher D. Metcalf

B.S., M.S., Computer Science
Yale University, 1988

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

SEPTEMBER 1997

Author . . . .
Department of Electrical Engineering and Computer Science
August 19, 1997

Certified by . . . .
Stephen A. Ward
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Chairman, Departmental Committee on Graduate Students

Managing Scheduled Routing With A High-Level Communications Language

by

Christopher D. Metcalf

Submitted to the Department of Electrical Engineering and Computer Science
on August 19, 1997, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

One of the most critical issues in multiprocessors today is managing the communications among processors. This thesis addresses the programming of systems (such as the NuMesh) that handle routing via high-speed reprogrammable scheduled routers on each node. A portable 'communications language', COP, used to express communications requirements, is presented.

Using scheduled routers to control data motion through the network has two advantages. First, by performing offline path scheduling for messages based on information extracted from the application, latency and congestion can be improved. Second, by removing the need for online decision making, such routers can run at extremely high speeds, further decreasing latencies and increasing performance.

Simple communication patterns known at compile time are relatively easy to schedule. However, data-dependent communications present more of a challenge. The compiler makes tradeoffs among different compilation techniques based on information in the communication program, including communication type and predicted traffic, as well as knowledge of the system size and layout. Further, the compiler chooses dynamically how to break up an application's communications into sequential phases, using communication relationships expressed in the input language. The compiler generates code that efficiently handles changing phases and ensuring data integrity while doing so.

This thesis shows that certain classes of applications, such as those that suffer from congestion with dynamic routers, show a significant decrease in run-time routing cycle count when using scheduled routing. Furthermore, reprogrammable scheduled routers are shown to provide a relatively general solution to application communication needs; applications with uncongested or data-dependent traffic are found to take approximately the same number of routing cycles, yielding a decrease in overall runtime given the potentially high cycle-speed of scheduled routers.

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

Acknowledgments

My years at the Laboratory for Computer Science have given me a remarkable opportunity to work with and learn from a range of bright and interesting people. This thesis draws upon that background not just for technical issues, but as a standard to live up to.

Steve Ward, my thesis supervisor, provided an important part of the support for this thesis. As well as starting the NuMesh project, he encouraged me to explore issues that were a little more fundamental and a little less mainstream, and both directed and participated in the chaotic and productive group meetings out of which came many of the most interesting results of the NuMesh project.

Thanks are also due to the other two members of my thesis committee, Gill Pratt and Frans Kaashoek. Their insistence on keeping the big picture visible, making the experimental framework of the results clear, and being precise in drawing conclusions, helped to make the thesis as readable as it is.

The other graduate students in the NuMesh group, along with the Alewife and CVA members of the overall Computer Architecture Group, gave me much of the day-to-day social and intellectual life of the lab. In particular, I would like to thank Dave Shoemaker for great collaboration and for refusing to allow me to ignore hardware, Pat LoPresti for helping me sound out ideas on languages and scheduled routing as well as putting up with continuous Tadpole bug reports, and Anne McCarthy for her competence, warmth, and supportive fire-fighting. My officemate Kirk Johnson was a source of many an interesting conversation, late night systems hacking, and friendly support. Thanks also to Frank Honoré, Russ Tessier, John Nguyen, and John Pezaris for many interesting times.

Thanks to the people who volunteered their time to read through drafts of this thesis and volunteer suggestions, Michael Littman and Joe Morrison, and Dave Shoemaker.

My parents provided me with encouragement throughout my thesis, giving me perspective from the other side of the Ph.D. fence and letting me know I was in their thoughts as I passed the various milestones of the thesis path.

Finally, and most importantly, thanks to my wife, Elaine, without whom I'm not sure I would have made it through—and without whom I certainly wouldn't have started at MIT when I did! Her support and encouragement were a large part of what kept me going; her advice helped keep me focussed on making progress week by week; and her shared happiness at every milestone passed was a joy. Just as important, even at times when my thesis didn't seem to be going the direction I wanted it to, her presence in my life provided a fundamental sense of balance and happiness.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

One of the most critical issues in multiprocessors today is managing the communications among processors. Traditionally, communication in multiprocessors was a major bottleneck, as was the case for the early Intel multiprocessors [37], for example. More modern machines (*e.g.* recent Cray machines [36]) have much higher-speed interfaces and networks; furthermore, recent work in communication techniques is helping to lower or mask latencies further (*e.g.* active messages [66] and rapid context switching [1, 61]). Communications, however, can still be a serious performance constraint.

Two approaches to reducing latency and increasing bandwidth are possible. One is to reduce cycle times in the network, thus moving data through the network more quickly, and directly improving both latency and bandwidth. The other is to make more intelligent use of the existing bandwidth in the network, such that the useful bandwidth can be increased and message latency better controlled (typically by providing guarantees of maximum latency). Both approaches can be applied at once by using a simple network processing element (with correspondingly fast switching times), running a reprogrammable scheduled router—essentially a simple finite state machine—to control data motion. In this thesis, this technique is referred to as *reprogrammable scheduled routing*; this class of routers is represented by the NuMesh router [68, 58, 59, 57]. Essentially, routing decisions are made offline (when the application is compiled), and at run time the routers simply execute repetitive schedules that handle all data transfers.

Pre-scheduling communications through the network has several clear advantages. Primarily, it allows for the routing scheduler to choose message paths to maximize specific user criteria (such as throughput and latency). Without some form of pre-scheduling, decisions can only be made based on local criteria; furthermore, routing hardware can only make decisions optimized for what the hardware designer thought was important. More importantly, without pre-scheduling, a router has no ability to look ahead to see what messages will be arriving shortly; it is a strictly "online" solution. Routing schedulers can examine the entire application's communication structure before generating schedules for the routers, and therefore represent a more general "offline" solution. Additionally, the more complex hardware needed for traditional data-dependent decisions can not achieve the short cycle time that is possible in principle with a highly-pipelined finite state machine. However, there are several limitations to the picture presented thus far.

Firstly, it is usually true that an application's communications are not completely deducible at compile time. For example, an application may have a broadcast that is performed along all the columns of a matrix, but the exact row may be unknown. At run time, all the processors may know who is broadcasting—or perhaps only the broadcasting processor will know. Using the communications language presented in this thesis, COP[1], an application can specify everything that is known about communication patterns at compile time: compile-time constant components, restrictions on what dimensions are used or what nodes are involved, whether all the nodes have full information on the needed pattern or not, and so forth. Furthermore, each communication *operator* (*e.g.*, broadcast or parallel prefix) can be tagged with information that the compiler can use to optimize the implementation: message size, number of messages expected to be sent, and maximum desired bandwidth, for example.

Given the information presented by the language, the compiler has a wide range of implementation options for data-dependent communication operations. If only a few router schedules are needed to handle the possible operand values for the operator, the compiler may include them all, along with run-time selection code. Or, if including multiple schedules is infeasible, it may make sense to simply allocate all the possible paths statically, and use only the relevant ones at run time. For more dynamic, but bandwidth-intensive communications, the compiler may choose to include code to compute the desired schedules at run time. Or, in the most dynamic situations, the compiler may fall back on generating code to perform online routing, using nearest-neighbor connections to forward messages through the mesh.

Secondly, it may be the case that an application is composed exclusively of communications whose source and destination can be uniquely deduced at compile time, but that there are simply too many such communications streams. A typical application will often move through a range of communication and computation *phases*; each phase will have its own static communications graph, and merging all the schedules into one global application schedule may result in greatly decreased bandwidth available on each stream. This thesis examines how to specify, implement, and optimize the use of phases at the router level, by coordinated reprogramming of the routers during the execution of the program.

Finally, it is important to support the notion of multiple threads of control in the program. An application may have different components that are running unrelated code at times during its execution. It is important to support this in the communication language, since otherwise the ability of the application to change phases to match changing communication requirements would be greatly constrained. For example, an application may want to divide the mesh into subcomponents, each of which runs independent loops several phases in length. Without a notion of 'disjoint components' it would be necessary to either synchronize all the phase changes, or combine all the communications into a single phase; either way would seriously limit the amount of bandwidth that could be scheduled for each communication.

1.1 Contributions of the Research

The primary contribution of this research is demonstrating the effectiveness of a scheduled router on a range of dynamic and multi-phase applications. The results demonstrate its ability to generate better cycle counts for some classes of applications, and similar cycle counts

[1] The Communication OPerators language

(and thus presumably faster wall-clock time) for other, arbitrary applications. Similarly, the provided results demonstrate that reprogrammable scheduled routing is a relatively general solution for routing. A secondary contribution is the design of a language that expresses the communication requirements of an application, in a way that simultaneously separates communication from computation, yet still provides sufficient semantic information to guide a routing compiler in producing efficient scheduled routing code.

Within this domain, this thesis contributes a variety of specific techniques for reprogrammable scheduled routers that allow for specific functionality or that improve performance.

- A wide variety of approaches for implementing a given communications operator, particularly when the operator involves data-dependent communications.
- An implementation of a virtual online router that allows for arbitrary communications to be issued by the application when absolutely necessary.
- Techniques for managing 'continuing operators' extending across multiple phases, without losing data from the continuing operators during phases changes, and without compromising other operators at phase-switch time.
- Algorithms for determining how to switch a node safely from running one phase to running another, without losing data in the current phase.
- Algorithms for determining what times are safe to communicate between nodes, given the possible presence of neighboring nodes that have not changed phases synchronously.
- A model for treating the router as a *cache* of currently-active scheduling information, handling such issues as optional phases (or phases selected at run time), as well as managing cache placement for groups of resources simultaneously.
- An algorithm for placing router-updating code in the application that allows for nodes uninvolved in a subset to nonetheless support communications for that subset.
- An approach for partitioning a mesh such that different threads of control can run independently, based only on which nodes are included in each communication operator and whether groups of operators are specified as running sequentially or in parallel.
- An algorithm for choosing how to partition a program optimally into communication phases, and simultaneously how to implement each communication operator most effectively.

Using these techniques, two backends for the COP compiler have been implemented, one for traditional dynamic routing and one for a reprogrammable scheduled router (NuMesh). Comparisons of the two backends are presented, examining the performance of the communications generated in each case.

This research does not include two critical components of scheduled routing. The first component is the process used to extract a communication-language program from a high-level language such as Fortran. The second component is the process used to schedule a single phase of static streams on a mesh. The research falls, in a sense, between the two levels, interfacing a high-level language to a low-level scheduling router.

1.2 Benefits of Scheduled Routing

Before proceeding further with the material, let us look more closely at scheduled routing itself, which can be dramatically superior to conventional routing in a number of different ways. Until now, it has been fairly difficult to access this functionality for all but a limited selection of applications.

1.2.1 Cycle Speed

One of the most significant advantages of scheduled routing is that it allows all routing decisions to be made offline, and thus out of the critical path at run time. Consider a typical routing decision made by a dynamic router: it must read a word from another node, examine the header of the node for its destination (and possibly adaptive-routing information), and choose a neighbor node (or virtual channel) to send it to. These decisions, which are in the critical path for message latency, take time. Pipelining techniques can be applied to this decision-making to reduce cycle times and increase bandwidth, but the latency for message transfer remains the same.

Furthermore, as is discussed in Section 1.4, a scheduled router can avoid all round-trip handshakes when doing flow-controlled data transfers. This allows scheduled routing to perform flow control within a single inter-node transfer time. Larger buffers and asynchronous flow-control signals can be used to avoid round-trip handshakes for dynamic routers as well, but this is costly in terms of pin resources and/or buffer management.

In [2], a comparison of dynamic routers is given, breaking down all the components of the cost. A simple deterministic router's cycle time is quoted at 9.8 ns, with a simple planar-adaptive router taking 11.4 ns. In [56], Shoemaker uses these component estimates to derive an estimate for the cycle time of a scheduled router using the same technology. He finds that the NuMesh scheduled router's cycle time is constrained only by the time for a single cross-node transfer, 4.9 ns. This is 50% of the deterministic router's clock period, and 43% of the simple adaptive router's. Improved signaling techniques or pipelining the cross-node transfer itself would decrease this cycle time even further with only minimal effect on the techniques used for scheduled routing.

1.2.2 Hotspot Avoidance

One important advantage of scheduled routing is its ability to make full use of a network's internal bandwidth when presented with data patterns that normally cause hotspotting in the network. For example, a bit-reversal routing can be efficiently distributed across the bisection by precomputing the paths that all the data will follow. Less-structured communication can also be routed cleverly so as to avoid problems from contention in the network. In general, when using all the scheduled streams at full bandwidth, the only time that communications will run slower than the actual path distance between nodes is when the network bandwidth needed is inadequate, not just because of contention artifacts.

Figure 1-1 shows a simple communication pattern that demonstrates this problem. Each node transfers data to the node whose number is the bit-reversal of its own number; thus node 2 (binary 0010) transfers its data to node 4 (binary 0100). When transferring data using standard

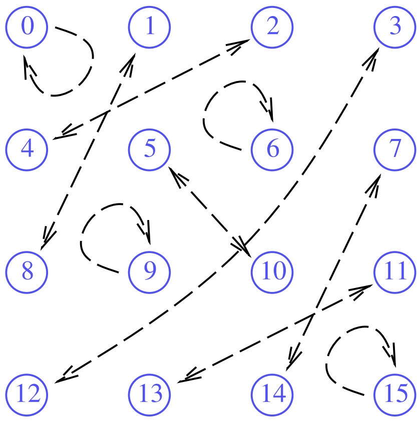


Figure 1-1: Bitreversal on a small mesh

dynamic-routing schemes, messages will tend to saturate particular links in the mesh. For example, using e-cube routing (where messages travel first in the x dimension, then the y), all the data from the top and bottom rows will flow using along the links on the edges of the mesh, using none of the central links. As the mesh grows (particularly for 3D meshes), the problem gets worse. With scheduled routing, good paths can be found for all the communications; Figure 1-2 shows possible scheduled paths for the 4×4 example.

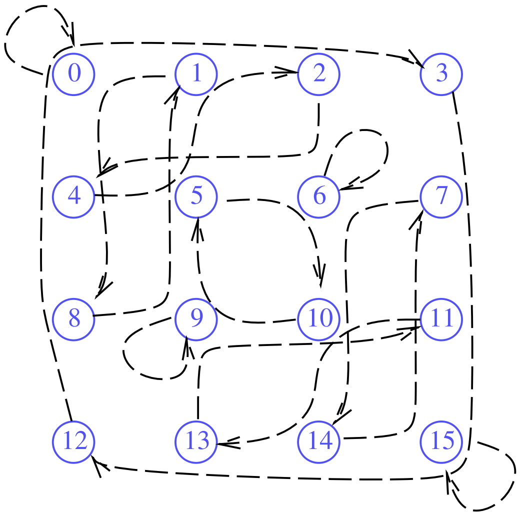


Figure 1-2: Scheduled paths for bitreversal on a small mesh

Figure 1-3 shows the results of a simple benchmark for bitreversal using different routing techniques. 512 data words are transferred between each pair of nodes with bitreversed addresses. Regular e-cube dynamic routing is shown, as is a simple adaptive routing technique from [21], and a scheduled-routing algorithm. The scheduled-routing algorithm schedules a route for all the streams at compile time by rerouting streams one at a time until it finds a good path, using only per-link bandwidths to determine legal routes. The dynamic routing cycle counts are gathered by simulation, and the scheduled routing cycle counts by scheduling a path and deriving the number of cycles a message takes to flow across the paths given the allocated bandwidths.

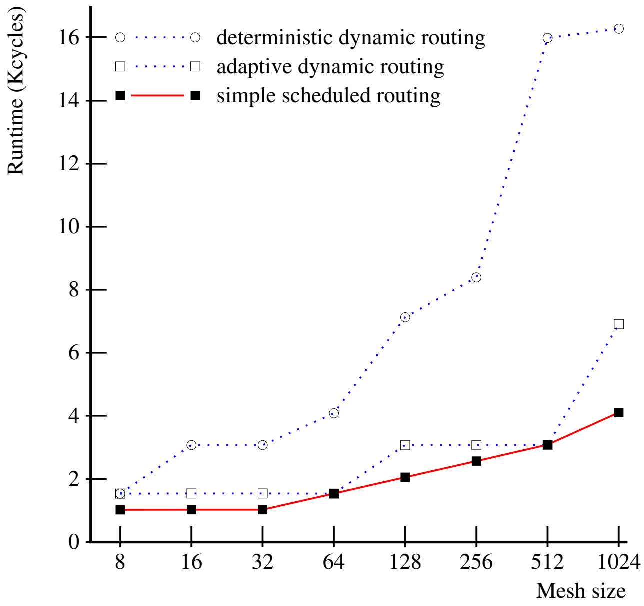


Figure 1-3: Bitreversal routing (dynamic vs. scheduled routing)

The graph shows that the latency (expressed in cycles) for the two dynamic routing techniques is worse than for scheduled routing, and rising faster as the graph grows larger. Adaptive routing comes close in cycle count to scheduled routing, but to do that it must pay a price in router complexity and resulting slower cycle times.

1.2.3 In-Router Algorithms

Beyond raw routing speed, however, scheduled routing offers the ability to build something of the structure of the algorithm into the network without compromising the speed of message delivery.

Consider the parallel prefix operation, for example. It takes an array of values $a_0, ..., a_{n-1}$ and an operator $\otimes$, and returns, on each node k, the value $a_0 \otimes a_1 \otimes ... \otimes a_k$. Any associative operation can be used for $\otimes$: add (integer or FP), OR, AND, XOR, MIN, MAX, *etc.*, as well as any user-defined operation; operands can be of any length. One well-known use of parallel prefix is the carry-lookahead adder. On a scheduled router, the prefix implementation can be partly expressed in the router, with nodes multicasting partial sums to all their descendants in the prefix tree with a single message send. As a result, there is no extra overhead in the processor from having to resend the data to its children, and the processor simply does the minimum number of reads and writes to the network necessary to provide arguments to the prefix operation being computed.

Other complex communication structures can also be built in the router to take data where it is generated and deliver it to where it is needed without requiring complex multicast setups in the processor, or requiring processors to forward data within the mesh. For example, data can be scattered and gathered within the mesh by constructing the router code to deliver the first word on a given stream to its processor and route the remainder of the message to a neighbor processor. Conversely, a stream could be set up that took data from its processor and forwarded it, then reconfigured automatically to take data from its neighbor only. Such operations on a traditional online-routed machine require multiple messages, each with its own setup time and header overheads.

1.2.4 Stream Prioritization

Scheduled routing can also be useful in situations where different streams of message traffic have different bandwidth needs. By preallocating differing amounts of schedule time to each stream, bandwidth is divided as necessary for efficient application performance, even when all the streams are being driven full-out: the higher-bandwidth streams don't crowd out the lower-bandwidth streams, nor do a number of lower-bandwidth streams combine to reduce the required bandwidth on one higher-bandwidth stream. Message prioritizing in traditional online routing systems is similar to this approach, but does not offer as fine a degree of control over the relative bandwidth needs of a collection of streams.

As an example, consider a rendering engine that generates images in real time using inter-node communications. Periodically, the system may change to a different image by distributing a large data set to all the nodes. However, the time-critical communications between the nodes as they render the last few frames of the image should not be compromised by the high-volume data set for the next image being downloaded into the mesh.

Similarly, it is also possible to tune a stream's latency to meet application needs. A given stream may be low bandwidth, but may want to run with as little latency as possible. This need can be met by appropriate scheduling, without compromising the ability of high-bandwidth streams to run at their full designated bandwidth.

1.2.5 Header Traffic

All online routing algorithms require some form of overhead associated with each message, typically in the form of a message header of one or two words holding the destination address, length, and so forth [18]. Offline-routed messages generally require no packet headers, since the message is identified by the scheduled time of transmission rather than by any data that it carries with it. In applications that send a large number of short messages, the headers can constitute a noticeable percentage of the total traffic carried, and avoiding their use can increase the bandwidth available to data.

As a simple example, consider an example where communications are typically messages with four data words; in this case, a dynamic router would add an additional word, increasing traffic in the mesh by 20%. All things being equal, a scheduled router would see 20% better bandwidth, along with improved latency.

1.2.6 Non-Cartesian Networks

Not all networks are as easy to route for as the Cartesian network; there, the e-cube algorithm easily achieves 100% of the bisection bandwidth on random traffic (consider two random nodes on opposite sides of a bisection; the probability of a message between them using any particular link in the bisection is equal to that for any other bisection link). Irregular networks may be convenient to build based on component availability or, for dedicated nodes, for predicted bandwidth needs among nodes. Such networks are extremely difficult to handle using online routing algorithms, since the route from one node to another may not lead through any obvious path.

Even regular topologies may be hard to get maximum performance from with online algorithms. For example, the diamond lattice network [63, 52] is an appealing network topology, with isomorphic 3D connectivity but only four neighbors for each node; this allows a simpler crossbar and more pins per I/O port for pin-limited packages. However, the diamond lattice suffers from the fact that using exclusively shortest path connections prevents the network from achieving 100% of its bisection bandwidth. In fact, no known online algorithm is able to provide full bandwidth across the bisection on the diamond lattice. Accordingly, scheduled routing can serve as an enabling factor for exploring the potential of non-Cartesian networks.

Figure 1-4 shows the difference in achievable bandwidth in a diamond-lattice network between traditional online routing and a version of scheduled routing. The graph depicts random traffic; the diamond-lattice mesh simulated is of size 455 (a 7-ary 4-diamond), with length-4 messages. The online algorithm is a suitably modified version of the Cartesian network's oblivious e-cube algorithm. As before, cycle counts are gathered by simulation of dynamic algorithms and a simple computation from scheduled bandwidths for the schedule router.

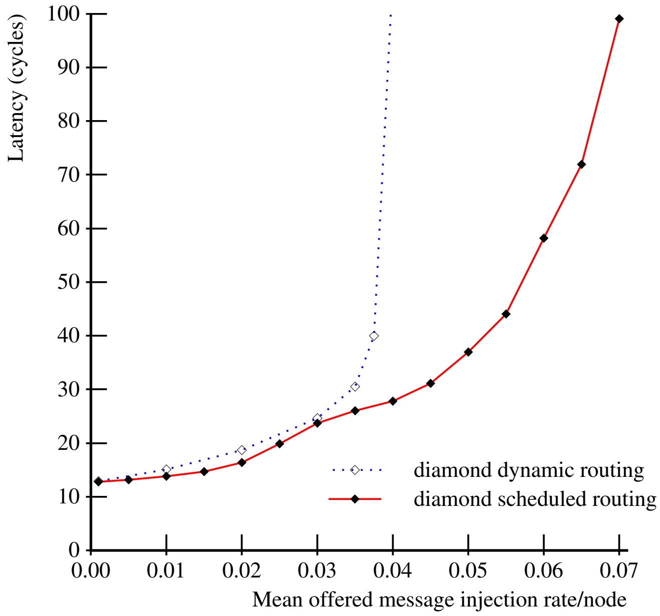


Figure 1-4: Diamond-lattice scheduled routing

The graph shows that the dynamic routing algorithm hits a maximum throughput limit around 0.04 mean offered messages per node (that is, the point when each node has a 4% chance of injecting a message on each cycle). By contrast, the scheduled algorithm continues to provide throughput (though with growing latency) up to 0.07, the largest mean offered rate per node shown.

1.2.7 Faulty Networks

Scheduled routing also allows faulty networks to be used efficiently. For a network with components that are faulty at compile-time, the same arguments apply here as discussed in the previous section. A network test can be performed before an application begins to run, and dead nodes and links can be worked around in the schedule.

If links or nodes die during the program execution, a low-bandwidth stream can be provided to notify all the nodes of the failure and switch to a checkpointing mode to dump the application state before rebuilding the communications graph and continuing. More sophisticated approaches can also be built into a runtime system that knows what communication operations a given node or stream is involved in, and can instruct neighboring nodes (using a low-bandwidth stream) to modify their schedules to accommodate the extra bandwidth demands on them caused by a neighboring dead node or stream.

1.2.8 Multiple Applications

While current research in scheduled routing has assumed multiprocessors with a single application at a time, this restriction is not absolute. Multiple applications and multiple users can share a scheduled network by allocating portions of each router to each application. The network will be timesliced along with the applications themselves, with a context switch on the processor accompanied by a context switch of the scheduled router. Direct access to the physical hardware would be mediated by some combination of hardware support (as in FUGU [47]) and operating-system support (as is done in the Exokernel [22]). The resulting multitasking system should provide similar gains to each process as could be found on a single-tasking system, though for each additional task, the smaller amount of router memory available per-task would cause extra reload time (as is discussed in Section 5.4).

1.3 Drawbacks to Scheduled Routing

Unfortunately, scheduled routing is not a panacea, and some applications will be unlikely to benefit from simple scheduled routing.

High-bandwidth applications that require primarily short-lived, highly-dynamic communications are not well-suited to scheduled routing. All three qualifiers are important to this limitation, however:

- When applications are low-bandwidth, it is easy to create multiple streams at compile time and choose among them at run time.
- When the connections are long-lived, extra setup time can be expended to create appropriate routing schedules and still benefit from high cycle speeds and low contention.
- Lastly, when communications are constant, or data-dependent in a structured manner, the scheduled router can support them directly at full speed.

Only when all of the above qualifiers are true does communication using scheduled routing become slower than dynamic routing; in this case scheduled routing must use a form of dynamic routing, as discussed in Section 4.2.

Applications with little or no compile-time knowledge of the *order* of communications are problematic in the same way as applications with short-lived dynamic communications. The notion of communication phases is dependent on being able to separate communications into sets, each with some temporal locality. If an application has a large number of possible communication streams, the result is much like the application with short-lived dynamic communications, just expressed somewhat differently.

While the thesis explicitly ignores the issue of converting high-level language inputs into structured communications, this process can be either helped or hindered by the choice of high-level language. Some languages, such as HPF [46] or Crystal [15], provide high-level communications operators; these operators easily allow for suitable communication schedules to be generated. Other languages, such as Mul-T [38] and Linda [14], have less emphasis on high-level operations and are harder to extract structured communications from.

For languages with lower-level primitives, the burden falls to the compiler to extract the necessary structured communications. As discussed above, if the compiler can only find short-lived, highly-dynamic communications in the language, the performance on a scheduled router will suffer. If the code only contains `send` and `receive` operations, or `fork` and `join`, the compiler must be relatively sophisticated to ascertain such things as a broadcast from a run-time source or a reduction.

One way to handle the problem of applications with dynamic communication characteristics is to include dynamic-routing hardware in the scheduled router. Doing so allows the compiler to schedule a certain amount of bandwidth to devote to dynamic routing; effectively, adding such support allows the hardware to support scheduled routing and dynamic routing both.

1.4 Scheduled Routing

The previous section presented the benefits of scheduled routing at a high level. This section examines the basic mechanisms of scheduled routing in sufficient detail to understand the key ideas of the thesis. Chapter 7.1 discusses the specifics of the NuMesh implementation of scheduled routing in somewhat more detail.

1.4.1 Scheduling Virtual Finite State Machines

The fundamental unit of scheduled routing is the *virtual finite state machine*, or VFSM. A VFSM handles moving a particular stream of data through a particular node. It includes some buffering for that particular stream of data as well as annotations (called the VFSM's *source* and *destination*) that indicate where its data come from and go to. Figure 1-5 shows a simple VFSM for a stream of data moving from node one through node two's router to node two's processor. Linking together all the individual VFSM sources and destinations on each node yields an overall VFSM chain that moves the data from a stream's original source to its final destination; in the given example, the stream's source might be node zero, and its destination is node two.

If it was only necessary to move one stream of data through each node, a single VFSM could be used as a router. However, it is much more common to need to move several unrelated

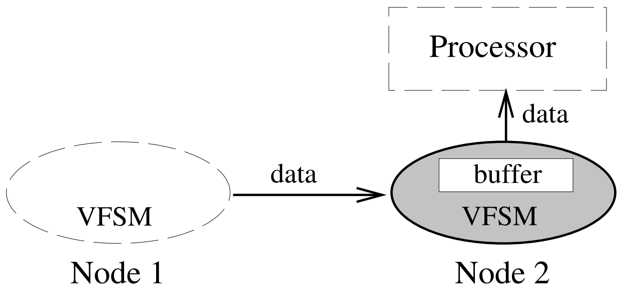


Figure 1-5: A virtual state machine transferring data

streams of data through each processor. This can be accomplished by scheduling the VFSMs in some predefined sequence on a node, essentially time-slicing the router hardware into multiple virtual finite state machines (thus the 'V' in 'VFSM').

However, it is necessary to be able to differentiate two streams of data that just happen to pass between the same two nodes at some point. To do so, the VFSMs are scheduled globally, so that when a VFSM on one node does a write at a particular time, the VFSM that reads the same stream of data on the neighboring node is scheduled one clock cycle later. Figure 1-6 shows how VFSMs might be scheduled on three nodes (in a one-dimensional array), carrying three streams of data among them. Stream A, with bandwidth 0.5, carries data from nodes 0 to 2; streams B and C, with bandwidth 0.25, carry data from 0 to 1 and from 1 to 2, respectively. Each word of data moves through the mesh at one cycle per node; the number of times a given VFSM appears in a schedule corresponds to its allotted bandwidth.

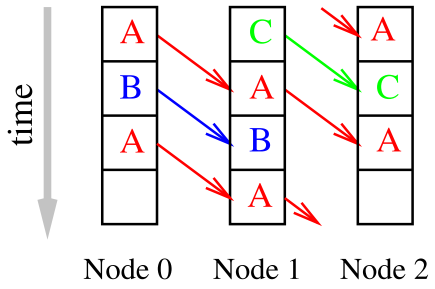


Figure 1-6: VFSM schedules on a small mesh

Another representation of how scheduled routing is shown in Figure 1-7, which demonstrates which cycles data is carried across the wires in the mesh; router nodes are shown as circles, with their attached processors in boxes. In the example, a data word might start on node 0 on the VFSM for stream A when it's scheduled at time 0, get passed to node 1 at time 1, then passed to node 2 at time 2, where the last VFSM delivers data to the processor. A similar word leaving node 0 at time 1 must be associated with stream B, and will therefore be delivered by the VFSM for stream B on node 2 to the processor, instead of forwarded to node 3 as is true for stream A. Streams of data in a scheduled routing environment are thus bound tightly to a specific set of VFSMs on the nodes chosen to route them from their source to destination.

The nodes in the mesh are synchronized at boot time, and remain synchronized from then on. Each node runs its schedule repetitively, returning to the top of the schedule after running

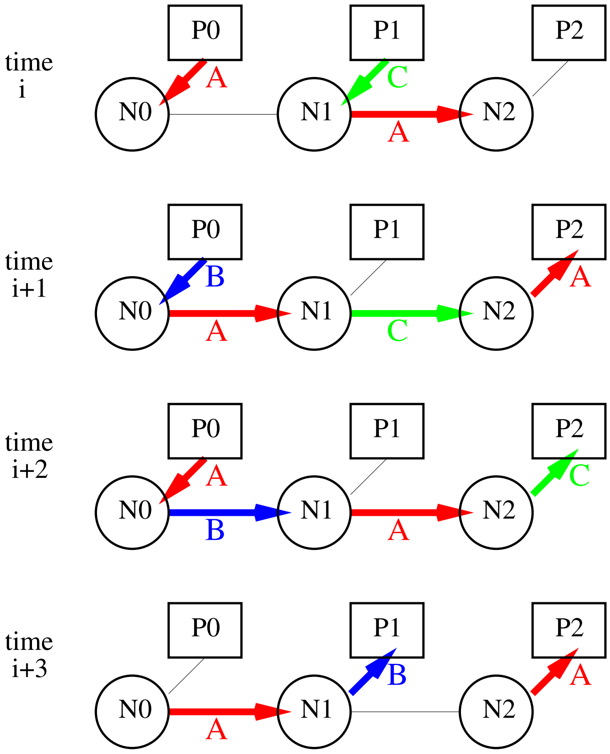


Figure 1-7: Data movement on a small mesh

the VFSMs scheduled last in the schedule. Thus, if the schedule is of length k, the 3rd *timeslot* in the schedule will be run at time $3, 3+k, 3+2k$, and so on.

The discussion so far has assumed that the physical hardware is capable of only a single communication action per cycle. Multiple physical FSMs can be included on each node that are each independently timesliced by VFSMs. These physical FSMs are called *pipelines*, using terminology that reflects the implementation of the current NuMesh hardware. Each pipeline is capable of scheduling an independent VFSM for each cycle. Thus, on a given cycle there may be a simultaneous transfer by one pipeline of data from the $+x$ to the $-x$ directions, while another pipeline is transferring data from the processor to the $+y$ direction. Co-scheduled VFSMs must have non-conflicting sources and destinations to avoid conflicts.

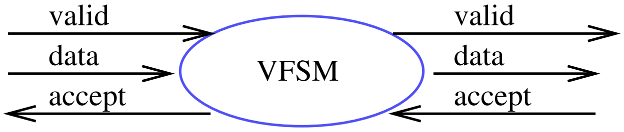
1.4.2 Flow Control in Scheduled Routing

So far, it has implicitly been assumed that whenever data is ready to be transferred it can always be accepted by the destination. However, if the stream's destination processor is busy, and the source processor is attempting to write data into the mesh, the system must handle the potential for a backlog in the stream. This problem (known as the *flow control* problem) is handled by a synchronized exchange of control signals along with the data.

Whenever a valid data word can be written to a neighbor, the VFSM asserts the *valid* line to the remote reader and sends the data. When reading, a VFSM determines whether it has sufficient buffer space to accept another word of data, setting the *accept* line to the remote writer accordingly. After setting the accept line low, the reader will simply ignore any data presented to it.

The difference from traditional online-routing schemes is that the accept line is driven by the reader at the same time as the valid line. Since the mesh is running schedules that are all offline-generated, a reader always knows when a writer is sending data, and it can generate an ‘accept’ during the same cycle as the data is written. This means that scheduled routing allows a single-cycle protocol for determining flow control.

When the writer VFSM discovers that the reader couldn’t accept the data word, it queues the word in its buffer. The next time the writer is scheduled, it will set the accept line low for *its* writer, and attempt to transfer the buffered word instead of reading a new one. Figure 1-8 shows one VFSM with its incoming and outgoing control and data lines, and summarizes what actions it takes depending on the state of those lines. These actions happen in synchrony across the entire mesh every cycle; this choreography—scheduling which VFSMs run on each node at each cycle—determines how data moves through the mesh.



Transfer Result	Action
read and write succeed	transfer success
read success, write fail	word buffered
read fail	node transfers buffer word if valid

Figure 1-8: NuMesh transfer actions

1.4.3 Router Reprogramming

The router may be updated under program control. Each scheduled router is assumed to have a schedule memory which can hold multiple schedules (of which only one is active). Each location in the router in which a schedule can be placed has some index associated with it. New schedules can be downloaded to the router by specifying the desired index and providing the schedule information. COP requires a router schedule memory large enough to hold at least two complete schedules, so it can load a new schedule into the router while the previous one continues to run.

Similarly, each router is assumed to have multiple VFSMs, of which only some are used in a given schedule. VFSMs can be used for the same operator in multiple schedules (as is discussed in Section 5.1); however, VFSMs can also be *reused* by changing the source and destination annotations associated with the VFSM, then using the VFSM in a new schedule. An index is associated with each VFSM that allows it to be referenced both for scheduling and for reprogramming its annotations.

1.4.4 Other Characteristics of Scheduled Routing

Processor Interface. At each end of a stream of data, the router must deliver data to and from the processor itself. The interface between the two can be specified as just another flow-controlled source or destination for the VFSMs. The architecture could require that the processor be synchronized to the mesh sufficiently to know that a data word in timeslot 5 belongs to a certain stream. However, to allow for a more general, asynchronous interface, multiple interface addresses are provided. The router can deliver data from stream A consistently to interface address 12; the processor can then read interface address 12 at its leisure to find data from stream A.

Scheduling Delays. Streams can also be delayed one or more cycles on a node. Normally, a VFSM is scheduled on a given cycle, reads the data from its specified source, then writes the data to the appropriate VFSM on the next node on the following cycle. However, global scheduling constraints may make it infeasible to schedule a stream on consecutive cycles in each node in its path. Accordingly, it may be necessary to delay a few cycles to find a suitable path. This can be done by allocating two VFSMs on a node to a stream. One reads from the neighbor node, then hands off to the other VFSM; the second VFSM, scheduled later, reads from the first one and then writes the data word to the destination node.

Data Forking. A data 'fork' allows a stream of data to be forked to two (or more) destinations within the router. This feature can be used to arrange a multicast stream that has multiple destinations. A data fork can be implemented by consecutively scheduling a set of VFSMs, the first of which is responsible for reading the data from the neighboring node, and each of which handles one of the destinations. Multiple data forks can be placed in a stream; this creates a distribution tree that can send the data to the destinations in an optimal manner.

1.5 Thesis Outline

The rest of this thesis is organized as follows: Chapter 2 provides a brief overview of the communications language, COP, motivated by the demands of reprogrammable scheduled routing. Chapter 3 looks at related work in the field. Chapter 4 examines how to handle data-dependent communications with a scheduled router; Chapter 5 examines how to handle multiple phases of communications in an application using scheduled routing. Chapter 6 looks at the techniques used to search out optimal implementations and phases. Chapter 7 provides details of the two current backends of the COP compiler, one for the NuMesh reprogrammable scheduled router and one for traditional dynamic routers. Chapter 8 presents some comparative data on performance for the two backends. Finally, Chapter 9 presents the conclusions drawn from the research. Two appendices are also included, one giving more details of the COP language, and the other providing a list of operator implementations.

Chapter 2

The Communications Language

This chapter presents an overview of COP, the Communication OPerators language (think of a traffic cop directing the flow of network traffic). The language features presented here are motivated directly by the needs of scheduled routing, and are sufficient to place the related work in context. Later sections will include further discussions of language features as they are motivated; a full reference for the language can be found in Appendix A.

COP assumes a multiprocessor model of *nodes* interconnected in a network; each node has a processor (*e.g.*, a conventional off-the-shelf CPU) and a local memory. Each node includes a *router* that handles the connections among the nodes. A COP program expresses all the communications in an application from a processor-to-processor perspective. The application is thus completely freed from having to know how data is moved around within the network, and simply calls COP compiler-generated functions to perform all I/O.

The chapter begins by presenting some motivation for a separate communications language; Section 2.2 introduces COP 'operators'. Sections 2.3 and 2.4 discuss COP's support for phases as well as for grouping operators together. Restricting operators to subsets of the mesh is considered in Section 2.5, and expressing data-dependent communication in Section 2.6. Section 2.7 discusses the COP computational model, and Section 2.8 briefly outlines COP's extensibility. Finally, Section 2.9 gives some COP examples, and Section 2.10 discusses design issues for the language; a summary section concludes.

2.1 Motivation

Programming a scheduled-routing communications substrate requires a traditional language compiler to become substantially more complex. There are several simple approaches that can be taken to try to avoid some of this difficulty:

- Limit the application to a set of constant streams in a single phase. The introduction discussed why such a model is inadequate, but it does have the advantage that it is relatively straightforward to implement and requires no close coupling between the stream router and the high-level language (HLL) compiler.

- Create the necessary schedules by hand for each application (or perhaps by an application-specific compiler, similar to [4]), allowing for any necessary data-dependency and schedule-switching by a close knowledge of the particular communications schedules created for that application. This solution, while allowing the necessary power for the given application, is insufficiently general.

Instead, let us consider the more general compilation possibilities. While it would be possible for an application to handle all its I/O requirements internally, the COP language allows for a high-level language (HLL) compiler to specify the communications using COP instead and leave all the communication details to the COP compiler. This *split-language* approach separates the burden of offline communications generation from the usual, language-dependent compiler chores such as partitioning and placement. This separation of responsibility means that many HLL compilers can use COP, and COP can handle a variety of different hardware backends, so that M languages and N hardware backends require only $M+N$ compilers rather than MN. Previous HLL compilers have not taken this approach because a dynamic router is a relatively trivial target for a communications compiler, unlike a reprogrammable scheduled router.

Partitioning the communication and the computation in this manner is a delicate balancing act. If too little information is provided to the communications compiler, the resulting application may not run as fast as possible, limited by its communications. On the other hand, if too much information is provided, the communications compiler requires a very closely-coupled interface to the HLL compiler, and becomes effectively tied to that compiler.

The remainder of this chapter presents the COP language; then, in the final section, the issue of separating the compilation of communications and computation is addressed again. For now, it is worth observing that COP is motivated additionally as a research vehicle, despite any potential drawbacks associated with its nature as a stand-alone communications language. Structuring the compiler around COP allows for an exploration of reprogrammable scheduled routing without the distraction of more routine compiler issues, and still offers insights into the mechanisms required for a compiler to target a scheduled router at any level of abstraction.

A *communication language* such as COP is distinguished from other languages by two key features:

1. It does not attempt to express an application's computation, leaving that to a separate compiler;

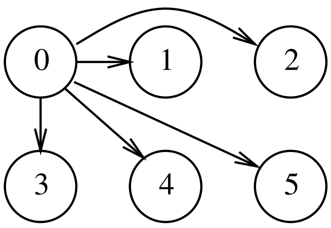
2. It is intended to be compiled rather than evaluated at run-time, thus providing a structured form of the actual application's communication needs.

COP, in fact, is not a language in the traditional sense of the word. A COP 'program' is fundamentally just a hierarchy of communications expressions; the language has no run-time variables or control structures. Since the input is evaluated in a Lisp environment, a wide variety of control structures are available at compile-time, but the Lisp environment basically functions as a macro language designed to make COP programs easier to specify.

2.2 Expressing Communications as Operators

A first question to ask for a language specifying communications is where to place the abstraction layer for the communication operations themselves.

The communications compiler should have the freedom to implement communications patterns in an optimal manner. For example, consider the case where the application wants to perform a broadcast to the mesh from node zero (as shown in Figure 2-1). The best technique to use will depend on the size and topology of the mesh, the relative speed of the interface and the routers, and the capability of the routers, among other things. Some options for a broadcast include a direct multicast (for a scheduled router), sequential writes to each destination (for a dynamic router), a flood-fill technique (for a dense mesh), or a tree-broadcast technique.



`(broadcast 'b 0)`

Figure 2-1: Broadcast abstraction

The application can not and should not attempt to determine which is the best way to perform the broadcast. Each technique requires the processors to interact differently with the mesh, so the application can't specify the broadcast semantics by simply listing a set of streams that it wants to use. Therefore, COP provides the necessary semantics by placing the abstraction barrier higher.

Accordingly, the basic unit of a COP program is taken to be the *operator*; each type of operator corresponds to a particular communication pattern. Operators range in complexity from a simple `stream` operator, expressing a single stream of data with source and destination known at compile time, to a `prefix` operator that performs a parallel prefix operator on the involved nodes. (The 'operator' framework has other advantages, which will be described throughout the chapter.)

An example operator follows; this one line is also an acceptable complete COP program (perhaps for an application that simply broadcasts work to clients who display the results locally):

```
(broadcast 'b 0)
```

The first argument to the operator gives the operator's *label*; this value is used to label the COP output for this operator (this issue is revisited later). The second argument, for a `broadcast`, specifies the source of the broadcast. The COP compiler can choose any of a variety of operator *implementations* for the broadcast, based on what will perform best in the particular context specified by the application and the target mesh. Additional *optional arguments* can be specified as annotations, giving information on approximate message count, maximum required bandwidth, and the like; these are introduced when motivated in the text, and enumerated in Section A.3.

A wide variety of global operator types are available in the COP library. Included among the set of operator types are `stream`, `reduce`, `allreduce`, `prefix`, `broadcast`, `collect`, `barrier`, `cshift`, `eoshift`, `transpose`, `bitreverse`, and `permute`. The `stream` operator is particularly noteworthy, as it allows arbitrary connections to be specified between pairs of nodes when no higher-level operator exists to describe the communication. Appendix A.1 describe all of the COP operator types in more detail.

Expressing communications as operators yields an additional advantage. As was discussed in Section1.2.3, scheduled routing allows for higher-level routing constructs to be executed (at least partly) within the routers themselves. The example used there, parallel prefix, expresses underlying semantics not readily reduced to simple, efficient communication primitives. By including such high-level operators as COP primitives, it becomes possible to portably access these features, increasing applications' performance on more capable routers without limiting the applicability of COP to simpler routers.

Additionally, the fact that scheduled routing requires *global* scheduling matches well with the use of *aggregate* high-level operators in COP. Since the routers' schedules must be controlled in a coordinated manner, aggregate operators that require global knowledge are an appropriate match.

The simple, one-line COP program above is used again in Section 2.7, where it is used to illustrate the interface between COP code and application code.

2.3 Supporting Phases

An important feature of any communications language is that it support phases: without such support, all the communications must be merged into a single phase. For scheduled routing, such merging would result in disastrously little bandwidth available to each operator, as all the operators would have to share the available scheduled bandwidth.

As is discussed in the next chapter, some existing communication languages support fixed, user-specified phases. That is, the communications language explicitly indicates the boundaries between phases. However, for the HLL compiler to attempt to derive phase boundaries is for it to take on itself the job of the communications compiler. Determining phase boundaries depends on message traffic, the effects of scheduling multiple operators in one phase, and the overhead for changing out of a phase with a given set of operators in it, among other things. Accordingly, making such decisions is beyond the scope of the HLL compiler.

It is also worth observing that since the notion of a 'phase' does not contribute much to compiling for a dynamic-routing target, it would be inappropriate to require phases to be specified in a language intended to be used for dynamic routing as well as scheduled routing.

2.3.1 Choosing Optimal Phase Boundaries

When determining where to place the phase boundaries, the COP compiler must weigh a variety of factors. With scheduled routing, dedicating a single phase to an operator achieves the following goals:

- For a high-traffic operator, dedicating a single phase to it may be critical to achieve the necessary performance.
- Some operator implementations generate complete schedules for all the nodes (as is discussed more in Section 4.1), and therefore require a separate phase.

However, sharing a phase among a number of operators has several benefits

- It allows the operators in the phase to be used unpredictably by the application (as discussed in Section 2.4.1).
- It eliminates the overhead involved in changing phases between the operators.
- It reduces the footprint of a given set of operators in the router, thus allowing for more operators to be held in the router at once.

As a simple example, let us suppose it takes 20 cycles to switch from one phase to another for a given application and mesh. If there are two operators to schedule, the compiler can predict how long the communications for each phase will take when the operators are scheduled together or in separate phases. Let us suppose that, if scheduled in separate phases, each operator will get 0.667 bandwidth, whereas scheduled into a single phase together they will only get 0.500 bandwidth each. If each operator is used to send one message 50 words long, it will take 100 cycles per operator (200 total) if scheduled in the same phase, and 75 cycles (170 total, including 20 for phase change time) if scheduled in separate phases. Thus, the compiler will choose separate phases. If the messages were shorter (less than 40 words), a single phase holding both operators would be a better choice.

2.3.2 Expressing Time

Accordingly, operators are grouped together simply by expressing that they are run sequentially (thus, whenever the application performs I/O on one of the operators in the group, it is guaranteeing that it has finished performing I/O on the earlier operators). Using this information, the COP compiler can determine where within a sequence of sequential operators to place phase boundaries (if any). For example, a broadcast to all the nodes, followed by a reduction, would be specified as

```
(sequential
  (broadcast 'b 0)
  (reduce 'r 0 'myfunc))
```

(The second and third arguments to the `reduce` specify respectively that the result should be left on node zero, and that the high-level application is supplying a function `myfunc` to combine values on each node.) The COP compiler can now choose whether to compile this construct to one phase or to two phases based on timing predictions.

If a series of operators will be used repetitively, the `loop` construct is used instead of `sequential`. A label for the loop is specified (as is done for operators), along with an approximate loop count. A loop construct indicates that the last phase in the loop may be followed by the first phase, which is (as is discussed later) important for correct phase changes on a scheduled router. Thus, if the broadcast and reduction were repetitively performed, the COP program might be given as

```
(loop 'l 1000
  (broadcast 'b 0)
  (reduce 'r 0 'myfunc))
```

2.4 Grouping Operators Together

2.4.1 Multiple Operators

Of course, an application can't always guarantee that it will use operators sequentially, so a construct is needed in the language that indicates that a group of operators may be used in any order. This is expressed with the `parallel` construct. For example, if there are three specific 'root' nodes that broadcast at unpredictable times, the COP description might be

```
(parallel
  (broadcast 'b0 0)
  (broadcast 'b1 7)
  (broadcast 'b2 12))
```

Operators appearing within a parallel construct must be scheduled into a single phase to allow the application to use them all unpredictably. The parallel construct is thus more restrictive than the sequential construct, since the COP compiler is free to treat a sequential as a parallel, but not the reverse.

An alternate form of the `parallel` command is available as a macro which can parameterize multiple instances of a parallel command. The `doall` command takes a variable specification (in the same form as a `let` statement, with list values for the variables), and converts it into a `parallel` command by concatenating the results of evaluating the body with the variables bound to all combinations of their list members in turn.

2.4.2 Continuing Operators

Sometimes it may be possible to isolate a set of operators which are used sequentially, but one or more other operators may be used unpredictably around the sequential operators. Thus, it is important that the language be able to express this notion to allow for efficient scheduling in this case. COP handles this by allowing the grouping constructs to nest arbitrarily.

As an example, consider stdio-style input, output, and error streams; the stdin stream would be broadcast from a master node to all the nodes, and stdout and stderr would fan in from all the nodes to the front end. Likewise, debugging streams might usefully be placed in the application, in parallel with the application's primary communications code. To accomplish this, the application places a `parallel` construct around the 'continuing' operator (*e.g.*, `stdin`) and a `sequential` group holding the operators that will be sequenced through while the continuing operator is active. Thus, to define a stdin stream that all the nodes can read while several different operations happened, you might do:

```
(parallel
  (broadcast 'stdin 0)
  (sequential
    (broadcast 'b1 17)
    (broadcast 'b2 0)
    (reduce 'r 0)))
```

If the compiler breaks the sequential portion up into multiple phases, the `stdin` operator will be defined in all of them. Any data sent using that operator will remain valid in the network even during router phase changes.

2.4.3 Multiple Threads of Control

One important feature that the language should be able to express is multiple threads of control. In other words, the language should be able to represent a more MIMD[1] model with different regions of the mesh executing different parts of the application. (Multiple threads on a single node are orthogonal to this issue, and their communication needs must be expressed using the multiple-operator methods of Section 2.4.1.)

Given a set of operators A0, A1, ... that are active on the left side of a mesh, and a similar set B0, B1, ... active on the right side, the concept of multiple threads of control can be represented as follows:

```
(parallel
  (loop A0 A1 ...)
  (loop B0 B1 ...))
```

The result will be two threads of control, one looping through the A operators on the left, one through the B operators on the right. The application thus has multiple phases that are divided not just in time but in space as well. The *spatial extent* of a phase is the region of the mesh in which communication and computation are occurring for the operators in that phase. In the above example, the A operators' spatial extent is the left side of the mesh.

For dynamic routers, the spatial extent is largely irrelevant for communications, though the compiler will assume the communications are mostly non-overlapping when it attempts to find efficient implementations of the operators on either side. For scheduled routers, multiple simultaneous phases implies that, rather than computing fully global schedules, "global" schedules are computed for subsets of the mesh independently. The issue of extents is discussed in detail in Section 6.1.

2.5 Communicating in Subsets of the Mesh

Applications need to be able to express that a given operator functions on a subset of the whole mesh (as was assumed implicitly when discussing multiple threads of control, above). For example, the user may wish to specify a broadcast on the first column of a mesh only. Accordingly, the language includes a `subset` operator that can be used to specify that the operators within it run on the given subset. Multiple operators may appear within a `subset` construct, in an implicit sequential context. Given a 4×4 mesh, a first-column broadcast would be specified as

[1] Multiple Instruction, Multiple Data

```
(subset '((0 4 8 12))
  (broadcast 'b 0))
```

Often, what an application needs to do is express that it is performing a broadcast, for example, on all the columns of a mesh. While this could be done syntactically by placing multiple `subset` constructs within a `parallel`, the `subset` construct itself can be used to express this more elegantly. The first argument to the `subset` construct is actually a list of subsets; the resulting COP code is equivalent to wrapping a `parallel` around separate `subset` constructs, one for each member of the list. To express a broadcast on each column of a 4×4 mesh can be done by:

```
(subset '((0 4 8 12) (1 5 9 13) (2 6 10 14) (3 7 11 15))
  (broadcast 'b 0))
```

This is equivalent to four `(broadcast 'b)` operators in parallel, each specifying a different column of the mesh. The broadcast source (zero) is taken to be *relative* to the subset, so the broadcast on the second column is from node 1, on the third column from node 2, and so forth. The language allows multiple *instances* of an operator, as in this case; however, multiple instances can't be specified as reading from or writing to the same node, and they must appear in the same context in the COP program. The four broadcasts specified implicitly in the example above are thus considered to be a single operator defined with four instances.

Some global variables are defined in COP to make parameterizing COP programs easier; they include `*nodes*` (the total number of nodes), `*xsize*`, `*ysize*`, and `*zsize*` (the size in each dimension). Like operator operands, these values are relative to the current subset context. It is also possible to nest `subset` constructs, with the `subset` argument being, again, relative to the enclosing subset context. Subsets are dynamically scoped, rather than lexically scoped like the rest of the language.

2.6 Expressing Run-Time Values

A key issue is how to handle values that are not known until run time. For example, for Gaussian elimination a pivot row may be chosen based on the input dataset, then broadcast to all the other rows. With dynamic routing, this sort of thing is easily done. With scheduled routing, as much information as possible needs to be supplied to the compiler so it can determine how best to implement the operator.

Run-time values are expressed with a special compile-time construct, `(runtime)`; this construct is a 'pseudo run-time' value, since its only use is at compile time, but it represents a run-time value. Used in lieu of a constant argument, this construct indicates that the value in question will be provided by the application at run time. For example, broadcasting from an unknown row position in a 4×4 matrix is expressed as:

```
(subset '((0 4 8 12) (1 5 9 13) (2 6 10 14) (3 7 11 15))
  (broadcast 'b (runtime)))
```

The possible values that a `(runtime)` operand can take on are limited by the operand it is used for. Thus, for a `broadcast` source, the run-time argument must be a valid node number from the current subset in effect; in this example, it must be in the range $0...3$.

The use of this construct is a large part of the power of COP, since it describes a limited form of data-dependency which can be implemented by using a variety of efficient techniques to avoid defaulting to pure online routing. The presence or absence of such run-time values for a given operator determines the extent to which the operator's schedule can be computed strictly at compile time, partly at compile time and partly at run time, or entirely at run time. The HLL compiler does not need to know how the compiler implements the operator, so the language does not expose where the schedule-generation actually takes place.

This construct does *not* indicate that an operator may (for example) deliver data to a different destination on each invocation; the value of the run-time argument must be provided before the I/O function is called, and the same value must be provided on all the nodes involved in the operation. If the operator is used within a loop, a new run-time argument may be provided each time through the loop.

Constraining Run-Time Values

An important requirement for a run-time value is that the application be able to provide as many constraints on its value as can be determined at compile-time. Without the ability to express these constraints in the language, in many cases an unknown communications pattern might have to be expressed with dynamic routing, at some performance penalty. The more information available on the run-time values, the better the compiler will be at choosing the most efficient implementations.

An important implicit constraint is the type of the operator itself. For example, a `permute` operator that specifies a run-time permutation yields more information than a set of `stream` operators with run-time sources and run-time destinations. In particular, it may be possible to efficiently generate a VFSM schedule at run-time to handle a permutation, whereas handling arbitrary communications with a run-time stream operator may require a much less efficient solution.

The `(runtime)` construct takes optional arguments that can be used to constrain the possible range of run-time values. The `:values` argument allows the application to specify that a given run-time value will only take on values from the specified list. Thus, a broadcast that may occur from one corner of the mesh or the other can be specified as:

```
(broadcast 'b (runtime :values '(0 15)))
```

A runtime value for an operator must normally be specified consistently by every node in the operator's subset. This allows for any necessary changes to be made consistently to the routers' schedules across the mesh. However, sometimes the application does not have complete knowledge on every node, and a way to express that is useful. An optional `runtime` argument, `:distributed`, indicates that knowledge of the run-time value is distributed across the mesh. For a broadcast, this means that each node only knows whether or not it is broadcasting, rather than knowing which node is broadcasting; this is expressed as:

```
(broadcast 'b (runtime :distributed t))
```

Implementations of operators which use this partial-knowledge flag can be as efficient, in some cases, as those in which all nodes have complete knowledge.

2.7 Computational Model

With the language presented in outline, the next section turns to a brief look at the computational model presumed by COP, and describes how the COP compiler interacts with a HLL compiler.

As illustrated in Figure 2-2, the front end invokes the COP compiler and passes in the communication structure in the form of COP code. The compiler generates a set of functions for each operator that can be used on each node of the mesh to perform communications. These functions are processor code, suitable for integration into the HLL compiler's internal representation. The actual router schedule code for scheduled routers is embedded in the generated processor code, since it is the processor that downloads router code at the appropriate moments. The HLL compiler then generates object code suitable to run on the processors of the mesh (either as a single program or as multiple node programs, depending on the HLL compiler).

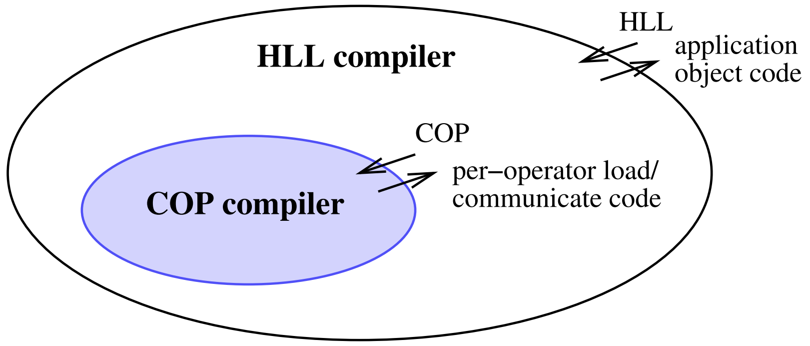


Figure 2-2: A high-level view of code generation

The current COP compiler assumes that C is the internal representation, and the returned processor code contains suitable C functions. Accordingly, the C compiler is used to perform the integration step between the COP output and the application-specific code.

Each operator's first argument, as previously mentioned, is a label. These labels define a flat namespace that the COP compiler uses to name the returned functions: thus, the `(broadcast 'b 0)` operator would cause functions named `b_write()` and `b_read()` to be returned by the compiler (among others). These returned functions are integrated into the HLL compiler's internal representation at the points where communication is performed.

2.7.1 The I/O Functions

For operators with a *directional* communications flow, such as `broadcast` or `stream`, the compiler generates *write* and *read* I/O functions. The writer node (*e.g.*, the broadcast source) calls the *write* function with the value to broadcast, while the other nodes in the subset call the *read* function. The other class of operators are called *functional*, where nodes both provide and receive values; these include, for example, `reduce`, `prefix`, and `cshift` (circular shift). For these operator types, the COP compiler generates a single *func* function, which on all nodes takes an argument and returns a value.

Since COP's output is a typed language (namely C), operators are currently required to specify the type of the input and output. By default, operators are assumed to pass and return one-word integer arrays. The `:type` optional argument for operators can specify different types; the basic types are `int` or `float`, with a suffix of '`32`' or '`64`' to determine word size, and an optional suffix of '`*`' to indicate an array type. For array types, the `:m` argument is used to specify the message length, if more than a single value is to be passed at once.

2.7.2 The Load Function

An additional function is associated with the compiled output for all operators. This is the *load* function, which coordinates the use of the router for that operator. It serves a number of purposes:

- It loads any necessary router information before beginning to use the operator.
- Its arguments are the values for any `(runtime)` operands, and it is responsible for updating the router when those values change.
- For the first operator after a phase change, it coordinates any necessary overhead in changing from the previous phase (such as barriers or required delays).

The I/O functions could be overloaded to handle all of this, by adding additional arguments for run-time operators, then testing when the values change, or when the operator is used for the first time, and updating the router as necessary. However, this seemed inelegant and inefficient, since the HLL compiler knows where to insert the code to perform these tasks before calling any I/O functions.

For a sequential operator, the load function must be placed before the operator's first use of its I/O function(s), and after the last use of the previous operator's I/O function(s). Where sequential operators are placed into the same phase, only the first sequential operator will load the router code for that phase. For parallel operators, the load functions for the operators must all be called before the HLL compiler calls any of the I/O functions for the operators.

As is mentioned above, `loop` constructs also have a label. This label is used by the compiler to generate a load function for the loop; such a load function may contain router reprogramming commands that can be hoisted out of the loop, and thus executed only once per entering the loop. No I/O functions are associated with loop labels.

2.7.3 COP/HLL Integration Example

Let us consider the simple broadcast example first mentioned in 2.2, `(broadcast 'b 0)`. Figure 2-3 shows a tiny but complete application; Lisp is used for the examples to simplify the presentation. The first part of the figure contains the one-line COP program for the application. The second part shows the application code for the master broadcasting node (node zero), and the third part shows the code for the slave ('worker') nodes in the mesh. Node zero is passed a list of 'work' of some sort, it broadcasts it to the workers, and they perform appropriate work based on, *e.g.*, their coordinates in the mesh. The `worker-function` is the function that does the work, and it is not included in the example; it could be any interesting function (such as, say, computing a portion of a Mandelbrot set).

COP code:

```
(broadcast 'b 0)
```

Application code for the master node:

```
(define (main input)
  (b-load)
  (foreach i input
    (b-write i)))
```

Application code for the slave nodes:

```
(define (main)
  (b-load)
  (while t
    (worker-function (b-read))))
```

Figure 2-3: Simple application code

In Figure 2-3, the HLL compiler is assumed to have generated a COP program with an arbitrary label ('b'), then used that label to generate the appropriate calls to the I/O and load functions (`b-load`, `b-write`, and `b-read`).

If a scheduled router is the target, the COP compiler will take the COP code and return functions similar to those in Figure 2-4. The load function downloads appropriate data (*e.g.*, VFSM configuration and a schedule) to the router using the hypothetical `cop-reprogram-router` function, then starts the newly-loaded phase running; `*new-phase*` represents the schedule index of the phase whose information has just been downloaded. The `b-write` and `b-read` functions check that they are running on a legal node, then write to the processor/router interface (interface address seven is used in the example). `*node*` is the subset index of each node in the set.

```
(define (b-load)
  (cop-reprogram-router (nth '((#<router code>) ...) *node*))
  (cop-start-phase *new-phase*))
(define (b-write val)
  (if (= *node* 0)
      (cop-write 7 val)))
(define (b-read val)
  (if (!= *node* 0)
      (cop-read 7)))
```

Figure 2-4: COP output for simple application

The HLL compiler takes the output code returned in Figure 2-4 and integrates it with the application code that it has generated for carrying out the computation. Figure 2-5 shows the final, integrated code for the master node (node zero). The code has been specialized for node zero, so references to `*node*` have disappeared. The HLL compiler will convert this into object code (including the router code); the object code will then be downloaded to node zero at run time, while the matching code for each slave node (not shown) is similarly downloaded.

Final application code for the master node:

```
(define (main input)
  (cop-reprogram-router '(#<node 0 router code>))
  (cop-start-phase *new-phase*)
  (foreach i input
    (cop-write 7 i)))
```

Figure 2-5: Final HLL code for node zero

2.8 Extensibility

An important characteristic of a language is extensibility. COP offers two forms of extensibility; more details on both are provided in Appendix A.

The simplest form of extensibility is for a user (or HLL compiler) to use Lisp primitives to define a new operator in terms of existing operators. The `transpose` operator is a (pre-defined) example of this; the operator is defined in Lisp to return a `parallel` group of appropriate streams.

A more substantive way to extend the language is also needed, however. Consider the following scenarios:

- An operator might require more sophisticated use of the routers than can be expressed by the existing implementations.
- An application might have an operator subset with a particular structure that an optimized implementation could take advantage of.
- An application might use a particular set of runtime values for an operator which an optimized schedule-generator could handle.
- Or, more generally, an operator may just not be as efficient if mapped down to streams.

To handle this, COP allows users to provide operator implementations which can take effect under the appropriate circumstances. Such operators are simply written in C (or any other language with a compatible application binary interface), then dynamically linked by the COP compiler at compile-time and chosen for use in exactly the same way that pre-defined implementations are chosen.

It is also possible to define entirely new fundamental operators, but it requires modification of the compiler to characterize the new operator type appropriately. Having done so once, implementations of the operator can be written as needed.

2.9 Example COP Code

This section gives two moderate-sized examples of COP code; others are presented in Appendix A. No HLL code is shown; instead, the computational aspect of each example is outlined.

The first example is matrix multiply, as shown in Figure 2-6. The COP code implements a simple parallel matrix multiply of $A \times B$, where columns of the A matrix are broadcast from a different column of processors on each iteration, alternating with shifting portions of the B matrix vertically.

```
(loop 'mul *xsize*
  (subset rows
    (broadcast 'b (runtime)))
  (subset cols
    (cshift 'c -1)))
```

Figure 2-6: COP code for matrix multiplication

The variables `rows` and `cols` are assumed to hold the necessary subset specifiers; `rows` is a list of subsets, each subset corresponding to a group of nodes to handle computations for a single row, and similarly `cols` for columns in the matrix. (Code generated by a frontend compiler would typically have these values expanded in place.) The subsets might correspond to x and y cross-sections of a 2D mesh, or, *e.g.*, Grey-code mappings on a 3D array.

As a larger example of how this language looks in practice, Figure 2-7 is an implementation of Gaussian elimination used to solve the matrix equation $Ax = b$. Due to the cyclic distribution of the standard Gaussian implementation, it is not necessary to worry about explicitly excluding rows that have already been eliminated, since a given processor will be involved in most operators at every step.

In the figure, the `r0` operator performs a reduction to get the index of the row with the largest ith element (the pivot row), along with the value of the element, and distributes it to all the nodes in each subset. The `b1` operator broadcasts that index, along with the computed scale factor for the specific row, from the ith element of each row to the rest of the row. The `s2` operator sends the ith row to replace the chosen pivot row, and the `b3` operator then broadcasts the pivot row to all the other rows, so an update step can be performed. These four operations are repeated for each row in the matrix. Finally, the back substitution is done by looping back up through the rows; `s4` sends the ith column to column zero (where the b vector is stored), and `b5` broadcasts the value of x_i to the entire column zero. Iterating through the matrix completes the back substitution, resulting in a solution for x.

2.10 COP Language Design Issues

Now that the reader has come to some understanding of the language structure, this final section considers some of the tradeoffs that went into the language design, as well as the question of a separate communication language itself.

```
; A 2D cyclic distribution of the matrix A is assumed,
; with the vector b mapped vertically to column 0.
; The result is left along column 0.
;
(define n 1024)        ; 1024x1024 input array
(define chunk (* 2 (/ n *xsize*))) ; words per node
(loop 'elim n          ; Gaussian elimination phase
  (subset cols         ; get maxloc with value
    (allreduce 'r0 'maxloc_func :m 3))
  (subset rows         ; broadcast maxloc and scale
    (broadcast 'b1 (runtime) :m 3))
  (subset cols         ; swap row i, bcast pivot
    (stream 's2 (runtime) (runtime) :m chunk)
    (broadcast 'b3 (runtime) :m chunk)))
(loop 'subst n         ; back substitution phase
  (subset rows         ; send col i to col zero
    (stream 's4 (runtime) 0 :m chunk))
  (subset (list (car cols))  ; bcast X(i) to col 0
    (broadcast 'b5 (runtime) :m 2)))
```

Figure 2-7: COP code for Gaussian elimination

2.10.1 Message Streaming

In COP, the message has been tied to the operator, and the operator consumes and returns only complete messages to the application. Some applications may find this unnecessarily restrictive; for example, a streaming function may wish to take individual words from the interface, perform some operation, and write a single word back out.

Fortunately, this ability can be specified with no loss of generality by specifying '`:m 1`' on a scheduled routing substrate. However, using a dynamic routing substrate, this specification would result in many one- or two-word messages, perhaps doubling the load on the network as a result of the per-message headers. For dynamic routing, a fairer scheme would involve larger messages, and a COP interface that allowed the header and the message body to be read separately. This style of interface has not currently been implemented, since the focus here is on scheduled routing, where the existing interface is adequate.

Complicating the picture further is the fact that some operators, such as `cshift`, require the implementation to read and write the message in an interleaved fashion. A read-header/read-body interface would have to be carefully written, by, *e.g.*, providing a layer of buffering between the interface register and the application to handle reading data during a write-data-word function. However, this limitation is not absolute for a communications language, and simply requires some extra semantics to be defined for the language.

2.10.2 Exposing I/O Wait Time

A related, and more fundamental, drawback with the current compiler is that it busy-waits for all I/O to complete. There are several things a more integrated compiler could choose to do with that time:

- Interleave multiple communications operators' I/O. For example, if the application wants to perform a shift in one direction at the same time as a broadcast in the other, it may be better to interleave the reads and writes to match the scheduled patterns in the router. This could substantially reduce I/O time when multiple operators are to be used at effectively the same time.

- More generally, communication can be interleaved with computation. If the HLL compiler determines that a given operator has 1/3 of the bandwidth from the node, it can schedule two routing-cycles' worth of computation between each read or write to the processor interface, thus reducing the overall application time.

The number of cycles between each slot where the network can accept a data word is known as the *gap* [65, 49]. Before attempting to optimize this number to a lower level, it is important to make sure that no other bottleneck exists: in particular, the processor may be unable to inject messages faster than one cycle in two (as is the case for the current NuMesh/SPARC prototype). In this case, the I/O wait time will generally be less of an issue.

Split-language techniques do exist for hiding I/O wait time. Interestingly, interleaved I/O is already used for the implementation of NuMesh online routing, as discussed in Section 4.2. A similar technique could be used for scheduled operators: the write function could place a message buffer in a queue for that operator and set up an interrupt handler which would send the message.

2.10.3 Hidden State

The language currently requires some state to be defined for many operator implementations. For example, a stream with a run-time destination is defined such that when the destination and the source are the same node, the implementation sets a run-time flag, copies the write function argument to a buffer, then later copies it back to the read function's buffer. Such copying and state would be unnecessary if a single compiler were generating both computation and communication code. (Although, of course, the HLL compiler is free to avoid the calls to the write and read function altogether if it determines that the source and destination are the same.)

An attempt is made in the implementation to expose some of this to the compiler in the returned implementation functions; for example, with the current compiler, the setting and testing of the local flag is visible to the HLL compiler, and the test of the flag can be elided. However, the semantics of C are such that an extra copy to the local-destination buffer are unavoidable.

A similar example occurs with the parallel prefix (scan) and reduction operators, which are passed a function pointer to be used on the operator arguments. In an integrated compiler, the user's function would be applied directly to the arguments. Again, some attempt is made in the current compiler to expose this mechanism; if the reduction function is defined 'inline' before the COP functions are processed, the user's function will be integrated into the operator function. However, more sophisticated operations are still difficult to express; using a streaming-message interface, a long-word sum-reduction might want to use carry-propagating add instructions, and access to such primitives would be difficult to guarantee in the COP environment.

2.10.4 Loop and Branch Prediction

The HLL compiler may be able to accurately predict communications operands one or more loop iterations (or branches) ahead. In this case, it may be possible to take advantage of this knowledge to load the router with the appropriate code. For example, in a loop with one runtime operator, where DMA is available, and the runtime operator's argument can be predicted a cycle ahead, the load of the next phase of the runtime operator could be started at the previous load, using the predicted value of the runtime operator. The split-language semantics do not allow this, since there is no way to express predicted future values of runtime operators.

It would be possible to express this information by defining syntax to express a load value for the *next* loop, not the current loop. This is, however, a step further away from the current clean interface, and a step closer to the integrated compiler solution.

2.10.5 Schedule Generation

The language as currently defined does not allow for the HLL compiler to request any particular clustering of scheduled communications slots in the schedule. Instead, the stream router is allowed to schedule the appropriate amount of bandwidth as it chooses. Allowing for such clustering would mean that the application could send an entire message in a short space of time, though perhaps with a longer synchronization time at the start of the message before starting the I/O. The current compiler does not attempt to control stream scheduling, though it would be easy to extend it to request that messages of up to a certain size be allocated in single chunks of successive schedule slots.

More significantly, if the application has a known pattern of communications (say, a systolic matrix-multiply), it could be beneficial to specify the preferred order of operator scheduling at the interface, such that the application could do a sequence of read, read, compute, and write without blocking on the schedule for any operation. Currently, all the HLL compiler can do is find the critical inner loop, determine its timing, then specify a schedule length for the communications.

2.10.6 Message Passing or Shared Memory?

One final point is worth emphasizing as a strict matter of language style. The current draft of the language is targeted primarily towards operations where all involved nodes are active participants. At first glance, this seems to be in contrast to, for example, the shared-memory model, which uses a request-reply structure for all communications.

However, the COP language is not about programming styles so much as communication. A simple shared-memory implementation with COP might just consist of a single operator per node:

```
(parallel
  (stream 's0 0 (runtime :dynamic t))
  (stream 's1 1 (runtime :dynamic t))
  ...)
```

This example uses the `:dynamic` optional argument to `(runtime)`, which specifies that a particular node may call the *load* function multiple times with new values of the specified operand, without assuming that other nodes will do likewise. Currently dynamic operands are only legal in a few places, such as runtime `stream` destinations.

A shared-memory handler could then be built on top of this by the application, using interrupts to access the memory and return data to the requesting node (similar arguments could be made, for example, for Active Messages [66]). Where COP becomes useful is when the compiler can extract phase and communication information from the application (*e.g.*, when a given section of the application performs a transpose using nested `for` loops), and convert it into COP operators. These operators may still be hooked up to a shared memory interrupt handler, but the communication will happen faster once the underlying communication patterns are extracted and given to COP by the frontend compiler.

2.11 Summary

This section has presented the outline of the communications language presented in this thesis. The language includes many features that are necessary to support efficient compilation to a reprogrammable scheduled router:

- high-level operators to capture useful semantics;
- grouping constructs to capture known timing relationship among operators, and facilitate creation of communication phases;
- a pseudo-runtime-value construct that indicates data-dependent values and can express known constraints for those values;
- ready extensibility;
- and portability to a range of routing substrates.

The limitations of the abstraction, while clear, do not seem sufficient to prevent the language from being used to generate efficient communications code in many cases; and they do not detract from the language's utility as a research vehicle.

Chapter 3

Related Work

This section provides a brief overview of some of the related work in the literature. It is divided into sections on communication languages, scheduled routing architectures, and scheduled communication in general.

3.1 Communication Languages

There have been a wide variety of projects examining communication languages and systems. This section attempts to place them in context with COP and this thesis. For each language, the features it supports in comparison to COP are mentioned, and (for the more relevant projects), some examples and additional details are provided.

3.1.1 The Gabriel Project

Digital signal processing (DSP) applications have been using 'communication languages' of various kinds for some time. A good example of this is the Gabriel project [8, 27], now the Ptolemy project [12, 13]. Their system uses block diagrams and a flexible signal-flow model to support multirate and asynchronous systems. It uses a *synchronous dataflow*, or *Sdf* model for DSP applications. In the Sdf model, each processing object in the application has a fixed connection to other such objects, and produces or consumes a fixed number of tokens each time it fires.

While Gabriel is well-suited to DSP applications, it lacks the generality required to be a general-purpose communications language such as COP. It does not specify communication phases, since its main target is signal-processing applications which run continuous pipelined processing. For the same reason, it does not specify any data-dependent communications.

Communications are represented by the connections between the processing objects, which are termed *stars*. The communication model annotates links with information on message size or message count in a manner similar to COP. The Gabriel compiler then uses these annotations to determine how to execute the stars. Stars may be executed all together on a single node, or may be distributed across multiple nodes.

Figure 3-1 shows an example of a Gabriel star. Notice that the communication characteristics are specified by endpoints at the star level, rather than at the 'operator' level as is done in

```
(defstar fft
    (descriptor "Computes the FFT of the input.")
    (param order 128)
    (input in (no_samples_used order))
    (output out (no_samples_made (* 2 order)))
    (state twiddle-factors nil)
    (init compute-twiddle-factors)
    (function fft))
```

Figure 3-1: Gabriel code for an FFT star

COP. It is only when stars' endpoints are hooked together that particular communications are instantiated. The example shown is an FFT; it specifies a parameter for the size of the FFT, and gives a default value (128). The inputs and outputs are specified as indivisible messages of size *order* and $2 \times order$. Internal state, an initialization function, and the main processing function are all named at the end.

3.1.2 OREGAMI/LaRCS

One language close in spirit to COP is LaRCS [42], the language component of the University of Oregon's OREGAMI system [43, 44]. The OREGAMI system is a set of software tools for automatic mapping of parallel computations to parallel architectures. LaRCS (the Language for Regular Communication Structures) is a description language in the spirit of COP. LaRCS is used by the OREGAMI mapping algorithms, as well as to route communications in the network topology.

Like Gabriel, LaRCS allows the specification of communications using simple, compile-time connections. However, unlike Gabriel, it also supports a notion of communication phases. LaRCS has no support for high-level communication operators, nor for run-time values or operator annotations. This is a fundamental lack in the language, perhaps caused by LaRCS' origin in graph description languages, which typically describe static communication structures. Accordingly, LaRCS cannot represent communications for applications with data-dependent communications. Nonetheless, LaRCS has been used successfully for an interesting set of applications.

The language supports multiple communication phases, but the phases must be specified by the user, rather than derived from relationships among the operators based on message traffic. Continuing operators and multiple threads of control can be specified syntactically, but it is unclear from the published work whether they are supported.

LaRCS programs consist of three components: node declarations, which describe the processes; `comtype` and `comphase` declarations, which are essentially functions that can be used to specify communications; and a *phase expression*, which specifies the the entire program's behavior in a phase-by-phase manner.

A sample LaRCS program is given in Figure 3-2. It describes an algorithm for simulating an n-body problem. The nodes are (virtually) arranged in a ring, and the n^2 interactions are computed by passing location information half way around the ring (thus doing $n^2/2$ computation), then exchanging data across the ring to provide the remaining information to the opposite node.

```
nbody(n,s);
attributes nodesymmetric;
nodetype body
   labels 0..(n-1);
comtype ring_edge(i) body(i) => body((i+1) mod n);
comtype chordal_edge(i) body(i) => body((i+(n+1)/2) mod n);
comphase ring
    forall i in 0..(n-1) {ring_edge(i)};
comphase chordal
    forall i in 0..(n-1) {chordal_edge(i);}
phase_expr
    ((ring |> compute)**(n-1)/2 |> chordal |> compute)**s;
```

Figure 3-2: LaRCS code for the n-body problem

The `phase_expr` provides the high-level picture of the n-body problem in the example. The inner loop consists of a `ring` communications phase, then computation; the |> construct specifies that these happen sequentially. The inner loop is repeated $(n-1)/2$ times, as indicated by the `**` construct; it is then followed by a `chordal` communications phase and further computation. The entire process is then repeated s times, where s is a parameter supplied to the LaRCS program. The || construct, not used in the example, indicates parallel execution of communication phases.

COP does not explicitly provide parameterized primitives in this way, but it includes the same functionality by virtue of being a Lisp-based language: the application can define suitable `ring_edge` and `chordal_edge` functions in Lisp, then create the desired phase information by using Lisp's basic control primitives, or one of COP's iterating constructs (such as `all`).

3.1.3 The PCS Tool Chain

A somewhat similar approach is used in the Programmed Communication Service tool chain [28], developed as part of the iWarp effort. The authors share the opinion of the NuMesh project, that long-term communication connections can both be beneficial to performance as well as prove usable to programmers. PCS provides a semantics roughly equivalent to LaRCS, but its use with scheduled-routing hardware makes it more directly equivalent to COP. The PCS information is used to create scheduled paths within the hardware for faster communications at run-time.

They use an *array program* as their communication language. It defines connections between nodes and mappings from node programs to nodes. Figure 3-3 includes a sample array program. The array program sets up communication links at compile time, and the nodes' runtime program code uses those links. Ports are named with strings local to each node, and are converted to physical ports by runtime function calls. PCS allows for users to specify either, both, or neither of the communication paths and node placements. If the routing is not specified, it currently (as of the time of writing) defaults to a simple row-then-column technique to generate scheduled routes.

Array program code (compile-time communications):

```
outport = create_port(node(0, 0), "out");
inport = create_port(node(0, 1), "in");
nw = create_network(outport, inport);
```

Node program code (runtime computation code):

```
pcs_init();
if (self == node(0, 0)) {
     port = get_port("out");
     send_msg(port, data);
}
if (self == node(0, 1)) {
     port = get_port("in");
     recv_msg(port, data);
}
```

Figure 3-3: A simple PCS program

PCS supports a notion of phases similar to that of COP. A number of precompiled phases can be loaded at runtime, and the `set_phase()` function can be called to switch among them. As is true for COP (see Section 5.2), changing phases is not a trivial matter. PCS takes the relatively simple approach of performing a barrier before every phase switch to ensure that the network configuration is consistent across the system. While their motivation for phases was primarily to work around the limited number of communication streams in iWarp, they acknowledge that varying bandwidths on streams may also be advantageous as part of changing phases. As for LaRCS, PCS phases are fixed by the user rather than derived by the compiler.

The node and array programs are composed hierarchically, in a manner similar to Gabriel; this allows nodes to be hierarchically grouped together into different tasks in different phases. An additional notion of an *overlay* composition model allows PCS to achieve some of the same goals as COP's operator grouping hierarchy. With the overlay model, PCS allows the mesh to be subdivided along different hierarchical lines in each phase, thus adding a degree of flexibility to the phases.

3.1.4 The CrOS Project

The CrOS project from CalTech [24] uses a *crystalline* programming framework. The authors present an MIMD model with nodes calling library functions to communicate with each other. The CrOS target hardware did not have a separate router, and accordingly the processors themselves performed the communication over dedicated point-to-point links. Their *loosely synchronous* model requires that a node cannot write to another node unless the destination node is prepared to receive the message; similar in spirit, at the processor level, to scheduled routing. Since CrOS was purely run-time, they did not support any notion of phases, or any distinction between compile-time and run-time values.

```
int int_add(int *ptr1, int *ptr2, int size)
{
  *ptr1 += *ptr2;
  return 0;
}

void main(void)
{
  ...
  sum = procnum * procnum;
  status = vm_combine(&sum, int_add, sizeof(int), 1);
  ...
}
```

Figure 3-4: CrOS code for a parallel prefix

However, CrOS does include a notion of *operators* similar to that used in COP. In addition to simple stream-style connections, they include primitives for doing transpose and various other array-style communications. A fragment of a simple sum-of-squares program is presented in Figure 3-4; the `vm_combine()` function is a CrOS operator that does a parallel prefix, or scan, across the nodes in the machine.

CrOS supports a version of online routing called the `crystal_router`, which handles online routing in a manner similar to that used by COP. Without the ability to extract information at compile-time, however, they do an 'inspector'-style step where routing information is collected in the first phase, then subsequent phases use the collected information to avoid having to examine all possible streams for input.

3.1.5 Chaos and PARTI

The Chaos [55, 30] and PARTI [64] work from the University of Maryland includes some of the same conceptual framework that COP does. The goal is to support irregular distributions of data in applications, unlike the standard block and cyclic distributions applied to regular data. Instead, an indirection array is used to tell each processor where to find the data that it needs. Partitioning algorithms are used to place the data, and (on the loop index space) to place the computations. Periodically, if distributions change, the Chaos library can recompute the irregular distribution. It supports communications optimization by grouping together all I/O needed between nodes at the beginning and end of each iteration in an automatic fashion. This use of run-time communications configuration is reminiscent of COP, and is in fact directly usable with COP; the "inspector" that determines communications could generate COP code, and the application could pass the data to a runtime-linked stream router to generate communications schedules.

3.1.6 Other Communication Languages

A wide variety of communication languages exist, and they will only be touched on briefly here. While not all of the languages listed below fully meet the two criteria for a communications language mentioned in Section 2, they all have, at least, some characteristics of such a language.

- The Conic distributed programming environment [48] is a Pascal-based language with ports defined in each module, a uni-directional `link` command, and a `broadcast` primitive. This is fairly restricted, since it neither captures multiple phases of communication nor allows a way of specifying stream usage information.

- The Prep-P automatic mapping system [6] includes a graph description language GDL, closely tied to their programming language XX. Its capabilities are similar to those of Conic.

- The Assign parallel program generator [53] handles partitioning, mapping and routing for static coarse-grained flowgraphs, similar to those used for Gabriel.

- GARP (graph abstractions for concurrent programming [33]) is a graph-rewriting language representing connections among agents. Communications occur only along graph edges. The language, while elegant, lacks notions of communication phases, runtime streams, and stream usage specifiers.

- Lee and Bier [39] discuss mapping a limited class of dataflow program graphs to a scheduled routing environment. They include support for a synchronization primitive to allow changing phases during execution of the dataflow graph.

- Express [23] and PVM [62] both provide some notion of connections, but they are only set up at runtime, and provide a layer of buffering that reduces performance. Their library primitives provide similar functionality to COP's operators.

There are a wide variety of languages designed primarily to express computation, rather than communication, but that include communication in the language. Such languages are largely outside the realm of related work, since they meet neither of the two criteria for communications languages. Languages in this category include, for example, Linda [14], Split-C [16], and CSP [29].

3.2 Scheduled Routing Architectures

There are a number of existing scheduled-routing architectures discussed in the literature. Two of them are presented here, iWarp and GF11.

3.2.1 iWarp

The iWarp architecture [54, 10, 11] is CMU and Intel's follow-on project to the Warp architecture. iWarp integrates the processing and routing units on a single chip, targeted to DSP, scientific, and image processing. The interface between the processor and router is an interface register file used for systolic communication, similar to the processor interface described in this thesis for scheduled routing. Memory-to-memory communication is also supported.

iWarp supports both traditional dynamic routing as well as a higher-speed *virtual channel* technique, where channels are created for long periods of time and used to connect two nodes that may not be adjacent in the mesh. iWarp nodes have a 20×20 communications crossbar used to route the channels in the mesh; virtual channels are pre-connected to an output channel, and dynamic messages use the usual contention mechanisms to bind to an output channel going the direction the header indicates they should go.

A virtual channel is created at runtime by sending a marker message which instructs each router as to whether the channel should turn in a given direction, or continue straight through (by default). Setting up the virtual channel is relatively slow; each node in the pathway takes four or five clock cycles to forward the setup token to the next node. However, once in place, messages can be transmitted over the channel at the full iWarp interconnect speed. Dynamic messages are treated similarly, but, essentially, tear down their channel behind them as they go.

iWarp channels include a limited 'reprogrammable' aspect. Messages can be read into a node, then the router modified to forward the remainder of the message onto subsequent nodes. Similarly, a node can write out a message, then strip the header from an incoming message so as to leave the two messages merged as one.

Systolic communication was found to perform very well on applications for which it could be used [25]. In general, applications performed better when more systolic paths were established, since communication latencies were lower. iWarp's system support includes a number of compilers for C and FORTRAN, as well as compilers for image processing and the like.

Overall, iWarp is the existing system closest to the reprogrammable scheduled routing system targeted by COP in this thesis, and their results were very promising. NuMesh extends upon their work in several ways, as discussed more extensively in [56]; in particular, NuMesh allows faster communications, more precise control over stream bandwidths, and faster phase changing and other router modifications.

3.2.2 GF11

The GF11 parallel computer from Yorktown [5] is a SIMD architecture with 576 processors and an unusual interconnect. The processors are connected through a three-stage Beneš network, capable of supporting any permutation of the processors, and reconfigurable among 1024 distinct configurations in a single cycle. Its main application is the numerical evaluation of some of the predictions of quantum chromodynamics.

The architecture has two key similarities to a reprogrammable scheduled router: it supports a notion of multiple phases of communication at the router level, and it allows for rapid transitions among phases. A COP backend for the GF11 would capitalize on the fast context-switch time as well as the large number of phases that could be held in hardware.

Each stage of the network consists of 24 24-input crossbar switches, with the middle stage connected to the outer stages with a perfect-shuffle fixed interconnection. By modifying the state of the crossbars any permutation can be supported, as well as other interesting communication operators such as broadcast. Configurations can be loaded into each crossbar for use on demand; a 4-neighbor torus, for example, would require four configurations, one for each direction. Configurations typically take some time to compute, and are loaded as part of the program startup; once loaded, the frontend processor can switch configurations on every cycle.

Their architecture, reported in 1985, was able to outperform a Cray 1 by a factor of approximately 100 on their target application. They suggest that similar performance might be obtainable for a variety of scientific and engineering applications other than chromodynamics.

3.3 Scheduled Communication

This thesis does not directly address how to schedule communication in a single phase, relying instead on a separate stream router to handle the allocation of routing resources for streams. However, to give the reader a sense of how scheduled routing is carried out at the low level, a brief description of some of the relevant work is presented here.

3.3.1 Agrawal-Shukla

Agrawal and Shukla at NCSU did a good initial analysis and simulation of scheduled routing for arbitrary compile-time streams [60]. Their work focussed on the notion of *output consistency*, a definition for pipelined computation that requires that data always be received by a node by the time the node is ready to consume it. They contrast traditional dynamic routing, observing that contention may stall messages beyond the time when they are needed by their destination. In particular, they observe that dynamic routing has no sense of changing message priorities: a message from a subsequent iteration of an application might arrive first at a switch and lock out an urgently-needed message from the current iteration of the application.

Their model uses a task flow graph (TFG), a directed acyclic graph whose vertices represent tasks and whose directed edges represent messages between tasks. TFG executions are overlapped as successive problems arrive to be solved. Their algorithm computes schedules for each router by first computing an optimal path assignment for all the messages, then computing which portions of each message are transmitted at what times. Together, this allows the creation of a schedule for each node to transmit the messages.

They simulated their work, comparing it to traditional dynamic routing on both tori and hypercubes. They found that their scheme was able to guarantee consistent message arrival times under most loads, whereas dynamic routing incurred frequent output inconsistency, causing stalling of the entire application.

3.3.2 Bianchini-Shen

Bianchini and Shen [7] presented a methodology for generating routing schedules for static problems, using an optimal polynomial-time algorithm. They examine 'traffic compilation' once data operators are mapped onto nodes, optimizing for overall bandwidth in the system.

Their network traffic scheduler iterates through the following four steps. A *saturated* link is one that has been allocated 100% of the traffic it can support.

1. An initial schedule is presented (possibly using a simple shortest path routing scheme).

2. Traffic volumes are simultaneously and proportionally increased until one or more links are saturated.

3. A new schedule is presented, by rerouting traffic from saturated to unsaturated links.

4. The previous two steps are iterated until an optimal traffic schedule results.

The traffic-smoothing technique they present is a version of multi-commodity flow, much discussed in the literature [41, 34, 35, 40]. The general form of the problem is to model a network graph as fluid pipes, where distinct communication source/destination pairs correspond to distinct commodities that must push through the pipes to reach their destination. An optimal schedule has been found when a *cutset* of saturated links is produced, *i.e.* a set of links that disconnect the graph when removed from it. A linear programming formulation can be shown for this problem, but can not be rapidly solved.

Bianchini and Shen present a *direct scheduler* solution to the problem that is polynomial in the number of links. They find the minimum spanning tree for the graph; then, for each node at one end of a saturated link not on the spanning tree, they check how the traffic is routed. If it is routed onto the saturated link rather than via the spanning tree, a reroute is possible to improve the overall link usage. Each phase of the algorithm takes only $O(L \log L)$ steps, resulting in a rapid solution of the routing problem.

3.3.3 Other Scheduling Work

This section briefly mentions a few other papers that have dealt with the subject of scheduled routing.

- E. D. Dahl showed an algorithm for scheduled routing on the CM-2 using a swap primitive [17]. His algorithm does no backtracking to figure the schedule and assumes unlimited buffering for data in the nodes.

- Bollinger and Midkiff presented a simulated annealing algorithm for link assignment [9]. Their algorithm essentially precomputes adaptive routes but assumes online routing (presumably non-adaptive) at runtime.

- Shin and Kandlur present a virtual cut-through route assignment algorithm [32]. As for Bollinger and Midkiff, they attempt to precompute adaptive routes for a known traffic distribution. The algorithm used is an incremental greedy path router.

Chapter 4

Managing Data Dependency

This chapter describes the techniques used to manage data dependency in a reprogrammable scheduled router. Section 4.1 examines the variety of ways that COP can implement an operator, focusing on solutions for data dependency; Section 4.2 looks in more detail at one particular technique, namely online routing on a scheduled-routing substrate.

4.1 Operator Implementations

This section examines the variety of techniques that can be used to implement operators. In Section 4.1.1 a set of techniques suitable for supporting data-dependent routing on a reprogrammable scheduled router is proposed; Section 4.1.2 looks at the different high-level ways that an implementation can be written to support a given operator's functionality.

The basic theory of supporting multiple implementations for an operator type is that it allows the compiler to pick the best one. This is particularly useful with a scheduled-routing substrate, but even with a dynamic-routing substrate, there are different ways of implementing the same functionality. The information provided by COP (such as message size, message count, and source and destination) allows the best implementation to be chosen. For example, the `broadcast` operator currently has three dynamic-routing implementations. One simple-minded implementation just writes the message sequentially to each destination; it is slow, but works with a run-time source and an arbitrary subset of the mesh. Another implementation requires a compile-time source and creates a broadcast tree in the mesh to forward the data. The third allows a run-time source, but requires a closely-packed subset; it propagates the data somewhat more slowly by sending it to all its neighbors in the mesh in a dimension-ordered flood fill. Depending on the mesh size, message size, and so forth, different implementations may prove to be most efficient in different contexts.

4.1.1 Approaches to Data Dependency

The most interesting aspect of implementations for scheduled routing is coming up with techniques to handle run-time values. This section outlines some implementation techniques for doing so.

If an application communicates to some remote node, but it is not known which at compile time, a stream can't simply be scheduled at compile time. However, there are a variety of work-arounds that can be invoked, depending on the degree of information present about the distribution of the possible destinations, the amount of data to be carried by the stream, and the size of the mesh.

An operator can be implemented in one of a number of different ways depending on the anticipated bandwidth and latency demands of the operator as compared with what the different implementations can provide. For example, if a run-time operator will transfer only a few words, the run-time system should not waste a great deal of time online generating a schedule, but should instead try to route the data quickly using online routing. The remainder of this section presents a spectrum of high-level alternatives for supporting data-dependent routing. (Alternatives not implemented in the current compiler are marked with a dagger, '†'.)

Pure Compile-Time

The far end of the spectrum is when the operator arguments are all known at compile time. In this case an efficient communications schedule can always be generated. Venturing further from this end of the spectrum incurs a price either in latency or bandwidth or both to route the data. Examples of this are many: to name a few that have already presented elsewhere, both transpose and bitreverse are simply sets of compile-time streams, as is the 'shift' phase of the matrix multiplication algorithm shown in Figure 2-6 on p. 38. Returning to the common broadcast example, a pure compile-time implementation of broadcast is simply a mesh-level multicast for a scheduled routing system.

Multiple Path Allocation†

The easiest way to handle data dependency if there is a small number of possible destinations is to schedule all of the paths and use the appropriate one at run time. This is particularly true if the amount of data to be transferred is small, since allocating multiple paths results in no extra message startup overhead time, unlike most of the schemes described below. This scheme rapidly becomes impractical as the the number of destinations grow and the amount of data to be transferred increases; in the limit, for example, to use this scheme for a random linear permutation of size n results in streams running as slow as $1/n$ words/cycle, and schedules of length $\Theta(n)$. For relatively small meshes, this provides a low-overhead, general-purpose solution to run-time operators. For example, for a two-node mesh with a run-time broadcast, the compiler might simply schedule one stream going each way, each one a separate compile-time 'broadcast'.

Output Discarding

A related but less bandwidth-intensive technique is to implement multiple destinations as a multicast tree (or, similarly, multiple sources as a reduction-style tree). Then, when determining where to send data, the destinations that are uninterested simply rewrite their node to discard the data instead of delivering it to the router. On a scheduled-routing mesh, this technique actually consumes *less* bandwidth than multiple path allocation, since a spanning tree is used with a constant bandwidth component across the entire tree, unlike the above solution. For example, this technique can be used to convert a compile-time broadcast to a stream with a compile-time source and a run-time destination; nodes other than the destination simply discard the broadcast within the router itself.

Dynamic Destination Selection†

A dynamic-destination stream operator may have multiple possible destinations, but all of the destinations lie close together. In this case, the problem may be best handled by allocating a single 'data stream' to carry the data across the mesh to where the cluster of destinations is, and only then split the message off to where it needs to go. A 'control stream' is set up so that at load time it carries a control word to the remote router that splits the data stream; the control words reprograms the router to point to the desired final destination. The data can then be sent at full bandwidth to the desired node, starting only a few cycles behind the control word. Multiple decision points can also be used, with each controlled by a separate control stream.

Multiple Schedules†

A different approach, when the number of run-time possibilities is relatively small, is to simply select an appropriate schedule based on the run-time arguments, load that version of the schedule on all the nodes, and then run at full bandwidth. This approach requires that all the nodes involved in the communication know which schedule to load. For example, in a broadcast, if all the nodes know who is broadcasting they can choose the appropriate piece of broadcast schedule to run: either a schedule that generates and forwards the data, or a schedule that expects to read the data from one direction and forward it to the appropriate onward directions. Since each schedule typically only requires a few hundred bytes, a fair number of schedules can be loaded into the processor to be selected at run time. If all the variant schedules can fit simultaneously in the router, furthermore, the run-time decision may simply consist of a comparison or two and a single schedule-change command to the router.

Run-Time Specification

A slightly more complex alternative is to generate a fairly generic schedule which can be customized at run time. If the generic schedule can be expressed as routable streams, the operator can be scheduled in parallel with other operators, since the customization can be applied to the VFSMs that make up a specific stream after stream routing has been performed. For example, a linear broadcast from a runtime source could be implemented by allocating two paths going through the mesh; these paths can be scheduled in parallel with any other paths. Then, the node that is doing the broadcast rewrites each stream to read data from the processor rather than the neighbor node; now, the broadcast node can write to the two streams and reach all the other nodes. (Of course, for subsequent broadcasts from different nodes those streams would need to be reset to their original values again.) This technique requires $2\times$ the bandwidth of a compile-time broadcast.

Run-Time Operator-Specific Scheduling†

If the operator needs to transfer a substantial amount of data, an entire schedule can be generated using operator-specific code, tuned to produce an efficient schedule using the run-time data that is available. In the simplest case, each node computes the same problem (*e.g.*, the scheduling problem for the whole mesh for a given permutation) and then puts the relevant part of the solution into its own router. More sophisticated techniques might involve solving (for this example) the permutation problem in parallel—using a built-in schedule for the routers during the parallel schedule generation—then switching to the resulting schedule to perform the permutation. Since the startup time for this technique is high, however, the compiler must ascertain that the amount of data to be transferred is large enough to amortize the startup schedule computation. This technique also tends not to be suitable in cases where multiple operators are running in parallel, since operator-specific code typically assumes it can use all the available schedule slots.

General Run-Time Scheduling†

If the operator has large data-transfer needs, but is merged with other operators, part of the schedule compiler can be included in the run-time image, and allow the nodes to generate their own schedules before they begin to use them. This schedule generation may be constrained by fixed schedule slots taken by continuing operators already scheduled in an earlier phase, but is otherwise the same problem that the compiler normally solves for continuing operators with compile-time arguments. A more ambitious solution might include incorporating parallel algorithms for computing stream allocations at run time.

Processor Intervention

If data-dependent decisions must be made, sometimes the easiest place to do so is in the processor itself. In this case, streams can be routed from one decision-point to another, with the processor involved in deciding which output streams to forward the input data to. As an example, a run-time broadcast can be performed by using a 'flood fill' technique. Nearest-neighbor streams are scheduled between all the nodes involved in the broadcast. For each broadcast, since all the nodes know which node is broadcasting, each node reads an appropriate stream and then forwards the data to the appropriate output stream(s). Controlling which dimensions forward the data to other dimensions yields a simple flood-fill algorithm for broadcast.

Online Routing

The ultimate fallback for all operators is online routing; this is effectively a generalized version of processor intervention. As discussed in Section 4.2, dynamic routing can be used to move all the data. Sometimes the compiler may know that the dynamic routing is constrained to one or two dimensions; in this case, it can construct code that does not check the unused dimensions and thus can run more rapidly. Sometimes (*e.g.*, for very short messages, or with a wide range of possible run-time values), the higher startup latency of online schedule computation can make dynamic routing be, in fact, the faster solution. General-purpose dynamic routing is handled by routing the message in progressive hops; after each hop, the processor code ascertains which stream to forward the message on for the next hop. Hops can be a single node or multiple nodes; the online routing decisions can be made either with the processor (for current implementations), or by an online routing module attached to the router (if cost/benefit considerations make it useful in the future).

4.1.2 Implementation Types

This section covers the several high-level ways that an implementation can be written for scheduled routing to provide the necessary functionality for an operator, and provides some insight into how the compiler can choose among them.

For primitive operator implementations (as opposed to those that are simply Lisp code), the implementation's interface to the compiler is specified entirely through a single structure that includes a set of functions for characterizing and including the implementation. Table 4.1 presents the important functions that make up the structure (also included in the structure, but not shown, are miscellaneous flags fields and some other book-keeping data).

Name	Description
metric	suitability of implementation to operator
replace	return replacement operators (optional)
streams	return streams to route (optional)
gen_router	set up schedules and VFSMs (optional)
gen_io	emit I/O functions (read/write or func)
gen_load	emit load-time startup code (optional)

Table 4.1: Interface functions for operator implementations

The one function in the table common to all implementations is the *metric* function. This function is called from the compiler with a pointer to an operator and a by-reference structure to be filled in with information on how this implementation will perform on the given operator. A metric function can simply return 'false' if it can't handle the given operator; this may be for a reason as simple as that it is the wrong type of operator, or that the implementation can't handle run-time operand values, or for more complex reasons (*e.g.*, a dense broadcast implementation would reject an operator with a sparse subset).

If the implementation can handle the given operator, it fills in the 'implementation information' structure. This structure holds best guesses as to the time to load the operator; the time to reload it on subsequent passes if one argument or the other has changed; the startup time for a message; and the inter-word time for a message. It also holds the number of I/O words required to run to completion, the number of VFSMs predicted to be used in each pipeline, the number of interface registers, and some other flags. All this information is made available to the compiler's search engine for trying to determine the best implementations to pick.

The following subsections discuss the four main high-level techniques for implementing operators: as dynamic-routing, stream-alias, schedule-generator, or replacement implementations.

Dynamic-Routing Implementations

Dynamic-routing implementations correspond to normal primitive implementations on a dynamic router. Such an implementation defines a *streams* function in its interface, which when passed the operator returns an array of streams necessary to handle the communications for this implementation of that operator. For example, a simple `stream` operator with just one instance would return the single appropriate stream from this operator; a `reduce` operator running in parallel on n rows of an array would return n reduction trees' worth of streams. For a dynamic implementation, the streams are used to guess traffic rates, as well as for dependency analysis.

The streams in a dynamic implementation will be single-source, single-destination streams; however, the sources and destinations may be specified either as compile-time or as run-time with a range of possible values. (A generic 'value' structure is used within the compiler to represent both compile-time and run-time values.)

Dynamic implementations can be used for a dynamic-router substrate, unlike stream-alias or schedule-generator implementations, which are specific to scheduled routing. They can also be used with scheduled routing, however, at a performance penalty; this is discussed in detail in Section 4.2.

Stream-Alias Implementations

The basic implementation type for scheduled routing is the *stream-alias* implementation. It also uses the *streams* function; the returned streams are used by the scheduled-routing backend to route actual streams, as well as to perform some dependency analysis. The streams are single-source but may be multiple-destination, implemented as a multicast in the router; however, the endpoints must be specified at compile-time so they can be routed at compile-time.

If the implementation needs to edit the schedule generated by its streams, it may include a *gen_router* function as part of its definition. This function is used to specify router-level information, above and beyond that given by the streams returned by the *streams* function. For a stream-alias implementation, a gen_router function typically isolates the portions of the router used by a given operator's streams, and modifies the router information (*e.g.*, VFSM sources and destinations) as necessary. For example, a stream with a run-time source can be implemented with a tree of point-to-point streams converging on the compile-time destination. Once the streams are routed, the gen_router function can update the router information such that data is forwarded automatically from one stream to the next, taking data to the destination without requiring intervention from the processors.

In addition to this compile-time modification, a stream-alias implementation may be run-time modifiable. This is a powerful technique, since a stream that is modified at run-time can be routed along with streams from other operators in a single phase, but such a stream is more powerful than a simple point-to-point stream. These operators rely on the *gen_load* function specified in the implementation definition. This function supplies additional code to be executed by the *load* function returned by COP as part of the per-operator code. For fully compile-time operators no such function is specified, since no load-time changes to the router are necessary. The 'Run-Time Specification' example of a run-time broadcast in Section 4.1.1 is an example of load-time modified stream-alias implementations: the gen_load function handles modifying the router to accept data from the local processor, or to forward data from its neighbors.

Schedule-Generator Implementations

The other fundamental implementation type for scheduled routing is the *schedule-generator* implementation. The implementation has no `streams` function, and instead relies solely on the *gen_router* function to generate the necessary VFSMs and schedules from scratch. These implementations cannot be scheduled in parallel with other implementations, since each schedule-generator implementation assumes it has complete control over the schedule of all the nodes in its phase.

Schedule-generators are useful for a variety of reasons.

- **Speed**. While a stream-alias implementation has to depend on the whims of the stream-routing code to get good performance, a schedule generator can include hand-optimized code to move data as fast as possible through the mesh.
- **Power**. A schedule generator can use every aspect of the underlying router's capabilities, rather than trying to express the desired functionality using the streams abstraction. For example, the `sch_prefix` implementation sets up two tightly-coupled *subphases* where nodes communicate with neighbors in different subphases and switch between them as the prefix runs.
- **Run-time**. A schedule generator can generate arbitrary schedule code at run time. Thus (as discussed above for 'Run-Time Operator-Specific Scheduling'), a run-time permutation can be provided to the implementation, and it will create a schedule at run time, download it, and use it, rather than relying on a dynamic routing layer or trying to lay down n^2 streams for all possible permutations.

All of the above implementations rely on the *gen_io* function to return the processor code necessary to make the implementations work. This function implements the *read* and *write*, or the *func*, functions that the processor calls. For simple operators like `stream`, these functions may just read or write to an interface register and return. More complex operators like reduce and prefix include code for the necessary semantics in gen_io.

Replacement Implementations

The final class of implementations consists of replacements. This class of implementation defines the *replace* function in the definition structure. When called, it returns an operator (or group of operators) that will carry out the function. For example, the `allreduce` operator is implemented this way by mapping it to `reduce+broadcast`.

A related operator 'implementation' is the *stub* implementation. This is typically an implementation that is used on one of the operators returned by a previous replacement, and it calls the other operators in the replacement group. For example, a `barrier` operator may be implemented as an `allreduce` along with a piece of stub code that calls the `allreduce` *func* function with an arbitrary argument and discards the result.

4.1.3 Meta-Implementations

The compiler can also create *meta-implementations* out of existing implementations to support operators with data-dependent arguments. There are two basic ways to do this:[1]

- A runtime-parallel implementation simply takes all the possible values of a runtime operator's argument, and instantiates a separate set of streams for each one. At run time the COP code uses the streams that correspond to the selected runtime argument. In this case, the compiler temporarily modifies the operator to have the appropriate compile-time arguments (one at a time), calls the stream function, and adds all the resulting streams into the set; then, when generating the code, the compiler simply creates a frontend function to choose which streams to use.

- For a runtime-subphase implementation, a runtime operator is decomposed to multiple compile-time subphases, each holding one compile-time-valued instance of the operator as well as any other operators present in the phase. Each *subphase*, accordingly, is the same except for an operand value in the operator in question. Doing this requires a separate phase compilation for each subphase; the implementation's stream function is called once for each compile-time flavor. This option is impractical when there are other sequentially-preceding operators in the phase, since before this operator's load function is called it will not be known what subphase to issue. If one were picked at random it would be necessary to reload it at that point, so it makes more sense to load a separate phase.

Notice that the multiple-phase meta-implementation, in particular, may cause an explosion of possible phases that need to be stored in the processor's memory; the number of subphases accumulates as the product of the runtime value range of each operator's operand(s). For just two operators, each with one operand that can take on any of 256 values (for example), there are already 64K possible subphases for one phase. Accordingly, the compiler can be given a limit on the number of phases per node, so as not to overwhelm the node memory with phase data.

Choosing which subphase to load is fairly straightforward. Each load function is responsible for taking its argument and using it to generating an *operator index* (in the range $0...k-1$, where k is the number of subphases associated with the operator) and storing it in memory. Once all the operator indices have been generated for a phase, the last load function invoked aggregates them into a single *subphase index* used to pick one subphase from the set and issue the router command to start running that subphase.

[1] These techniques are not implemented in the current compiler.

4.2 Online Scheduled Routing

For some communication patterns, the most efficient implementation involves simply providing online routing. In particular, any communication pattern where the communication sources and destinations change rapidly relative to the message size, and where there is no constraint on the values of the sources and destinations, is likely to be most efficiently implemented by online routing.

Furthermore, providing support for online routing in a scheduled router is a guarantee of generality: by providing the same functionality as a dynamic router, the compiler ensures that any application that can be run on the dynamic router can be run on the scheduled router. For applications that require a great deal of online routing, scheduled routing will not perform as well as dynamic routing; however, it seems likely that a large class of applications have sufficient exploitable regularity in their message traffic that the fraction of online routing necessary will be low enough not to compromise the overall run time significantly.

4.2.1 Implementation

Online routing cannot be provided without intervention from the processor (or, if available, equivalent hardware in the router). This section assumes that the processor manages the routing using some form of interrupts to read and write data to the interface. If interrupts are not available, provisions for polling would need to be made in the generated code. Most of the discussion is also applicable to the case where the scheduled router has dynamic-routing hardware available.

The current implementation of COP relies on an interrupt-driven system that provides buffering in processor RAM for messages. A future implementation of NuMesh (or a similar system, such as RAW [3, 67]) could include traditional online-routing functionality in hardware.

Online routing is handled by providing one-hop streams between all adjacent nodes involved. One VFSM reads an interface address dedicated to, *e.g.*, sending data to $+x$, and routes the data out the $+x$ port. The neighboring processor similarly schedules a VFSM to read data from its $-x$ port and deliver it to an interface address dedicated to reading online messages from its $-x$ neighbor.

The online-routing code manages a number of queues on each node. Interrupts are initially enabled for data arriving on each active input stream. As data arrives, the code parses a header word holding a node number, 'destination queue' (discussed below), and length. A message buffer is allocated and queued for the appropriate output interface address, and interrupt notification for empty status on that address is requested. The input handler fills the message buffer at the same time that the output handler empties it. Messages are originated by creating a new message buffer with the passed data and placing it directly in the appropriate output queue. As each output queue empties the code disables interrupts for that output. Messages terminating at the node are placed in the specified destination queues; to read them, the application code calls a routine that waits for the message buffer to be completely filled in, then returns the data.

Each stream of an online operator is assigned a destination queue to deliver their messages to on the destination node. If memory, and header-word bits, are no object, each stream can simply have a distinct number, so that each queue is used only by a single operator. Thus, an operator can issue a read and be guaranteed to find only messages intended for it waiting on the queue it reads.

Given, however, that memory and header bits are both a resource that should be managed efficiently, the same techniques can be used here as are used in Section 7.2.2, assigning destination queue numbers in the same manner as dynamic-router destination interface addresses are assigned. This would then guarantee that that an application can read from a given local queue and read only the messages for the given operator. This analysis makes the scheduled-routing online code asymptotically faster, since reading requires only constant time once the data has arrived, regardless of the number of messages already queued for other operators.

4.2.2 Resource Allocation

Naïve objections to implementing online routing on top of scheduled routing include that it would require all the operators in a phase to be routed online; or, if not, that it might consume wire bandwidth out of proportion to the actual communication needs; or, for a 6-neighbor network, that every node would need 12 interface addresses dedicated to online routing. All of these statements turn out to be false.

While it is possible to dedicate all the timeslots in a phase to online routing, doing so is not necessary. If an operator that uses online routing shares a phase with an operator that does not, communications for the two operators can be scheduled in the same proportional manner that is used to schedule timeslots for offline-routed operators. Furthermore, fewer online-routing timeslots can be scheduled between nodes that have less online communications volume.

To ascertain how much online-routing bandwidth to allocate and how many interface addresses are required, the compiler iterates through the operators that are using online routing in the phase. For each operator, it examines the possible source/destination combinations that might exist, and allocates bandwidth along the e-cube routing path taken for each case. Thus, each operator yields a set of links that may be expected to support a certain data volume from that operator in this phase. For each link, all the contributions for all the operators in the phase are included. Two operators running sequentially during the phase would contribute the MAX of their bandwidths to the link, whereas two operators running in parallel would contribute the sum of the bandwidths to the link.

Consider the example of a 2D mesh with one stream (A) specified with unknown source and destination, and another stream (B) in parallel with A, whose destination is unknown but whose source is always processor zero.

```
(parallel
  (stream 'A (runtime) (runtime))
  (stream 'B 0 (runtime)))
```

In this case, a fixed bandwidth is allocated for every link corresponding to operator A; the bandwidth allocations on all the positive-going x links in the first row, and all positive-going y links, are then increased, corresponding to the e-cube delivery patterns of operator B.

The streams associated with online routing are associated with the phase itself, rather than any specific operator. When the stream information is passed to the stream router, the phase's online-routing streams are included along with any streams from operators using offline routing. On return from the stream router, the resulting bandwidth for each online-implemented operator can be computed by re-walking all the links, and, for each operator, setting its bandwidth to the MIN of all the links in all the paths its data might use.

Online routing would normally be considered to consume one interface address for read and one for write for each direction (up to twelve for a six-neighbor mesh). However, as for bandwidth, online-routing interface addresses can be assigned only where needed by the specific operator(s) implemented with online routing. For example, a stream operator with runtime source and dest on a one-dimensional subset of the network would only need a total of four interface addresses (and only two on the nodes at the ends of the subset). Assigning interface addresses on a per-node basis can thus noticeably reduce interface address pressure.

4.2.3 Future Extensions

A variety of extensions could be used to improve the performance of online routing over scheduled routing.

One extension not currently implemented in the COP compiler is to allow non-nearest-neighbor streams. One simple version of this would be express channels [19]. Some nodes would have an additional set of streams available to them beyond the basic nearest-neighbor sets; these streams would connect to other nodes distant in the mesh. This would decrease the latency experienced by non-local messages, though it would also decrease the bandwidth available on the other network streams, as well as increase the interface address pressure. The address pressure in this model can be decreased by skewing the set of nodes responsible for managing express channels in each direction, such that each node only has one in-express and one out-express (at most) in addition to its nearest-neighbor streams. Assuming that express channels are k links in length, using them would reduce the number of routing steps for a path of distance l from l to, on average, $k/2 + l/k$; however, the cost would be a bandwidth factor of two for co-scheduling the express links.

A similar approach might be to abandon the mesh network for online routing and overlay a virtual hypercube-style network on the mesh, again trading off some bandwidth for latency. Here, however, the number of streams rapidly exceeds the schedule RAM. For a 3D mesh of size N there are N virtual hypercube streams crossing the bisection, but only $N^{\frac{2}{3}}$ wires. This gives a $N^{\frac{1}{3}}$ slowdown, and more importantly bounds the size of mesh that can fit in a given schedule size. For example, with a maximum schedule length of 64, a full hypercube can only be done for meshes about $4 \times 4 \times 4$. However, a coarser-granularity hypercube with a nearest-neighbor step for final routing might still be of benefit.

Using fixed interface addresses for reading and writing maximizes bandwidth and minimizes latency for the online streams. However, particularly if using something like express channels, address interface pressure can be significant. A single address could be used to read all online messages, but as the messages could then be interleaved Every word would need to be tagged to dispatch it to the correct message, at some cost in network bandwidth as well processing overhead. Similarly, a single address could be used to write all messages, reprogramming the router to connect the address to a different VFSM each time the interrupt handler switched to writing a new stream. Both solutions have a place when online routing is infrequently used and the interface addresses are a scarce resource for other co-existent operators;

however, implementing and assessing this is left for future work.

A shared-memory model would be easy to add to processor-based routing. Messages destined to a particular virtual interface address could be interpreted as memory-read requests. For such messages, the interrupt handler would not queue the message for later reading, but instead read the specified memory location(s) and return the results to the given processor with a return message. More sophisticated memory semantics (such as Fetch-and-Op) could also be included at relatively low incremental cost in the handler. This implementation of shared memory could easily be used under a software shared memory implementation such as CRL [31].

Online-routing hardware could also be included directly on the router. Online-routing timeslots would be scheduled just like any other data transfer. If a message were trying to go in the $+x$ direction, the routing hardware would just wait until the VFSM responsible for moving data from the online-routing hardware to $+x$ is scheduled, and then provides the appropriate word to the neighbor. Similarly, when the message has reached its final destination, once a VFSM to move data from the online-routing hardware to the processor interface were scheduled, the data would be written to the interface, with the online routing engine providing an interface address (perhaps a cycle early) as well as the data word. Internal queues would be necessary to allow messages to bypass one another when accessing the interface addresses. Additionally, if desired, a few wires could be added between nodes to hold a virtual lane indicator for routing between nodes.

Chapter 5

Managing Multiple Phases

This chapter looks at the question of managing multiple phases in a scheduled router. It starts with a discussion of the 'continuing operators' that traverse phases in Section 5.1. Section 5.2 discusses how to ensure all the data I/O is complete before changing phase, and Section 5.3 explores the restrictions placed on wire usage by consecutive phases. Section 5.4 discusses how to allocate router memory to the various phases, and Section 5.5 presents an algorithm for associating phase loads with operator load functions.

5.1 Continuing Operators

Before addressing the management of multiple phases in an application, this section will first discuss the one 'exception' to the phase-switching model: operators that continue from one phase to the next. Such operators require special handling. For example, as was discussed in Section 2.4.2, a user may wish to have a particular broadcast operator available throughout the application:

```
(parallel
  (broadcast 'stdin 0)
  (sequential OP OP OP ...))
```

If the compiler decides to break up the `sequential` clause into multiple phases, it becomes necessary to take special measures to ensure that the `stdin` operator is valid through all the phases. It is important to guarantee that any data that was in transit during a phase change remains valid, and that the operator is available for reading and writing in each phase.

5.1.1 Simple Continuation

The first issue to realize is that it is not sufficient to place a copy of the operator in each phase—*e.g.*, to have a separate broadcast from node zero added to the set of operators in each phase from the example above. If this was done, there would be separate VFSMs associated with the operator in each phase, and the compiler (or application) would have to ensure that all the VFSMs from the previous phase had cleared before switching to the new phase; however, the semantics of a continuing operator is that the source may write data into it at any time, so it is difficult to guarantee that the operator's stream(s) from the previous phase are empty before switching to the new phase.

Instead, the compiler simply forces the stream to use the same VFSMs in the following phase(s). This way, the stream can simply be ignored during any phase changes; any buffered data in an intermediate VFSM will simply be re-launched once the schedule for the new phase is initiated, since the VFSM will be the same in both the new and old schedule.

In this simple case, continuing streams do not have to be scheduled in the same time slots. The buffering associated with a VFSM means that the compiler can safely stop scheduling a VFSM for some period of time, then reschedule it, without losing any data; the data will simply wait in the buffer. If no buffering were allocated to VFSMs (perhaps because the compiler were able to assert that the receiving processors could always keep up with the data flow), it would be impossible to switch phases like this, since data would be dropped as soon as the compiler stopped scheduling a VFSM belonging to a stream actively carrying data.

Furthermore, a continuing stream can be rescheduled to have different bandwidth if necessary. Consider the case where the continuing operator shares one phase with some operator with traffic $4T$, and the next phase with an operator with traffic T. The continuing operator should end up with proportionately more bandwidth in the second phase, since the continuing operator is likely to be in the critical path unless its bandwidth is increased.

Regardless of the scheduling, the exact sequence of VFSMs that carry data must be preserved. In particular, this requires that the stream follow the same route, use the same pipelines on each node, and even include delays in the same point(s) in the new routing. When routing the stream in the second and following phases, this is given as a requirement for the stream router. Doing so reduces the router's flexibility somewhat, but still allows it to choose arbitrary timeslots in the global schedule to place the stream in the new schedule.

5.1.2 Continuation to Multiple Phases

The situation is more complicated if the operator continues into a part of the application with multiple threads of control. In this case (as discussed in Section 2.4.3), there may be multiple simultaneous phases with disjoint spatial extents. In this case a somewhat more restrictive solution is required. For example, consider the case where the `stdin` operator starts in a phase with some setup operators, then continues to a phase where portions of the mesh are independently running through a several-phase loop:

```
(parallel
  (broadcast 'stdin 0)
  (sequential
    (parallel OP OP ...)          ; setup phase
    (parallel
       (loop OP OP OP ...)        ; left side of mesh
       (loop OP OP OP ...))))     ; right side of mesh
```

Since the `stdin` operator is global, but portions of its path may be in different phases at the same time, it is necessary to reuse not just the VFSMs but their exact scheduling as well. The precise set of schedule timeslots for the stream form an 'interface' that all the phases must adhere to so that the continuing operator can run seamlessly despite any changing phases. Thus, the stream router is given not just a list of VFSMs to reuse, but a list of VFSMs and schedule slots to reuse.

While this is necessary for correct operation in the disjoint-phase case, it is not used by default. It requires the operator to have a fixed bandwidth from phase to phase, and it also makes the remaining operators harder to schedule, since they must work around fixed 'obstacles' in the schedule where the continuing operator(s) have been nailed down.

5.1.3 Continuing Online-Routing Operators

Continuing online-routed operators are handled somewhat like offline-routed ones. A continuing online-routed operator requires some subset of the online streams to be present in each phase it is present in. Those streams are routed normally in the first phase that the operator is present in. In the following phases, if there are continuing online-routed operators present, the possible links for each online-routed operator are traversed and the routing information from the first phase where that operator occurred is copied in. As a result, the online-routed streams will work just the same as regular offline streams: they will usually just be rerouted using the same VFSMs and different schedules, but may be fixed to their existing schedule when they cross phase boundaries (as discussed above). Where new online streams are present in a following phase (or where the following phase does not have any continuing streams), new VFSMs and new schedule slots are allocated as would be done for any offline-routed operator.

5.2 Changing Phases

Changing phases during the execution of an application can be of critical importance when the nature of communications changes periodically. This section examines the issues involved in performing a phase change.

A phase requires all the nodes in the phase's spatial extent to run coordinated router schedules. Once a given node has completed the communication and computation relevant to that phase, it instructs the router to shift to the next phase. Once some nodes have shifted to the next phase, data communication can continue at will for the new phase. Only when all the nodes are in the new phase can the compiler guarantee that, for example, a long-distance stream communication between two nodes will be able to deliver data from one processor to the other.

An important restriction is that nodes must always switch to the correct next phase. For example, if a given phase is optional, it must be skipped or not skipped uniformly by all the nodes involved. Similarly, if a node is involved in some phases and not in others, it must be careful to sequence through the correct phases; if a node is in some phase's spatial extent, it must be careful to run the schedule for that phase at the appropriate time.

Before a node can switch to the next phase, all communications that involve the nodes must have finished. (Communications from continuing operators are exempt from this requirement, as their communications will continue to be supported in the following phase.) Simply waiting until the last I/O word has been written to the last operator in the phase is not sufficient. All written data may not have left the router (or indeed left the processor interface); additionally, data from other nodes may still be in transit through the node in question. In either of these cases, the processor can't instruct the router to change phases, or the data will be stranded in the previous phase (or possibly injected inappropriately into a new operator in a subsequent phase, depending on whether the VFSM in question is reused later).

The text below assumes that phase changes are always issued by the processor. In fact, a phase change could be issued by neighbor nodes as well. One simple way to handle phase changing would be to have a final barrier (*e.g.*, a reduction across the phase), followed by a multicast that directly instructed the routers to change phase. Doing so is not straightforward with a simple scheduled router, however. As discussed in Section 5.4.1, there may be different schedules resident in different routers. While the correct schedule could be guaranteed to be available *somewhere* in the router, special handling would be needed to ensure that the correct schedule was started on all the involved nodes. Accordingly, the request to change phases is simply passed through the processor on each node.

There are three issues involved in terminating a phase. The first is terminating a single operator in isolation; the second is figuring out which operators are relevant for phase termination; and the last is using this information to determine how to terminate the phase.

5.2.1 Operator Termination

There are a variety of ways to determine whether a node has transferred all data related to a given node in a phase. Remember that these techniques are only applied at the time of a phase transition, after the operator has been used for the last time in the phase. When operators are used multiple times throughout the phase, there is no need for any particular technique to guarantee that the data is transferred correctly, since no phase changes are yet being performed.

Message-Bound Implementations

The most straightforward case is to rely on the operator's message size. If an operator writes a relatively short message, it may be entirely buffered in the mesh; the writer may finish writing the message long before the reader starts reading. By contrast, a suitably long message can provide synchronization guarantees. Messages are read synchronously on the remote end by COP (except with online routing, as discussed in Section 4.2). Accordingly, when an operator's message size is longer than the amount of buffering in the mesh between the source and the destination, the reader will have to start reading before the writer can finish writing.

As a result, since data is guaranteed to be being drained from the mesh by the reader, the data currently on the node's router will be forwarded off the router and onto the neighbor node in a sharply bounded period of time. While it is possible that the remote node may encounter cache misses while writing the data read from the interface, that time can be factored into the length of time to delay to ensure that sufficient time has passed for the last data word to be able to leave the writer node. Additional sources of delay on the reader, such as interrupts (*e.g.*, from the online routing mentioned previously) can be temporarily suspended to make the necessary timing guarantees. Implementations that offer global synchronization guarantees in this way are referred to as *message-bound*.

Operator implementations may have varying degrees of complexity, which result in different definitions of a 'long' message. For a simple multicast-based broadcast, the number of

VFSM and processor interface buffers present along the longest path from the source to any destination can be used. Writing one more than this many words guarantees that all the other nodes in the mesh are reading the broadcast, since otherwise the writer would have stalled. For a more complex operator, such as a circular shift, the compiler must follow multiple streams from node to node around the mesh to determine the number of possible buffer registers that must be filled for a given operator before it can guarantee that the I/O function completing guarantees a degree of synchrony with the other nodes in the subset. Still other implementations, such as parallel prefix, can automatically make a 'long message' synchronization guarantee for any length of message, since the operator itself involves synchronizing the nodes in its subset.

So far the discussion has focussed only on delays for the writer node. This makes sense, since when the reader function completes, clearly all the data on that stream has left the router. However, some operators use multiple streams through a single node. For example, a circular shift operator on a linear array of nodes has one long stream that goes from one end-node to the other. When an intermediate reader node completes, it must also perform a pause to ensure that the tail end of the message on the long stream has time to move through the node.

This technique is implemented in the compiler by having the returned implementation-specific information structure hold a pointer to an array of per-node phase delays, if the operator is message-bound for that implementation. Then, if a phase change follows a particular operator, those delay values are used before changing phase.

Processor-Bound Implementations

When the message size is short, a different technique can be employed to determine when the operator has finished writing. A single operator implementation may be implemented such that when the 'read' or 'write' function returns control to the processor, it guarantees that the node has finished all routing for that operator's data. Such implementations are called 'processor-bound'.

As an example, a standard multicast implementation of broadcast will not be processor-bound—when the write function completes, data may still be present in the router. A processor-bound implementation of broadcast includes the source node itself in the multicast. By ensuring that the last portion of the multicast on the source node is the write back to the processor interface, the processor is guaranteed that, once it reads the final word of the message back from the router, the router will perform no further I/O on behalf of the broadcast operator.

Making such a guarantee can be harder for other operators. For example, a linear, nearest-neighbor end-off shift can include multicast-to-source on each stream, thus making it processor-bound. By contrast, a circular shift on the same array adds a single stream traversing the entire mesh (as mentioned above), and without further explicit multicasting to the intermediate nodes there would be no way to know when a given node had finished routing data for the operator.

Performance of processor-bound operators will suffer if the stream would normally be allocated bandwidth greater than half the processor's interface speed; that is, for an optimal processor interface, a processor-bound operator can't run faster than 50% bandwidth. This is due to the fact that normally, a processor-bound implementation requires some nodes to perform an extra I/O on every cycle. In practice, this limitation is not severe, because if multiple operators are scheduled in a phase there will often be network bandwidth limits such that no one operator will be able to run that fast.

A significant aspect of this technique is that when a node finishes the phase and instructs the router to change phases, it has no idea how long it will be until its neighboring nodes also change phases. This fact is considered further in Section 5.3.

5.2.2 Determining Which Operators are Terminating

When there are several operators in a phase, it is necessary to determine which operators are terminating in the current phase, since continuing operators are irrelevant to a phase change and can be ignored. Additionally, it must be determined which operators may be called last; that is, which operators may immediately precede a phase change.

Optional Operators

Before presenting the algorithm used to determine this status for each operator, it is important to consider that an application may use certain operators under some circumstances and not others. For example, in a Gaussian elimination, the user may wish to find the row with the largest pivot value to improve the numerical stability. If in the ith iteration of the algorithm, the chosen row is not row i, the application will swap it with row i. To improve performance, the application may choose not to use the operator at all if the chosen row *is* row i. In that case, it must pass this information to the COP compiler, since it effects which operators may run last in a phase (among other things).

Taking the Gaussian elimination example in Figure 2-7 (p. 39), this application-level optimization can be added by replacing the last `subset` group in the elimination loop with

```
(subset cols
  (optional
    (stream 's2 (runtime) (runtime) :m chunk))
  (broadcast 'b3 (runtime) :m chunk))
```

The additional `optional` construct is used to mark a list of operators as being optional; it gives its arguments implicit sequential context. Optional operators may, of course, be scheduled in with other operators into a single phase. If multiple operators or grouping constructs are listed in an `optional` clause, they must all be chosen or skipped together; a set of independently-optional operators must each be enclosed in a separate `optional` construct. In a similar vein, all the nodes in the operator's subset must make the same decision as to whether or not to use the operator in any given phase.

Deriving 'Terminating' and 'Last' Status

Let us now turn to the algorithm that determines the status of the operators in a phase. An operator is 'terminating' if this is the last phase in which it appears on this node. An operator is 'last' if it is one of the operators that could be invoked immediately prior to the phase change (and is terminating). Given the presence of multiply nested sequential and parallel operator groupings, computing these flags is non-trivial.

Marks are maintained in the operator tree to guide the search. The algorithm starts with the operators most recently added to the phase (*i.e.*, those that appear furthest down in the COP program). It skips any operators that have already been 'seen' (as discussed below), which will be the case for all continuing parallel operators. All operators not previously seen are thus immediately marked as 'terminating'. The algorithm walks up the operator tree for each terminating operator; if it reaches the top of the tree without encountering a marked node, it learns that this operator may be executed last in its phase. If the algorithm hits a marked node, it inherits the 'last' status of the marked node.

Having set the operator's 'last' and 'terminating' status, the algorithm walks up the tree to the root again, this time setting the mark on each operator as it goes. It clears the ancestor nodes' 'last' status initially as it walks up the tree, since the remaining operators in the phase will be earlier in the tree and thus presumably earlier children of a sequential group. However, if the algorithm hits an `optional` group in the tree, it sets the 'last' marker on all higher ancestors, since operators preceding an optional may themselves run last in a phase.

During the same walk to the root of the tree, the algorithm watches for parallel groups, and marks all their left side (earlier) descendants as seen, as well as 'last' if the original operator itself was last. A harder question is to pick out which parallel operators are terminating in this phase; the algorithm initially marks the left side parallel group descendants as terminating, but once it comes to a sequential ancestor with further children, it stops marking descendants of higher parallels as terminating. This is a consequence of the fact that any sequential group split between phases will have any higher-level parallel constructs also split between phases, and thus not finishing in the current phase.

5.2.3 Barriers for Phase Termination

Termination issues for a phase consisting of a single operator were considered above. For multiple operators in a phase, or if the spatial extent of the phase is larger than (some or all of) the subsets of the operator, things are more complex.

Having found the set of terminating operators, the compiler must next find the combined spatial extent of the terminating operators. The nodes in this extent are the important ones, since they are the ones that must switch phases. (The extent of the phase itself is normally taken to be the union of the extents of the operators *starting* in the phase, not terminating; this definition is appropriate when routing the streams in the phase. This discrepancy is returned to in Section 6.2.2.) Operators whose subset is a proper subset of the phase's combined extent do not *cover* the phase; that is, for that operator, some of the nodes in the phase's combined extent do not perform I/O.

Any barrier action can be performed separately over each *partition* of the phase's extent. The compiler first divides up the operator instances into disjoint partitions by grouping together any operator instances whose spatial extents overlap. The barrier techniques are then applied per partition. For example, one part of the phase may end with a message-bound broadcast covering it, and another part may simply have a few streams; each part would have a different barrier technique. This technique works because any node only communicates with other nodes in its partition of the phase; as a result, it is irrelevant to the nodes in one partition what the state of nodes in other partitions is. Whether its router has more data to carry or not does not depend on those other nodes.

One consequence of allowing separate barriers over each partition is that streams from one partition must be constrained not to be routed into another partition. In general, this is unlikely to be an issue, since partitions generally represent equal-traffic instances of operators, and misrouting into a neighboring instance would simply take necessary bandwidth away from that instance. However, the compiler must guarantee that this is done by passing the partition as a constraint to the stream router.

It is worth observing that any phase with no following phases—such as at the end of the application, in the absence of an overall loop construct—is privileged, since it need not include a barrier. Given that the application never switches out of such a phase (except by user intervention, *i.e.*, a global reset), there is no need to waste bandwidth incorporating an explicit barrier construct.

The possible options for terminating a phase are given in Table 5.1; the first 'type' was mentioned in the preceding paragraph, and each of the remaining types are discussed in the following subsections.

Name	Description
none	No following phase, so no barrier is performed
msg-sync	Ops are message-bound; brief pause at phase change
hijack	Create message-bound MST from terminating ops' streams
barrier	Insert an explicit barrier in the phase
no-op	All ops processor-bound; local synchronization only

Table 5.1: Barrier techniques for safe phase termination

Message-Bound Synchronization

A convenient way to synchronize for phase changes is to do it implicitly, using long messages. If all the 'last' operators in the phase have sufficiently long messages, this approach is feasible; it is also efficient, since it requires no additional communication, and yields a globally-synchronized phase change.

If some of the last operators do not cover the phase, this approach will not work. Nodes that are not involved in the operator's I/O operation (even if their routers carry traffic) can't know when it is legal to switch; furthermore, since they perform no blocking flow-controlled reads or writes, no guarantees can be made about where in the code those nodes' processors are.

An additional constraint must also be placed on the use of this technique for phase termination. If the phase includes operators that are not message-bound, and whose subsets are not subsets of the 'last' operators, completion of the last operator's I/O and a short wait no longer guarantees that the node will be cleared of traffic. Consider a phase on a 2D mesh with one row-wise and one column-wise operator. If the row operator is last, and message-bound, it may complete while data from the preceding column operator is still in transit (*e.g.*, on a wraparound circular-shift stream). Only if the terminating non-last operators are message-bound themselves, or else a subset of the last operators, can the compiler assume that they have completed on a given node when the last operator has completed.

It is possible even in the two-operator scenario in the previous paragraph that a previous message-bound operator may require a longer delay than is provided even by the entire following message-bound operator. Consider, for example, a mesh with long columns and short rows; the column operator may require a long delay for the synchronization, while the following row operator requires a short delay. This is handled by tracking when each operator is run, and setting a time value for which the COP code must wait before changing phases, each time a message-bound operator is run. The time value is only updated if set to a later value, however, thus handling the two-operator situation spelled out here. When it is time to switch phases, and message-bound synchronization is being used, the COP code simply waits until the clock reaches the given time value. At that point, the application is guaranteed that all data from operators terminating in this phase has left the router, and the COP code can safely instruct the router to change phases.

Some or all of the required delay in this mode can be overlapped by performing any required router reprogramming while the required delay time elapses. This way useful work can be accomplished while waiting for data to finish propagating off the node.

Hijack Synchronization

If, for some reason, a message-bound implementation can not be used to perform the synchronization implicitly, an explicit synchronization step can be used instead. A priority is to avoid giving up scheduled timeslots in any phase, since that would reduce the bandwidth available to the operators throughout the affected phase.

Accordingly, one solution is to *hijack* existing streams to create a synchronization barrier before the phase change.[1] Once a node has finished all I/O for a phase, it can reuse any stream that starts or ends at its processor interface and is associated with a terminating operator. Continuing operators can not be hijacked, since they are carrying data using VFSMs that will continue to be valid in the following phase.

For each subset, then, the compiler attempts to find a synchronizing path through the nodes that allows it to perform the equivalent of a message-bound operator's synchronization, but to perform it explicitly from within the phase-changing code itself. Essentially, what the compiler does is find a directed minimum spanning tree (MST) for each disjoint subset, then compute a necessary message length to achieve synchronization. At phase-change time, each node reads and/or writes the appropriate number of words; when it has finished, it knows that it has synchronized with the other nodes, and it simply waits a few cycles to ensure that any data still on the router has left before instructing the router to change phases.

The simplest example is if there is already a compile-time broadcast operator in the phase. The compiler can hijack the operator's multicast stream and use it, alone, to perform the phase-change synchronization. However, it may be the case that there is no such operator; consider the case of a flood-fill broadcast. In this case, the compiler finds a set of nearest-neighbor streams that allow the synchronization message to be broadcast, and arranges for each node to know which stream to read from and which stream to write to such that the message is forwarded.

[1] Hijack synchronization has not yet been implemented in the current compiler, but is on the short list for follow-up work.

To find a directed MST, a cost is assigned to each stream. The ideal stream has high bandwidth and covers a large portion of the mesh. The minimum bandwidth of any stream in the MST is used as the first-order cost of the entire tree (since it determines how fast the message can be pushed through), and the length of the necessary message is computed by adding up the maximum number of buffers between the source and the farthest destination in the MST. One node is taken as the source in the MST, while the others are final destinations.

Some streams may be reprogrammed by their implementations to accomplish special goals (*e.g.*, output discarding), as discussed in Section 4.1.1. To allow those streams to be hijacked, a flag is included in the implementation-specific information that indicates whether or not the compiler must reset the streams to have a conventional source and destination(s) in the processor interface on the first and last nodes of the stream respectively. Some streams may not be usable at all; typically this is because they are designed to terminate in the router itself rather than at the processor, or because they are used in some specialized manner (such as for online routing). Such streams will be marked as unsuitable for hijacking by the implementation.

Explicit Barrier Synchronization

If neither of the above approaches is feasible for a phase, it is no longer possible to avoid taking schedule slots away from the actual data I/O.

The simplest approach is to simply add a `barrier` operator explicitly to the phase, and arrange to call it from the phase-changing code. The barrier is then routed and scheduled like any other operator, and when control returns from the barrier's I/O function the COP code knows that all the nodes have completed routing data. Typically a processor-bound broadcast is used to finish the barrier, so that the COP code can immediately change phase when the barrier completes. Notice that a barrier is just an operator like any other; in particular, the compiler must still guarantee that it covers the terminating operators' subsets and that it is processor- or message-bound.

Local Synchronization

Alternately, if all the operators in the phase are processor-bound and cover the phase, local synchronization can be performed, switching immediately after the last operator is done. This is possible because, no matter which terminating operator is executed last on the node, it will only return control when no further data from that operator is using the node. Since this is true for all the terminating operators, no terminating operator's data will need that node when the last operator to run returns control to the application. If this can be shown to be true for a phase, no barrier need be scheduled into the phase and the COP code can change phases simply by issuing the command to the router. However, as discussed in Section 5.3, this may make some schedule slots in the *next* phase unavailable, since the next phase's streams must be scheduled to avoid conflict with the streams in this phase.

Naïvely, one might expect that it would suffice to consider only the operators that ran last in a phase. Thus, with a sequential `cshift` followed by a processor-bound broadcast, it might seem logical to consider only the broadcast: if it covered the phase and was processor-bound, that would perhaps suffice. However, this is not true. As an example, consider a `cshift` on a long linear array followed by a broadcast from the center of the array. Nodes near the center could finish their contribution to both operators quickly, but their routers would still be required to handle the one wrap-around `cshift` message. As a result, all the operators in a phase are required to be processor-bound before it is possible to skip using a barrier.

5.2.4 Alternate Subset Specifiers

It is possible to 'help out' the barrier code by involving the processors more actively in the operators' communication. One of the main reasons to add an explicit barrier to a phase is that particular operator(s) do not cover the phase. Consider the case of a sparse subset, where the nodes involved in processor I/O are not all mutually adjacent. In this case, any operator defined over the subset will not cover its extent, and may require an explicit barrier to exit from the phase. This section discusses a language-level technique to make operators cover the phase and yet still express the same semantics, by allowing an alternate technique for specifying subsets.

The technique involves the use an optional argument, `:valid`, to an operator. This argument specifies that only specified nodes in the subset will be presenting valid data to the operator. With this technique, all the nodes issue loads and read/writes, but only some of the nodes are actually involved in the operation. As an example, an application might specify

```
(prefix 'p 'myfunc :valid '(0 2 4 6))
```

In this case, all the nodes will run the `p_load` function, and all the nodes will issue `p_func` calls as necessary. The nodes not included in the `:valid` set can present any argument to the `load` or `read/write` functions, and should ignore any returned values.

This SIMD style requires all the nodes to know when the operator is being used, but it makes phase-switching easier, since all the nodes are participating in the operator. Further, it makes it possible to use optimized, schedule-generating implementations that require a connected subset. In general, whenever using sparse subsets of this nature, the user should prefer `:valid` if the nodes not directly involved in the I/O are waiting for the operator to complete anyway; otherwise, use `subset`.

5.3 Inter-Phase Wire Constraints

It is important to consider carefully what happens as nodes are changing phase. The compiler guarantees (as discussed in the previous section) that a node does not change phase until after all its communication data has left the router. However, during the time between its change of phase and its neighbor's change of phase, strange interactions can occur between the nodes.

It is important, for example, to avoid the situation where a VFSM on a node in one phase issues a write, and at the same time a VFSM on the appropriate neighbor node, in another phase, issues a read. The write and read belong to different high-level phases, but the hardware doesn't know that, and data is transferred inappropriately, upsetting the application. The remainder of this section discusses how to handle this issue in more detail.

5.3.1 Finding ‘Live’ Communications Edges

There are a number of different interactions that can be seen between adjacent nodes in different phases, having to do with which nodes are reading and writing, and whether the relevant VFSMs in the previous phase continue to be scheduled in the current phase.

Let us focus on wires connecting nodes together. Let node n be the node that is switching to the new phase, and node m be the adjacent node still in the previous phase. For each schedule timeslot t the compiler must consider what used to be happening on the wire between the nodes, and what it wants to happen in the current phase.

Previous op terminated			Previous op continues		
	now			now	
then	R	W	then	R	W
R	yes	yes	R	no	no*
W	yes†	no*	W	yes†	no*

* May reuse if a global barrier or a dependency

† Cannot reuse if router’s valid/accept lines are aliased

Table 5.2: Legality of reusing $\langle$*wire/timeslot*$\rangle$ pairs in following phases

To begin with, consider the case where the wire was used by a VFSM from an operator that has terminated as of the new phase (Table 5.2 gives a quick summary of the following text).

If, in the previous phase, time t was used to read data from the adjacent node m, any operator is free to read or write to m at time t in the new phase. Node m in the previous phase will have a VFSM that writes to n, but since the COP code only change phases once n has finished its I/O, the compiler knows that m will not try to write to n again in the old phase. Accordingly, n can start trying to write or read in that time slot immediately without fear of erroneous data transfers to or from the previous phase.

However, if in the previous phase data was written to m at time t, things are a little trickier. First, let us consider the case where n tries to read from node m in the new phase (this is called a ‘write-to-read’ reuse). Node m was reading from n in the previous phase, and, in the simple scheduled-routing model, this cross-phase interaction is benign: during the phase mismatch, each node is trying to read from the other, so no data will be exchanged. However, an obvious optimization for scheduled routing hardware is to use the *same* wire to transfer valid and accept information; if a node is reading, it interprets the remote value as ‘valid’, and if it is writing, it interprets it as ‘accept’. If the hardware employs this optimization (as does NuMesh, for example), this time slot can not be reused for reading, since during the phase mismatch time both nodes will assert ‘accept’, both will interpret the remote accept as a ‘valid’, and both nodes will determine that the (undriven) data wires hold a valid word.

The final case is where n used to write data to m at time t (as above), and wants to write data to it again in the new phase (a write-to-write reuse). In this case, the compiler must ensure that n does not write data to node m while it is still on the old phase, since it will accept the data as if it were coming from node n’s previous phase. Unless the compiler can prove that this will not happen (as is discussed below), it can’t reuse this timeslot for writing.

When continuing operators are present (that is, streams persist across the phase change), the situation is slightly different. If the operator is not scheduled into all the same timeslots, the compiler must be careful of any slots the operator is no longer using. If n was previously writing to m as part of a continuing operator, the same considerations apply as above: converting the slot to a read in the new phase is only allowed if the valid and accept lines are not aliased, and scheduling a different write VFSM there is only allowed if the compiler can prove that it will not write before the neighbor switches phases. Naturally, the *same* VFSM can be scheduled to continue writing in the new phase in that timeslot.

However, if n was reading from m, it is possible that m continues to send valid data to n even after n has switched phases. As a result, if n were to use the timeslot in the new phase to write data to m, both nodes would end up writing at once, unless the compiler can prove that n will not write before m changes phases. More significantly, a new VFSM can not, under any circumstances, read from m in that time slot, since n could incorrectly receive data from the continuing operator's time slots in the previous phase. The compiler tries to handle these cases by preferentially assigning the same timeslots to the continuing operator's VFSMs; if the operator's bandwidth is the same or greater in the new phase, the compiler will try to assign all the old timeslots to it in the new phase.

5.3.2 Allowing Write Reuse with Barriers

One way to prove that the next node is already in the new phase before writing to it is if the previous phase ended with a global barrier (*i.e.*, anything except the 'Local Synchronization' discussed in Section 5.2.3). In this case it is possible to guarantee that simply waiting a few extra cycles after changing phase will leave the neighbor nodes in the new phase as well. When this guarantee can be made, the compiler is free to reuse any timeslot marked with a '*' in Table 5.2.

The delay time that must be spent after instructing the router to change to the new phase varies. The compiler must determine how long it will be until the neighboring nodes change to the new phase as well. Depending on the type of global synchronization, the compiler examines either the message-bound operator(s), the hijacked stream(s), or the barrier implementation to determine how many extra cycles to wait.

In all of these cases, the compiler follows the streams through the mesh to the processor interface on the neighbor. The number of buffer registers in the stream gives the number of words that must clear before the neighbor can change phase; the minimum bandwidth allocated to the stream(s) gives the speed with which it can clear those words. Combined, it gives the minimum time to wait before the neighbor node can change phase. The compiler may also have to add a hardware-specific delay before the phase change is complete; for example, there may be a router pipeline delay to wait for after the router is instructed to change phase, and there may be a delay before the node can pass the command to the router to change phase. These numbers can be derived exactly by looking at the scheduling details of the neighbor nodes.

So far the discussion has implicitly assumed that each node must wait for all its physical neighbors. In fact, if in the new phase a node is not communicating directly with some neighbors at all, it can ignore them completely. If the previous phase is a mesh-wide broadcast, and the next phase is purely row-wise communication, there is no need to worry about how long it is until a node's column-wise neighbors join the new phase, since the node is not communicating with them. Accordingly, each node waits only as long as necessary to get all the neighbors it is communicating with in the new phase into the new phase; 'neighbors' is used below to mean only neighbors the node is communicating with.

Similarly, if a node did not communicate with its new neighbors at all in the previous phase, the compiler does not have to worry about inter-phase wire dependencies, because the wires between any two such nodes were not used at all the previous phase. Accordingly, considering the case where a row-wise partition of the mesh is followed by a column-wish partition, there is no need to wait at all after performing the phase change; none of the node's new neighbors were neighbors in the previous phase, so there are no problems with communication overlaps.

5.3.3 Other Wire-Reuse Considerations

Some phases can follow more than one other phase. For example, the phase at the head of a multiple-phase loop can either follow the phase that precedes the loop or the last phase of the loop. In general, the restrictions of all previous phases must be handled when scheduling. Furthermore, some phases consist of multiple subphases; this must be taken into account in the following phase when determining what timeslot/direction pairs are legal to use. If the previous phase uses subphases sequentially (*e.g.*, the parallel prefix schedule-generator implementation), only the last subphase's reads and writes are considered as relevant to the current full phase. However, if the previous phase has multiple subphases one of which is chosen at runtime, all the communication streams used by all the subphases must be considered.

Furthermore, it may sometimes be necessary to constrain the use of wire slots by the *following* phases; for example, when scheduling a loop, some phase must always be scheduled last, and the compiler must constrain that phase both by the phase(s) before and after it. Making this transformation is easy, however; simply do a row/column transpose of each part of Table 5.2, and determine which table to use by whether the *current* streams are continuing, rather than by whether the *constraining* streams are continuing. The need to be aware of loops in this manner is one of the stronger reasons why the `loop` construct is present in the COP language.

Upon first consideration, it may not be clear that it is only necessary to examine the preceding (or, if appropriate, following) phases on the local node to learn all the possible communication attempts that may be happening on neighbor nodes. In particular, one might think that when a node is involved in different subsets of operators, it might be temporarily adjacent to a node in a phase not related to either the current or previous phase. However, consider that COP requires some form of synchronization during phase changes. A node can only move into the following phase once it has synchronized to some extent with the other neighbors in its phase (as discussed in Section 5.2). As a result, no node in the same extent can get more than one phase ahead of a neighbor.

Some scheduled routers may allow a hardware solution to the wire-reuse problem. For example, an N-coloring of the phase graph could be done and then nodes could exchange $\log N$ bits on each transfer. Only if the bits match would the transfer be considered successful. The bad 'accept' and 'valid' values could be arbitrarily chosen as the all-ones bitmask. To reduce the number of wires and pins, this could also be done via reprogramming: a node's neighbors could set a 'phase number' field for each communications direction, and the node itself could set a node-wide phase number field. Only when they matched would reads and writes from or to the given direction be accepted (other than reprogramming writes, of course).

5.3.4 Dependency Analysis to Minimize 'Live' Edges

After a local barrier, the compiler can attempt to minimize the number of timeslot/direction pairs that are unavailable. The number of such pairs can be reduced by performing dependency analysis on the operators. Doing so, in principle, allows the compiler to determine when it is safe to reuse a given $\langle$*wire/timeslot*$\rangle$ pair based on a guarantee that the neighbor node has joined the current phase.

Creating the Dependency Graph

To compute dependencies, a directed graph representation is used, where each vertex corresponds to some action, and each edge corresponds to a dependency, *i.e.*, an indicator that the vertex at the head of the edge represents an action that happens after the vertex at the tail.

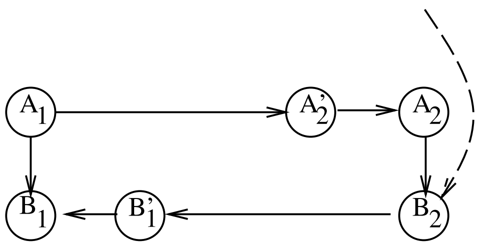
The algorithm starts with vertices that correspond to reads/writes for a given operator on a given node, and labels those vertices with the name of the operator subscripted with the node number (*e.g.*, A_2). There are called 'I/O vertices'; each such vertex is linked to vertices with the same subscript by walking the operator graph and determining which operators are active on each node and what the relationships between them are. On a given node, the algorithm can eliminate all operators not active on that node (*i.e.*, if the node is not in the operator's subset). Walking the remaining operator graph provides time relationships between sequential operators, as well as loop information and information on optional operators. This information is used to create the initial round of edges, with each I/O vertex (operator) pointing to all the operators that can follow it on the node in question.

Now the algorithm adds vertices corresponding to the time of initial arrival of data from an operator at a node, and labels these 'arrival vertices', like the I/O vertices but with a prime marker (*e.g.*, A'_2). Then it adds edges from each I/O vertex where writing is performed to the arrival vertex on the destination. Since different implementations of operators have different communication characteristics, this information on who writes where is implementation-dependent. I/O vertices can't simply be linked to I/O vertices in this step, since a given vertex may or may not write before another vertex does.

Finally, the algorithm adds edges from each arrival vertex to its corresponding I/O vertex, since the data will arrive at the vertex before it can be read. (These two vertex types are a simplification; the start and end of all writes, arrivals, and reads could be represented as separate vertices, and linked appropriately; however, the dependency information needed here and in Section 7.2.2 is available through these two vertex types alone.) Figure 5-1 presents a simple example of a dependency graph for a short fragment of COP code. (The dashed line in the figure is explained later.)

Computing Wire-Reuse Dependency

The compiler supports creating the graph structures discussed above, since they are used by the dynamic-routing backend (Section 7.2.2). However, the compiler does not currently check for dependencies when determining wire reuse, using only the conservative approximations available with global barriers. Accordingly, the remainder of this subsection simply sketches the necessary techniques.




```
(sequential
  (stream 'A 1 2)
  (stream 'B 2 1))
```

Figure 5-1: Dependency graph for simple COP fragment

Let us consider an operator B in some new phase, which from node n ($n = 2$ in the sample figure) might write during some timeslot t on wire w, used in the previous phase by some operator writing from node m ($m = 1$ in the figure). If, for some operator A also in the new phase, B_n *always* follows A_m via A'_n, B can freely ignore the restrictions on wire w, timeslot t. Ignoring the dashed line in Figure 5-1 for now, node 2 is able to reuse the timeslots that were used to communicate with node 1 in the previous phase to route B in in the new phase. Thus, the 'no' entries in Table 5.2 can be replaced with 'yes' entries for operators in the new phase that pass the dependency test.

'Always' is taken to mean that any preceding control-flow path on the local node must lead to a dependency on A for it to be a valid dependency in this case. Thus, if an operator B under consideration has an optional predecessor, the control-flow path to B will include a bypass of that predecessor. Unless this bypass path itself leads to an operator that writes to n in the current phase, there is no guarantee that B will write before n changes phases. This example is shown in Figure 5-1 by considering the dashed arc to be a dependency from a previous operator, presuming A to be optional; in this case, B on node 1 can't reuse timeslots used to communicate to node 2 (though a later operator in the same phase might be able to). If, however, both operators A and B are grouped under the same optional marker, A will be permitted to ignore the timeslot restrictions for n, since it will only be running if B is running. Similarly, loop-carried dependencies will generally be ignored, if no dependence will be found when the loop is not followed.

The dependency analysis so far has assumed that the constraining phase precedes the current phase in execution. If the constraint is an already-scheduled next phase, the same analysis is used, but reversed. That is, restrictions on operator A's use of the wire to m can be avoided if, for some operator B also in the current phase, A_n *always* precedes B_m via B'_m.

This analysis essentially allows the presence of earlier operators in the phase to serve as a form of synchronizing barrier between nodes n and m. It would also be possible to clear all of the write-timeslots for the operators in the new phase by adding a 'local' barrier at the beginning of the phase. A 'local' barrier could simply be loopback streams to all the neighbors; the implementation would have to be careful about what timeslots it used, but all the remaining implementations in the phase could safely use all the remaining timeslots. This would be an appropriate strategy when more timeslots were unavailable due to previous phases than the local barrier itself would use.

5.4 Managing a Scheduled Router as a Cache

For a large application, it may be true that the number of subphases required by a given node exceeds the memory capacity of the router. This may be true for either the schedule memory, the VFSM-annotation memory, or both. As a result, phase data must be loaded and unloaded from the router as various regions of the application code are entered and exited. This section discusses how to perform this loading optimally.

Interestingly, this approach simplifies the coordination of multiple programs per node in an application. Rather than distributing separate processor and router code to each node, the router code is embedded in an array in the processor code, and the processor downloads it to the router as necessary.

Throughout this discussion, the router's memory is considered as a *cache* for the complete set of router schedules and VFSM annotations stored in the processor's main memory. (A similar effect would be obtained, if suitable hardware were present in the router, by storing all the schedule information in a dedicated DRAM attached directly to the router, and transferring data into the router's high-speed schedule and VFSM-annotation memory as necessary.) This 'router cache' is managed via software, computing mappings for schedules and VFSMs at compile time and loading the necessary data into the router at run time. A version of the *opt* replacement strategy is computed, including modifications necessary to support optional phases, run-time phase choices, and the requirement that multiple VFSMs be loaded at a time for each phase.

5.4.1 Caching for Router Schedules

Mapping the schedules for the subphases on a given node to router memory is the first problem that will be examined. The available schedule memory is partitioned into *slots*. The amount of schedule memory is divided by the length of a schedule to find the number of available slots. (Typical values for the NuMesh router would be from 2 to 10 slots.)

For a small application (or a large schedule memory size), it may be possible to load everything at boot time. However, if the schedules' total size is more than the space available, points must be picked in the execution graph at which to load new phases' schedules, overwriting the schedules for phases that have already been completed. Preference is given to constructs in loops, to avoid having to do a reload operation each time through the loop.

Cache maps, which map phase data to specific slots in the router, are computed for each node separately. While it would be possible to compute a global mapping from phases to schedule slots, this mapping is carried out on a node-by-node basis to ensure that the limited router memory is optimally used.

However, in a typical application many nodes will share the same set of phases, despite varying spatial extents and the presence of disjoint sets of operators. Accordingly, the compiler tracks the sets of phases for which schedule cache-assignments have already been computed, and reuses those assignments for subsequent nodes with identical phase sets. Thus, if an application has a disjoint group whose two childrens' extents partition the mesh along the x axis, and another disjoint group later in the application whose childrens' extents partition along the y axis, only four schedule cache maps will be computed regardless of the size of the mesh.

For each schedule cache-assignment, an adjacency graph is initially computed for all the phases in the application by walking the operator tree. For each leaf operator in the tree, the compiler iterates through its associated phases to mark which phases each phase follows. If the phase consists of sequential subphases (*e.g.*, parallel prefix), each subphase is marked as following the previous subphase; if the phase consists of multiple phases chosen at runtime, they are all marked as following the previous phase. For parallel operator groups, the compiler recurses down the last child (since the other children are guaranteed to be free of phase changes, as discussed in Section 6.1.4); for disjoint operator groups, it recurses down the branch corresponding to the node subset under consideration.

Sequential operator groups require some special handling. The compiler walks the children, recursing on each child. For children marked as optional, the compiler recurses once to set the childrens' previous phases, then carries on using the union of the initial previous phases with the returned previous phases. This reflects the fact that the phase after an optional phase will have multiple 'previous' phases. Similarly, at the end of a loop, the first child is marked as having previous phases corresponding to the last child.

Dijkstra's algorithm is then used to compute the shortest-path distance between any two phases, based on the computed adjacency graph. Using this data, the compiler walks the phase tree to compute the mapping from subphase number to cache index. As it does so, it maintains a list for each slot of which phase has been assigned there. For each new subphase, each slot is examined, choosing the slot whose nearest phase in the distance graph is furthest (thus, the max of the mins of the distances determines the best slot to use). Ties are broken by using the slot with fewer assigned phases. This technique ensures that every time a schedule is evicted, it's the schedule that will next be used farthest in the future.

In fact, the phase tree is traversed not once but twice. The set of phases is initially divided into 'optional' and 'non-optional' phases. A non-optional phase is one that will be executed every time through a loop. An optional phase is either a phase where all the operators in it are grouped under an 'optional' clause in the source, or it is a subphase that may or may not be chosen at runtime, depending on the relevant 'load' value. The first time the phase tree is traversed only for non-optional phases. The second pass computes mappings for the optional phases, thus ensuring that the common path through the graph is given the best cache mapping. Optional phases are then placed in the best slot given the existing non-optional phases. For example, runtime-choice subphases in a loop will be distributed across the available slots not used by other phases in the loop, and thus, at run time, on some iterations of the loop the desired subphase may still be resident.

5.4.2 Caching for Router VFSMs

Cache-mapping VFSMs is similar to cache-mapping complete schedules. Conceptually, each VFSM for a phase is a distinct entity, but if there are more VFSMs required for an application that are available on a particular node, VFSMs from earlier phases are *reused* (as discussed in Section 1.4.3). Accordingly, 'virtual VFSMs' are mapped to the set of available VFSMs in each phase; the term ‘slot’ is used to refer to each VFSM resource just as it was used for each available schedule location in schedule memory. The cache-mapping is computed for each VFSM separately, but the compiler has to be able to guarantee that all the VFSMs for a particular phase map to different slots, so there are no conflicts during a phase at run time.

To accomplish this, the compiler attempts to cache-allocate all n VFSMs of a phase at once using a ‘greedy’ strategy. First, however, all the continuing operators in this phase are identified, and the compiler examines the cache map to find and reserve the slots to which their VFSMs have been allocated. Those VFSMs do not need to be explicitly reloaded in following phases, since all phases including a continuing operator are guaranteed to consecutively follow the phase that initially loads them.

Each VFSM in the phase picks a cache mapping much as is done for schedules in Section 5.4.1 above, by finding the slot whose next VFSM to be mapped is farthest away; that slot is then marked as used so that subsequent VFSMs do not try to map to it. Additionally, all the phases that have a VFSM mapped to the chosen slot are marked as ‘excluded’; these phases will then be ignored when examining distance for subsequent VFSMs in this phase, with the result that the fewest number of phases should require subsequent reloading.

Since the hardware may support multiple pipelines, cache-allocation must be performed for VFSMs for each pipeline. The compiler can take advantage of multiple pipelines to perform more aggressive VFSM cache mapping by permuting the pipelines in each phase as part of the cache allocation. This is straightforward, since each pipeline is independent. Let us define cost $c_{i,n}$ to reflect adding n VFSMs to pipeline i with an existing set of VFSMs from previous phases already cache-mapped. By suitably permuting the pipelines in which the phase’s VFSMs $n_0, n_1, ...$ are placed, the total cost can be minimized; for a pipeline permutation $p_0, p_1, ...$, the compiler minimizes $\sum_i c_{p_i,n_i}$.

Some pipelines on a given node may be carrying VFSMs from an operator in a previous phase; these VFSMs must remain fixed in the same pipeline, which in turn fixes the new VFSMs in this phase that were scheduled around the continuing operator’s VFSMs. However, the compiler uses any remaining flexibility; for example, if there are more than two pipelines in each node, and a single pipeline is constrained by the presence of a continuing operator, the compiler still explores the possible permutations of the remaining pipelines.

A possible future extension to the VFSM cache mapping would be to consider VFSM aliasing. VFSMs that had been present in previous phases could be reused without reprogramming in the next phase if they had the same annotations. Similarly, within a phase a VFSM could be reused if it was used by sequential operators. And, VFSMs in related subphases (*e.g.*, from runtime-subphase meta-implementations), which are more likely to share the same VFSMs than unrelated phases, could be aliased together. Currently such aliasing is not performed, since most VFSMs have different annotations; and in any case most of the overhead in cache management comes from loading the schedules, which are not good candidates for aliasing.

5.4.3 Runtime Cache Management

For each cached item (schedule block or single VFSM), the compiler computes the set of other items that share the same cache mapping. This allows the generated COP code to track the state of the cache on the processor as it runs, by marking each item as cache-resident or cache-nonresident, then unmarking relevant other phases when new phases are loaded into the cache.

Each pipeline's VFSMs, as well as the schedule data, is handled separately; when one is found to be unavailable at load time it is downloaded to the router. This form of runtime mechanism is required in the face of optional phases and subphases selected at runtime.

A further extension to the compiler would be to compute cache occupancy at compile time where possible and omit the runtime checks for those cases. Additionally, for loops the compiler could hoist any cached items that have no conflicts within the loop to the loop's own load function, if available. Finally, loads for a sequential series of phases could be chunked together if they were found not to conflict in the cache. Furthermore, when it is necessary to reload VFSMs for a phase, it may only be necessary to reload some rather than all the VFSMs in a pipeline. For now the compiler does not do such analysis, since the runtime checks are relatively cheap.

An additional feature that would potentially reduce phase-switch time would be to take advantage of DMA where available; currently the compiler has no support for such a feature. At the start of a phase, a DMA request could be issued to write the next phase's VFSMs and schedule into the router, thus hopefully converting every phase change into a single change-schedule command, with the overhead of actually loading the router hidden at times when the processor interface was otherwise idle.

5.5 Finding Load Operators

The last issue involving phase management is which chunk(s) of VFSM/schedule code to associate with which COP operator load function. In a simple program, with one operator per phase, and all operators of the same extent, one chunk of VFSM/schedule code could simply be associated with each load function. In more complicated situations it becomes more difficult.

The situation is simplified slightly by placing some restrictions on where the application program may legally place the returned load functions within the compiled code, along the lines presented in Section 2.7.2. For an operator appearing within a sequential group, the load function must be placed after the last I/O of the previous operator and before its own first I/O. For a parallel group, all the load functions must be placed before the first I/O of any operator in the group, and the load functions must be called in the same (arbitrary) order as the order that the operators were specified in the COP code. Similarly, loop load functions must be placed before the loop, and after any I/O associated with operators which run sequentially before the loop. Load functions for operators within a loop must be placed such that they are executed each time through the loop. Finally, all nodes in an operator's subset must issue the load, even those nodes that will not be performing I/O using the operator in question.

5.5.1 Choosing a Load Operator for a Phase

Whenever there are multiple operators in a phase, the compiler must determine which operator will be the 'load operator' that is responsible for loading the code for the phase. There are various issues to consider:

- The router code must have been loaded before any I/O functions are called.
- The router code can't be loaded until all run-time choice decisions are made as to which subphase to load.

Recall that COP requires that, for a `parallel` construct, all the load functions for the involved operators be called before any I/O operators. However, with `sequential` constructs, the load function(s) will be called between each child of the `sequential`. As a result, the compiler iterates through the operators in the phase (in program order), and stops on the first operator that has a sequential relationship with the following operator. This operator is marked as the load operator.

To handle the case of n multiple runtime-choice operators, COP requires that the first $n-1$ such operators simply set a global variable corresponding to which of their choices they wish. The load operator then combines all the values appropriately to get a single subphase index, which determines which subphase it loads.

The above description is a simplification; it assumes that all the operators in a phase have the entire extent of the phase as their subset. Consider the case where this is not true: for example, consider a 2D mesh where some phase consists of a sequential group with a broadcast on column zero followed by a broadcast over the whole mesh. In this case, the nodes in column zero will make the column broadcast the load operator, and the other nodes will make the mesh-wide broadcast the load operator.

As a result, when determining which operator is the load operator on a given node, the compiler must also make sure that the operator's subset actually includes the node; if not, the operator is skipped. The algorithm to compute the load operator for a given phase and node is thus to walk the operators in the phase, ignoring ones whose subset does not include the node, and stop as soon as it comes to an active operator that stands in a sequential relationship to the next active operator.

5.5.2 Associating Multiple Phases with a Load Function

What happens when *none* of the operators associated with a phase are defined for a particular node? Consider, for example, the case of red/black broadcasts on a 2D array: one operator has the even nodes in its subset, and one operator has the odd nodes. Suppose that each operator is in its own phase, terminated by a barrier, with the 'red' broadcast occurring first. In this case, the black operator's load function (run on the black nodes only) must load the code for the red broadcast, then participate in the barrier for the red broadcast's phase, and only then load its own router code.

Meanwhile, as the application moves into the second phase, the red nodes have finished their only operator. This seems like a problem, since there is no 'hook' to hang the load of the black operator's phase: the application will not be executing any more operators on the black nodes! Accordingly, COP defines a function that is called on all nodes at termination time, `cop_fini()`. This function will handle loading the red operator's phase code on the black nodes before terminating. (A similar function, `cop_init()`, is called on all nodes before calling other COP functions; it handles any COP-specific setup operations.)

In fact, if the operators have sufficiently irregular subsets, a particular operator's load function on a given node may end up loading several phases, one after another, and running the appropriate barrier function between each one. (As discussed in Section 5.2, a phase where some operator's subset does not cover the phase requires a barrier.)

Thus, the final step in the algorithm to find the load operator for a given phase and node is as follows. If an operator can't be found in the phase to associate the load with, the compiler walks through the operator graph looking for the first operator that does include the node, and schedules the load to be performed by that operator.

Chapter 6

The Compiler Search Engine

This chapter presents a sketch of the techniques used by the compiler to convert the arbitrary hierarchy of parallel and sequential grouping of operators into a hierarchy of scheduled phases. As part of that process, it examines how best to handle multiple disjoint threads of control in an application. Section 6.1 discusses how to convert the COP input into disjoint sets suitable for supporting multiple threads of control, and Section 6.2 discusses the search algorithm that jointly optimizes for implementation and phase boundaries. Unlike previous chapters describing techniques only suitable for reprogrammable scheduled routing, much of this material is applicable to any compiler backend. The chapter closes with a section discussing how the chosen implementations are converted to schedules and VFSMs for scheduled routing.

6.1 Multiple Threads of Control with Scheduled Routing

Most of the introductory discussion implicitly assumed that more than one operator is never active at once on different parts of the mesh. Doing so simplifies the issue of which nodes may be involved in routing operators' communications—any node can. For example, a stream operator may choose a roundabout path from the source to the destination in a phase where simple e-cube type routing would lead to contention in an online router. This section examines the issues involved in restricting the set of nodes that an operator can use to route its data.

6.1.1 Restricting the Set of Nodes for Routing

Let us consider an abstract example application where two portions of the mesh are executing independent sections of code, or perhaps executing the same sections of code but in an unsynchronized manner; COP code representing this behavior is as given in Figure 6-1. There are a variety of ways that the COP compiler targeting a scheduled-routing backend could choose to handle such a scenario.

First of all, consider the case that each operator is transferring a very small amount of data; in that case, it may make sense to simply schedule all six operators into a single phase. Most of the routing for the A operators will presumably stay on the left, and similarly for the B operators, but if necessary the router can use left-side nodes to improve the routing of the B operators, and vice versa.

```
(parallel
  (subset '((...))     ; left side of mesh
    (loop 'A 1000
      (op A0 ...)      ; "op" represents any operator
      (op A1 ...)
      (op A2 ...)))
  (subset '((...))     ; right side of mesh
    (loop 'B 1000
      (op B0 ...)
      (op B1 ...)
      (op B2 ...))))
```

Figure 6-1: Simple abstract multi-threaded code example

However, consider what happens when each operator is transferring larger amounts of data. In that case, the phase-switching overhead is amortized by the time required to route the data, and the optimal solution (most likely) is to have each operator running in a separate phase. Now it becomes more of an issue whether to allow the router to make use of nodes in the wrong half of the mesh for routing operators' streams. A straightforward approach (and one that is followed up below) is to assume that since the operators are active on different parts of the mesh, they can simply be restricted to their sides of the mesh. The stream router can be configured to disallow routes outside of the appropriate side of the mesh, and the two sides of the mesh can run completely independently.

Now let us imagine that operator B1 can achieve somewhat better bandwidth by using a few nodes from the left side of the mesh. Perhaps operator A1 isn't using those nodes much anyway, so bandwidth is available, and using them might decrease the overall communication time. However, since the A loop runs independently to the B loop, it is not sufficient to schedule the extra bandwidth for B1 just in A1's phase; the same VFSMs must be scheduled to run at the same time during A0 and A2's phases as well, or else B1 will lose portions of its connectivity when the A loop changes phases. Assuming b bandwidth could be added to B1's phase, *all* of the A operators must be disallowed from using that bandwidth. If there are k operators in the A loop (3 in the example above), that effectively costs the A loop kb in available total bandwidth, b for each A operator. Only if fewer than k^{-1} of those slots would otherwise be used by the A operators does the tradeoff make sense. Given that the COP compiler generally tries to optimize phases to make full use of the available routing, this is unlikely to be true.

So far only the possibility that just one of the B operators might need nodes outside its half of the mesh has been considered. Once any of the A *or* B operators might do so, things become potentially even worse. Each operator might need to allow some of the bandwidth of the nodes on its side of the mesh to be scheduled away to *each* of the operators on the other side of the mesh. Each phase p could add b_p bandwidth to itself only at a collective cost to the operators on the other side of the mesh of kb_p.

Furthermore, bear in mind that if an A operator takes bandwidth from a few nodes on the B side, the B streams that use that node will run more slowly, with less bandwidth available to them. Since overall performance is typically constrained by the bottleneck stream for a given operator, this will result in, effectively, all the B operators running more slowly just to make available a small amount of bandwidth to some A operator. Only if the shared nodes are underutilized to begin with can this situation be avoided, and as pointed out above, the COP compiler normally tries to avoid underutilized nodes in any case.

Additionally, supporting this approach requires a rapid growth in the complexity of the router path-selection code. To route A1's phase, for example, the compiler must know what bandwidth has been reserved for any of the B operator's phases; but to route a B operator, the compiler must know in turn how the A operators were routed. This circular dependency can be broken in a number of ways, such as greedily routing operators in the order they are given in the program, and accepting any resulting unfairness in bandwidth allocation, or perhaps by some form of iterative approximation, but it adds substantially to the compiler complexity.

This conclusion suggests that the compiler can reasonably ignore the case where donating bandwidth from multiple phases to some other phase may *improve* application performance, and concentrate on supporting the common case, where such overlapping usage has a deleterious effect on application performance. Accordingly, operators must maintain their independence when appropriate.

To this end, the notion of operator *spatial extents* are introduced (also known as 'keep-ins' in the VLSI routing community) along with the notion of a compiler-generated `disjoint` grouping construct. The first section below examines how to reasonably restrict the set of nodes that an operator can use for its communications; the next examines how to restructure the COP program, using spatial extents, to specify multiple independent sections of the mesh.

6.1.2 Deriving Spatial Extent Information

Given an operator, how can the compiler choose an appropriate *spatial extent*; that is, a set of nodes which can be used for that operator's communications? The COP code for each operator contains subset information on where that operator performs I/O. However, the subset information may not correspond to the set of nodes that must be involved in managing the communications: for example, the subset may be sparse (*i.e.*, nodes in the subsets may not be neighbors), or the subset may have an irregular shape with good communication paths between some nodes using paths across nodes not in the subset. And, as mentioned above, sometimes the presence of other operators suggests the use of indirect routes for improved bandwidth.

It might seem reasonable to have the application provide information on spatial extents to the compiler, rather than have COP derive the extents. However, doing so would require the HLL compiler to know enough about the characteristics of the topology and the router that it could specify an appropriate spatial extent for the operator; as discussed below, not all topologies have obvious extents. As this is a communications-level decision, it is appropriate to leave it to the communications compiler. In particular, the language is cleaner if information is specified only on the endpoints of communications, and the details of mapping the operators are left to the communications compiler.

In general, deriving information on where a communications operator may best route its data is subject to simultaneous optimization. A larger spatial extent for an operator may allow better misrouting under contention. On the other hand, a smaller extent may allow adjacent groups of operators to run independently of each other, since each node is dedicated to routing entirely in one group or the other, but not both.

For COP's target, rectangular 3D Cartesian meshes, it is natural to consider the spatial extent of an operator to be the smallest 3D box that can contain its subset. A variety of alternate extents could be imagined: a border of one or two nodes could be included in each extent, or extents could follow more precisely the boundaries of an irregular subset, or perhaps extents could be computed by performing an early routing step and examining the resulting paths. However, the extent model used here is easy to compute, easy to manipulate, and offers reasonable internal routing opportunities while not causing undue interactions between physically adjacent operators.

Computing spatial extents for other network topologies than Cartesian may not be so straightforward. In particular, the diamond lattice topology has no similarly obvious definition of extents to use, since restricting operators to minimum-distance paths would dramatically increase the contention in the network. With such a topology, it would almost certainly be necessary to use larger extents, with the result that it would be substantially harder to divide operators into disjoint sets that could be handled separately. As discussed further below, this would make implementing separate threads of control within a program much more difficult.

It is due to such compiler-specific decisions on determining appropriate spatial extents that the COP language only includes high-level information on whether operators run in parallel or sequential, rather than requiring the high level to know when operators may truly be occurring on disjoint sets of nodes.

Once a set of operators have been chosen for a phase, the compiler can, in fact, be more liberal in the set of nodes used to route those operators. Rather than using only the extent of the operator to route the operator, the compiler takes the union of all the extents of operators in a phase to be the *phase extent*, and routes all operators within that extent. (Strictly speaking, continuing operators are not included when determining the phase's extent, as discussed in detail in Section 5.1.)

6.1.3 Restructuring COP Parallelism with Spatial Extents

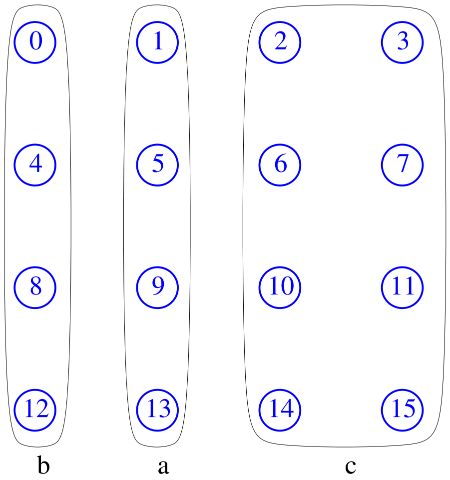
Once spatial extents have been determined for operators, it is necessary to consider how these extents combine. For example, if there are two operators that are marked to execute in parallel (or, in fact, sequentially), it may be that the operators have disjoint extents. In this case, they can be handled completely independently, as discussed at the start of this section.

To determine the spatial extents of operators and groups of operators, the compiler starts from the leaves of the operator tree and works its way up. Each operator instance (corresponding to one subset in the COP input), is assigned an extent by COP. The extent of the operator itself is then generated by taking the union of the extents of the operator's instances, and the extents of each group of operators is recursively generated as the union of all their children in the operator tree. The extent of the root of the operator tree gives all the nodes that will hold router code in the COP output.

With the spatial extents of all the nodes in the operator tree in hand, the compiler can attempt to determine which operators or groups of operators are disjoint from which others. The result is a rewriting of the tree from an input that holds simple `sequential` and `parallel` groups to one that also holds `disjoint` groups (and may be somewhat restructured as well). This restructured operator tree allows the compiler to generate better output.

Restructuring Parallel Groups

Parallel structures are most likely to be converted to disjoint structures, since they may often be specifying operators in different parts of the mesh that are occurring at the same time. For a given parallel operator, the compiler starts by optimistically converting it to a disjoint operator. Then it walks the children. For each child its spatial extent is tested against the extents of all the previous children; if it overlaps, they are combined into a single parallel group. Take for example a simple case on a 4x4 grid, in Figure 6-2.




```
(parallel
  (subset '(1 5 9 13)              ; column 1
    (cshift 'a 0))
  (subset '(0 4 8 12)              ; column 0
    (broadcast 'b 3))
  (subset '(2 3 6 7 10 11 14 15)   ; columns 2-3
    (stream 'c (runtime) (runtime))))
```

Figure 6-2: Simple disjoint parallel example

After converting, it is the same structure, but a disjoint group. If the subset of c was changed so that it included node 1, the result would be a disjoint group holding two children, one being the b operator, and one being a parallel group holding a and c. Similarly, if b's subset were changed to `'(0 1 2 3)`, the transformation would yield a single parallel construct.

Restructuring Sequential Groups

A sequential group seems an unlikely candidate for conversion to a disjoint structure, but it can happen. An application may know that it will perform several operations on disjoint subsets sequentially, and accordingly use a sequential construct. This leaves room for the compiler to generate barrier constructs and run each operator on the whole mesh one at a time, if doing so is optimal, or to run each in a disjoint set, if allowed by the spatial-extent overlap. When doing the latter (that is, converting a sequential group to a disjoint group), the compiler is explicitly decoupling the sequential operators, such that they run strictly in parallel.

Sequential groups are converted to disjoints by just walking the children. If each child's extents overlap with that of the child before, nothing is changed. On the other hand, if they do not overlap, the previous child is converted to a disjoint group containing both children. More following disjoint sequential children can then be added, or if later children's extents overlap the disjoint group's, they continue to be simple sequential children.

In the case of the example above, if the `parallel` were replaced by a `sequential` this rewriting would still generate a `disjoint` from the result. Adding node 1 to c's subset (as was done above) would result in a sequential group of two children, the first being a disjoint group holding a and b, and the second being c by itself.

6.1.4 Sequential Descendants of Parallel Groups

Once these transforms have been carried out, the result may be an operator tree that contains disjoint sequential groups, each of which can be executed independently. However, if multiple sequential groups' spatial extents require that they be executed in parallel, things become much more complex. As a result, in the current implementation each parallel group must have at most one child with sequential descendants. (In fact, the current compiler requires that it be the last child.) This is equivalent to saying that all operators after which a phase change is performed must have nearest common ancestors that are sequential or disjoint groups. In other words, for a specific node, all such operators must be fully ordered in time, and any parallel ancestors just contribute 'continuing operators' that are included in each phase.

There are several reasons that this is required. First, for simple subsets, it is unnatural to specify several sequential lists to run in parallel, since that suggests that the high-level compiler does not know anything about the relative timing of the sublists. A more natural formulation would be to place them all in parallel if no mutual timing information was known, or to make use of a sequential list of operators and parallel groupings of operators if some timing information was known. Second, for interleaved subsets, if the subsets are performing different lists of actions at the same time using this construct, it will be necessary to merge them all into a single parallel case regardless, since the COP model does not support nodes changing router code on neighboring nodes' routers. (By comparison, for disjoint subsets it makes sense to have sequential sublists, since they may be performing completely unrelated code for some period of time.) Third, this construct would require substantial complexity in the compiler to handle switching among the cross product of all the phases.

Since this case is so obscure, the current implementation requires the front end to rewrite the input such that only the last subtree of parallel group has any sequential descendant groups. As discussed above, the process of converting parallel and sequential children to disjoint groups will remove some apparent problems of multiple sequential descendants of parallel groups.

It would be possible, of course, to get much more complicated in the compiler, and do multiple passes over the input, each time with different 'virtual last parallel children', for each parallel structure. Then the compiler would require descendants of all other children to suppress phase breaks, resulting in a single parallel phase. This was deemed unnecessarily complicated given the rarity of parallel nodes with multiple children with sequential descendants.

6.2 Searching the Implementation Space

Previous sections have alluded to the issues involved in choosing phase division points, picking optimal operator implementations, and finding disjoint portions of the communications program. This section discusses how to search the 'implementation space' to find the best selection of phases, and how to represent the result to include non-overlapping operators.

To generate a set of phases from a tree of operators, the compiler has to recursively inspect the tree looking for sequential groupings which it can use to create multiple phases. Phases are divided by *phase breakpoints* that are placed between two children of a sequential group.

6.2.1 The Search Model

The compiler scans through the COP program by doing a path-based branch-and-bound search technique. It starts with a *search node* representing "the whole program", holding a pointer to the root of the COP program. Search nodes are kept on a priority queue, and the compiler pulls them out in an ordered manner to extend the search space.

Each search node corresponds primarily to an under-construction *phase tree*. The phases are grouped into a tree during the search, with internal nodes corresponding to sequential or disjoint groupings. A COP `sequential` construct may or may not map exactly to a phase tree sequential group, since some merging of sequentialized COP operators may occur in the phase tree for a given search node. Phase structures also hold the predicted time to execute themselves (or their subtree in the case of phase groups). Each phase structure also holds the number of subphases required by the phase or phase subtree.

Each leaf node in the phase tree holds a list of operators included in the phase, along with relevant information for each operator, such as the implementation being considered for the operator and some informative data about performance with this implementation on this operator. Finally, the compiler keeps track of some operator-specific flags.

The search node also holds other information. A pointer is kept to the current operator in the COP node to be expanded; this pointer is used to walk the COP program, and when a search node's *operator pointer* reaches the end of the program the search node represents one complete implementation of the program. Another pointer is kept into the phase tree; this represents the current *active phase*. Operators may be merged into the active phase at each step, or the compiler may *close* the active phase, and create a new active phase to put operators into. Other information stored in the search node includes, among other things, a list of chosen *replacements* of various operators with other operators (as discussed below), as well as a list of operators that provide the current parallel context for each new phase.

6.2.2 Structuring the Phase Tree

Nodes are added to the phase tree as the compiler traverses through the COP program's operator tree. These represent both chosen phase breakpoints (for children of sequential phase groups) and MIMD operations (for children of disjoint phase groups).

Sequential and Parallel Groups

As the compiler walks the tree, if it traverses from one child of a sequential group to the next child, it may choose to try closing the current active phase. It does this by placing both the closing and non-closing alternatives into the priority queue as separate search nodes. Closing a phase is itself discussed further below.

For a parallel group, the current COP definition requires that only the last child may have sequential descendants (as discussed in Section 6.1.4). When the compiler comes to the last child, it pushes the current active phase's operator list to the top of the *parallel context* stack in the search node. Since there are guaranteed to be no phase closures in the other children, there is only one unclosed phase in the tree, and it is currently active. The current parallel context is then used in the group's descendants to initialize each newly opened active phase. When the compiler finishes traversing into the last child, it pops the parallel context item off the search node's context stack and continues.

Disjoint Groups

Disjoint groups are more difficult to handle. When the compiler first encounters a disjoint group in the operator tree, it immediately creates a phase-tree disjoint group following the current active phase, which it leaves unclosed. As it processes each child of the disjoint group, the compiler creates a child of the phase tree disjoint group and sets the active phase to point to it. As the compiler recurses down the operator tree, that active phase may in turn be split into further sequential groups, and indeed further nested disjoint groups.

The complexity comes when the children of a disjoint group are finished. The compiler removes the first sequential phase from each child of the disjoint and bundles all their operators together into a single set. This set is then placed in a phase immediately before the disjoint phase group—either the existing previous phase (if it wasn't closed), or else a newly created phase. Either way, the result is a single phase holding all the operators active at the start of the disjoint phase. The compiler then closes the combined phase (as discussed below). Combining the first phase of each disjoint section allows the compiler to handle terminating the combined phase with a single barrier if necessary, which may be required if operators spanning the disjoint group are ending in this phase.

In the second part of the disjoint-group close, the compiler takes the *last* sequential phase from each disjoint group's child. These phases are removed from the disjoint, and their operators are collected into a new phase, following the disjoint phase group. This phase now becomes the active phase. If the disjoint group is a child of a sequential group, the compiler handles forking the search node into close/noclose options using the aggregated active phase in the usual manner. As mentioned at the end of Section 5.2, if a barrier is necessary it will be configured to operate independently over each disjoint subset.

The complexity of the disjoint handling not only allows correct behavior under certain circumstances (such as terminating operators that cover multiple disjoint phases, as mentioned in Section 5.2.3), it also simplifies other parts of the compiler. For example, even in the presence of disjoint operators, stream routing can be done on single phases rather than needing to collect several disjoint phases at once. Similarly, the 'active phase' remains a single phase at all times, rather than expanding to become a disjoint subtree of active phases, as would be the case if dis-

```
(sequential
  A
  (disjoint
    (sequential B C D E)
    (sequential F G H))
  I)
```

Figure 6-3: COP disjoint example

```
(psequential
  A,B,F
  (pdisjoint
    (psequential C D)
    G)
  E,H
  I)
```

Figure 6-4: One possible way to handle disjoint COP

joint children were not pulled out into a single phase at the end of a disjoint construct. Finally, this algorithm results in the first phase listed for an operator always temporally preceding all other phases listed for that operator.

As a simple example, consider the COP program in Figure 6-3. Let us consider a search node where the compiler closed after after all the sequential children except A. After converting the operators into phases and applying the disjoint closure algorithm above, the result is the set of phases shown in Figure 6-4; ‘psequential’ and ‘pdisjoint’ refers to phase-tree grouping constructs.

As another example, disjoint groups in the operator graph may be lost altogether after conversion to phases—for example, if the application contains two intersecting disjoint sets such as rows and columns of a 2D array. In that case, the user might input

```
(parallel rowop1 rowop2 ...  rowopn colop1 colop2 ...  colopn)
```

The initial disjoint generation (as discussed in Section 6.1) would result in an an internal form with two disjoint groups under a parallel operator. As the compiler evaluated this form, the first disjoint group would resolve into a single phase holding all the `rowops`. When the compiler moved into the last child of the parallel the `rowops` would be moved onto the parallel context stack, and pushed onto each child phase of the disjoint group as it was evaluated. Popping out of the second disjoint would then combine all the children (taking care not to duplicate the n `rowops`, one from each child) into a single phase, thus eliminating all traces of the intermediate disjoint group.

6.2.3 Picking Implementations for Phases

The structure of the phase tree is only part of the story. It is also necessary to determine which implementations to use for all the operators. Taken together, these two can be used to predict the overall execution time for the application. A wide variety of operator implementations are available, as discussed in Section 4.1 and enumerated in Appendix B. This section discusses what restrictions are placed on their use.

Schedule-generator implementations can be used only when they are the only implementation in a phase. Since starting a new phase with non-empty parallel context immediately adds things to the active phase, schedule generators can't be used in that case. Similarly, after adding a schedule generator the phase must be closed immediately. Accordingly, for a given operator, the compiler determines whether it can close after the operator by walking up the tree; if it finds a sequential ancestor with a later child, it can close, whereas if it finds a parallel ancestor with a later child, it can't close, and thus can't use a schedule-generator implementation.

There are a few other situations where the compiler also ignores implementations outright. When compiling for dynamic routing hardware, it ignores any implementation not marked with the 'dyn' flag. Users are also allowed to force a given implementation, in which case the compiler ignores all the other implementations for that operator during the search.

Once the inappropriate implementations have been eliminated, the *metric* function for each implementation is called. If the metric function returns true, a new search node is created that binds the given operator to that implementation. The result is a set of new search nodes, one for each applicable implementation.

Replacement implementations are handled specially. For them, the compiler simply creates a new search node that adds the old operator and the new operator(s) as a pair to the replacements list in the search node, then places the search node back on the priority queue, with its operator pointer now pointing to the modified graph. Since the compiler normally only walks forward in the operator graph, jumping to a partially-attached fragment of the operator tree poses no problems to the search code, since it makes sure the 'next' and 'parent' fields link the new operator(s) to the rest of the operator tree.

6.2.4 Closing a Phase

Closing the active phase involves computing the cost of the phase and adding that to the total cost for the search node so far.

Normally the compiler chooses when to close purely by implementation requirements and the COP program structure. One additional constraint for scheduled routing is that for each phase there is a hard limit on the number of streams that can terminate in or cross through the node, equivalent to a little less than twice the schedule length. The compiler has exact information on how many streams terminate on each node, and it can guess at how many streams cross nodes by looking at the average length of the streams in the phase. Any search node that accumulates so many operators that it violates this criterion is rejected.

Once the decision to close a phase has been made, the compiler figures out whether a barrier is needed to allow the close. Barriers are not needed for the dynamic-routing substrate, nor if a schedule-generator implementation is being used. The issues involved in selecting a barrier scheme are discussed in Section 5.2. If the compiler chooses to add an explicit barrier operator to the phase, it recursively invokes the search algorithm to find the best implementation of a `barrier` operator type for this phase, then continues with the top-level search.

Every time a phase is closed, a predicted total time for the communication in that phase is computed (as is examined in the following section). That predicted time is used to update the parent nodes in the phase tree. Eventually some search node traverses the whole COP input; this yields a predicted time for all the application's communication. The compiler updates that bound as other search nodes reach the end of the COP input, and uses the bound to prune other search strategies along the way if it finds that they are already predicting a longer time than the best current bound. While somewhat slow, this solution guarantees that the compiler will find the best solution; and since this thesis is more interested in how well COP and scheduled routing can do than on implementing the best compile-time/run-time tradeoff in the compiler (a thesis in itself), this implementation has been chosen.

There are some obvious alternate techniques. One is to independently choose the locally-optimal set of operator implementations every time a phase is closed; however, this runs a risk of locking in some subphase-intensive implementations early on that preclude more rewarding subphase-intensive implementations later on. Similarly, it becomes harder to pick the best implementation for operators that extend across multiple phases; tentative assignments could be done, then a back pass at the end could try to rework them, but this might miss out on more extensive implementation changes that would create the best results.

Another alternate technique would be to use simulated annealing on a single phase tree, switching implementations and modifying which sequential children are phase-merged. This method might produce good results, but again does not guarantee optimality.

6.2.5 Computing Timings for an Implementation Set

Regardless of how the space is searched, once a phase is closed the compiler must estimate how long the set of implemented operators will take to execute when the program is run. The implementation-specific timing information collected by the calls to the *metric* functions for the operators in the phase is used to compute this value. The timing is computed as the sum of three components: the sum of the I/O time predicted for all the operators, the time taken by the phase change, and the time to perform any router programming operations. Using this information, the compiler eventually chooses the implementations and phase breakpoint strategies that result in the lowest approximate total run time.

Computing I/O Time

First, consider how much total time is spent in the I/O function(s). The first question to answer is how much time a single I/O operation takes. Bandwidth is measured as 'inter-word time', which is the number of cycles between the time when successive words can be sent to the router. Thus, an inter-word time of one yields maximum bandwidth. Using these definition, the total time for an I/O operation can be given as a simple equation, $wB_{io} + l_{io}$, multiplying the number of words to transfer by the I/O bandwidth, and adding a latency factor.

The implementation-specific information returns approximate values for two or three of these unknowns. Normally the w value is simply the message size specified by `:m` in the operator definition, but more complex operators may require multiple messages to compute a value, and use a correspondingly higher value of w. The l_{io} value, similarly, is usually the distance through the mesh, but may be higher if appropriate. Schedule-generator implementations, since they have complete control over the global schedule, can provide precise values of the inter-word time, B_{io}. Stream-alias implementations, on the other hand, do not know their final inter-word timing, since it depends on the presence of other streams in the mesh. The inter-word time for stream-alias implementations is estimated using a metric that is discussed later. When computing bandwidths, the processor interface's maximum bandwidth, B_{if}, is used as a limiting factor on B_{io}; thus, if the interface is slow, the compiler does not favor an implementation with which the processor can't keep up.

Normally, the implementation's latency value is added to the predicted time. However, some operators may be used by the application not for round-trip messages, but rather as a way of overlapping communication and computation. In this case, the compiler should not worry about optimizing the latency; while latency can still play a role in application timing, it need not be considered a first-order effect. If the COP input specifies '`:pipelined t`' as an optional argument for an operator, the search engine omits the l_{io} term returned by the implementation when computing predicted I/O time.

Having found the time for a single operator, that value must be multiplied by the number of times the I/O function will be called. COP can specify a '`:c` c' message count value that gives an approximation of the number of times the operator's I/O function will be called in the inner loop. Then, the compiler determines how many times the inner loop will be run, multiplying the loop counts of enclosing `loop` constructs, to get a total loop count, L_{tot}. Multiplying these two values together gives a total predicted I/O count for the operator.

The 'callout' functions used in operators such as `reduce` and `prefix` are purposely ignored in these computations. The callout functions (*e.g.*, 'add' for an add-reduce) increase the total time required for the operator. However, since typically the reduction trees are of roughly similar shape for all the implementations, ignoring the callout function time does not incorrectly favor one implementation over another. (Future work might include an operator annotation giving the function time.)

An additional factor may increase the time for certain operators. If more streams are coming to or from the processor than the interface address space, it may be necessary to multiplex multiple streams into single addresses in the interface. To accomplish this, the router can be reprogrammed to 'hold' the stream currently using a given interface address (*e.g.*, by changing its last VFSM's destination annotation to make it deliver to nowhere), and take another stream 'off hold' by arranging for it to deliver to the interface address in question. The reprogramming step should be accounted for in the predicted timing. Assuming that performing k reprogramming actions takes time kB_r on average, and that two reprogramming actions (for example) are needed to change an address's stream, an extra penalty of $t_{iface} = 2cb_r$ is assessed to each operator multiplexing an interface address (for each one's particular value of c). Operators not sharing an interface address have $t_{iface} = 0$.

Thus, the total predicted I/O time is found by computing the per-operator time given appropriate values for c, L_{tot}, w, B_{io}, and l_{io}, then summing the time for each operator in the phase:

$$T_{io} = \sum_{ops} cL_{tot}(wB_{io} + l_{io} + t_{iface}) \tag{6.1}$$

Computing Phase-Change Timings

Each time the COP code changes phases, some overhead cost must be paid, which must be modeled in the cost function. Rather than modeling the cost to enter the phase, the cost to exit the phase is modeled, since that is where any barriers for the phase are performed.

The first step is to consider the barrier function needed for the phase. The techniques described in Section 5.2 are used to choose an appropriate barrier and model its cost here. For a message-synchronized phase, the length of time needed to pause is given. For a hijack-synchronized phase, the minimum bandwidth of the spanning tree and the computed length of the message are used to determine how long it takes to send a synchronizing message, plus a few cycles to allow the final data words to leave each node's router. For a barrier operator, the required time is given automatically, since the operators that make up the barrier are treated as any other operator. Finally, for the no-op case there is no time penalty. The length of time for whichever barrier is used is represented as t_{barr}.

The actual switch to the next phase is then modeled. This analysis assumes that the router already holds the information for the new phase; time to load the phase is factored in later. Switching to the new phase requires some additional overhead, based on the speed with which the router can be reprogrammed. Assuming the switch can be done by passing a single command to the router, the reprogramming takes another B_r cycles to perform.

A few additional cycles need to be added to the barrier computation for all but the no-op barrier. These cycles are used (as discussed in Section 5.3) to ensure that each node's neighbors have switched into their next phase, before allowing the processor to begin using the new phase in the router. The number of cycles is given by the implementation-specific information for message-sync phase, computed for hijack-sync, hard-wired to a small constant for explicit barriers, and zero for no-ops. These synchronizing cycles are called t_{sync}.

The phase is typically rescheduled a number of times during the course of the application's execution. The exact number is determined by the number of loops that enclose the phase. Some loops may be internal to the phase itself; for example, there may be a two-operator inner loop where both operators are scheduled into a single phase. The compiler must determine how many loops are nested within the phase and discount those. Some loops may be partially included in the phase; *e.g.*, if the first two operators of a three-operator loop are in one phase, and the other operator in a separate phase, the phase must be scheduled once for each pass through the loop. A phase may even need to be scheduled over multiple loops; for example, a phase may hold the last operator from one loop and the first operator from another. The total number of loops that the phase is present in but does not fully include is called L_{ext}.

Thus, the total predicted scheduling time is:

$$T_{sched} = L_{ext}(t_{barr} + B_r + t_{sync}) \tag{6.2}$$

Computing Router-Loading Time

Another contribution to the total application time comes from the time spent loading the phase into the router.

The number of words that are required to program the router will vary depending on the details of the implementation. The schedule is likely to be the dominant portion of the router programming information, since each VFSM is usually listed multiple times per schedule. (Schedule generators can give an exact worst-case value on number of VFSMs used, and the stream cost metric can also come up with rough estimates of VFSM use.) If the schedule is length s, and, *e.g.*, it takes one word per schedule entry, it would require (at least) s words to pass to the router to provide the information for the next phase. The number of words necessary to reprogram the router is called w_{prog}.

So far, router reprogramming has been assumed to happen at a bandwidth of B_r. It may be possible for a given implementation to reprogram the router at a higher bandwidth; for example, if reprogramming is handled out of the schedule, briefly switching to a schedule that does *only* reprogramming may allow for higher bandwidth at some cost in overhead from the initial phase change to the 'reprogramming phase'. This is represented by using B_{rbulk} as the average bandwidth to reprogram the router for larger reprogramming, with a l_{rbulk} latency factor to represent any possible latencies. The total time to reprogram the router once is thus represented generally as $w_{prog}B_{rbulk} + l_{rbulk}$.

A more difficult question is how many times the router needs to be reprogrammed. As discussed in Section 5.4, the compiler attempts to keep the most relevant phase information loaded in the router at all times, by only reusing schedule or VFSM-annotation memory belonging to the phase that will be used next as far as possible into the future. However, during the search process that information is not yet available, so it must be approximated.

The current compiler approximates this by assuming that the router memory is very large, and thus the initial load costs are paid only once. While simplifying the cost function, this may result in a slight loss in optimality. A future extension would be to update predicted per-phase times every time a loop is closed, by determining whether or not the phase data that makes up the loop can fit in the router. If a loop of four phases can fit in the router, then the router loading price only needs to be paid once for that loop. If there are five phases, then two phases need to be reloaded each time through the loop; two can be chosen at random and their load time penalized appropriately. This would continue for each enclosing loop, penalizing phases depending on whether they can fit in the router together or not.

An additional complication comes from the fact that some phases reload their data each time through the innermost loop. There are several types of phases of this sort, as discussed previously: run-time schedule-generators, multiple-schedule implementations, and runtime-subphase meta-implementations. In each case, they may potentially download a complete new set of schedule information into the mesh at each inner-loop load.

A COP language feature allows information about application behavior to be presented to the COP compiler. A `(runtime)` value may be annotated with the optional argument '`:block` b', where b is the number of load iterations where the run-time value does *not* change. Thus, $b = 1$ (the default) means the run-time value is assumed to change each time through the loop, while $b = 10$ would suggest that the value only changed infrequently. (This assumes that the *load* function remembers the value it was previously invoked with, and does not reload if it is given the same value again.)

Depending on the type of implementation, a multiplier m_{load} is thus chosen for the number of times the phase must be loaded. For a regular phase with no run-time load requirements, m_{load} is 1 (or, if inner-loop cache occupancy is tracked as discussed above, the predicted value from that analysis is used). If the phase is loaded (potentially) each time through the inner loop, that yields $m_{load} = L_{tot}/b$, the total number of loops enclosing the phase divided by the provided `:block` value.

Some implementations perform minor adjustments each time through the inner loop, though they do not reload the entire schedule. For example, a run-time broadcast (as described under 'Run-Time Specification' in Section 4.1.1) might need to update several VFSM annotations each time through the loop (subject to the `:block` value, b). If a given operator needs to send w_{reprog} words of reprogramming information to the router to reprogram, a cost factor of $(L_{tot}/b)(w_{reprog}B_r)$ is added to reflect the time to perform the reprogram. This assumes that reprogramming is sufficiently low-volume that there is no benefit in performing bulk reprogramming, as may be done for the entire phase information. Finally, the reprogramming times for each operator are summed, taking $w_{reprog} = 0$ for operator implementations that do not require this kind of reprogramming.

The total predicted load time is thus:

$$T_{load} = m_{load}(w_{prog}B_{rbulk} + l_{rbulk}) + \sum_{ops}(L_{tot}/b)(w_{reprog}B_r) \tag{6.3}$$

The total cost for the phase can now be computed by summing T_{io}, T_{sched}, and T_{load} from Equations 6.1, 6.2, and 6.3.

6.2.6 A Cost Metric for Stream Bandwidths

One critical piece of information used to determine timings in Section 6.2.5 is the predicted time per data word, used to derive the stream bandwidth b. For schedule-generators, that time is known exactly. For stream-alias implementations, it must be predicted in some manner.

While the compiler could present the streams to the stream-router module (see Section 6.3) to get exact timings, this is likely to be unacceptably slow for a cost metric function. Even a simpler solution (such as using multicommodity flow and ignoring the need to schedule router timeslots) still takes longer than can be afforded when cost functions are being computed. Instead, an approximation algorithm is used to determine a likely best bandwidth for each stream. (Any online router streams, as discussed in Section 4.2, are included in the set of streams for which bandwidths are derived.)

The algorithm is based on the notion of streams sharing resources. Two basic classes of resources are used:

- **Node** resources are taken to mean the bandwidth in and out of a processor. The maximum bandwidth here is equal to the number of pipelines, unless constrained by available interface bandwidth. Each node resource includes all streams that have a source or destination on that node.

- **Bisection** resources are the x, y, and z dimension bisections of the mesh. The extrema values for the coordinates of the stream sources and destinations are computed, and the extrema's average in each dimension is taken as the bisection. The maximum bandwidth allowed across each bisection is the product of the distance between the extremas in the other two dimensions. Each stream that has the source and a destination on opposite sides of a bisection is added to the resource associated with that bisection.

Ideally, the compiler should simultaneously solve the constraints placed on all the streams by the resource restrictions. However, again, the goal is to get a quick solution suitable for a cost metric. Since the model itself is not exact, the solution does not have to be exact either. Therefore, the compiler uses a quick heap-based priority-queue solution to find a greedy solution to the problem.

After placing the resources on the priority queue, they are removed starting with those resources that are the furthest over their maximum bandwidth. For each resource removed, the traffic count for each stream using that resource is considered, and the available bandwidth prorated across the streams as a relative fraction of their traffic. Then the compiler scans through the streams. It may find that some streams already have the target bandwidth, or less, from other resource limitations. In this case it eliminates those streams, and their share of the total traffic, from consideration, and repeats the calculation.

Once the compiler has the set of streams that need to be decreased, it finds the operator corresponding to that stream, and bumps down the bandwidths on all the streams in that operator appropriately. This is done on the assumption that even if some part of the operator can run faster, the application is likely bound by the slowest stream as the bottleneck, and therefore the compiler should give away its bandwidth to other operators instead. (An exception is made to this rule for the online router streams, since they are used independent of each other.)

As the compiler iterates over the streams whose bandwidth are decreasing, it takes care to update all the resources associated with each stream appropriately, and to move each resource further from the head of the priority queue if necessary. Once the highest-priority resource is found to have a total bandwidth that matches its allowable bandwidth, the algorithm is complete. The 'word time' per operator, B_{io}, is now given by the final bandwidth of each operator's streams.

While the discussion so far has been targeted at the scheduled routing substrate, a largely similar algorithm is used for the dynamic routing substrate. Streams with run-time sources or destinations are not linked to any node constraints, but the range of possible values (as specified by `:values`) is examined to determine if each stream should be linked to any bisection constraints. Almost the same heuristics are used for computing bandwidth at each constraint, but streams from the same operator contribute only the MAX of their bandwidths to the relevant resources, since an operator's streams will not all be used at the same time. Arguably, sequential operators should not sum their stream bandwidths, but since operators run at different times on different nodes, the overlap is such that it's probably fair to include the effects of overlapping sequential operators as contributing to the overall average stream bandwidths. An interesting avenue for future work is trying to predict more accurately the likely stream bandwidths from operators on a dynamic-routing network in the face of contention and sequential or parallel structured communications.

6.3 Stream Routing

For a dynamic routing substrate, having chosen implementations and phase divisions, the compilation is largely done. For a scheduled routing substrate, there remains the final step of generating schedules based on the implementations. Phases containing a single schedule-generator implementation use the implementation's code to generate their schedules; all the other phases must use a stream router. A separate stream router is assumed, such as Tadpole [45] for NuMesh; such a router takes static, fixed streams, and finds a schedule for them over a given set of nodes and a given schedule length.

6.3.1 Basic Stream Router Interface

The stream router is provided with a list of nodes and streams, and instructed to route the streams using the given nodes. The nodes provided correspond to the phase's extent, so that the stream router does not try to misroute any operators outside of the legal bounds for the phase. The streams provided to the stream router correspond to the streams that are active in this phase.

A stream router should accept a set of 'relative priorities' of each stream it is asked to route. This priority would consist, for COP, of the amount of traffic carried on each stream. The relative priorities would then be used whenever two streams conflicted for a given resource; typically this would mean a wire between two nodes, but could be applied for any limited resource. The router would partition that resource between the streams using it in a way that gave each stream relative bandwidth corresponding to its relative priority. Since decreasing a stream's bandwidth to fit into one constrained resource might change the amount of bandwidth it uses for other resources it shares, the stream router should correctly balance the needs of all the streams from a global perspective.

When the stream routing finishes successfully, the returned stream-route information is converted into scheduled routing instructions, creating any necessary router-specific fork, delay, or pipeline-transfer semantics. The final result is information for each node on what VFSMs are required in each pipeline, as well as a complete schedule of how to run those VFSMs. The compiler walks through the returned stream-route information graph for each stream and constructs a tree corresponding to the usage of VFSMs and pipelines for that stream. This tree is attached to the stream structure for later use, such as for run-time updates by data-dependent operator implementations.

6.3.2 Stream Router Cost Function

Normally, the stream router is allowed to take care of working around any idiosyncrasies of the target hardware. However, COP's reprogramming efforts may result in violating hardware restrictions unless care is taken. For example, consider the case that there are hardware restrictions on accessing the processor interface. Since the stream router would not know (for example) that the compiler will be reprogramming a given VFSM to read from the interface, COP must be careful to stay involved with the stream-routing process and prevent the stream router from scheduling VFSMs that would cause conflicts on the access after COP performs its reprogramming.

This is implemented with an additional *cost function* passed to the stream router. This function is invoked for each $\langle \mathit{node,pipeline,timeslot} \rangle$ triple that the stream router is considering using to route a stream. The function can return zero if the stream is good, or a larger number to indicate a higher cost. An 'infinite' value can be returned to indicate an impossible connection.

The cost function is used for hardware-specific constraints (see Section 7.1.4 for NuMesh details) as well as for one general scheduled-routing constraints, that of $\langle \mathit{wire/timeslot} \rangle$ re-use after phase changes. For each possible link of a route that the stream router proposes, the compiler checks the previous (and/or next) phase(s) to determine what streams might cause inter-phase conflicts. It determines whether the operator is terminating, and whether the constraining phase ended with a global barrier (and determines the same for the local phase, if the potential conflict is from a next phase). The compiler then applies the techniques described in Section 5.3 to determine whether there will be a conflict, and if so, it prevents the stream router from using that link at that time for that stream.

The cost function also attempts to ensure that a continuing operator will reuse schedule slots that have already been scheduled by applying a small penalty to each slot it uses other than its previously-scheduled slots. The resulting costs will encourage the router to choose a route for the stream that uses as many of the already-scheduled slots as possible.

Chapter 7

Back-End Implementations

This chapter discusses implementation details of the two existing compiler backends. One implementation supports the NuMesh reprogrammable scheduled router (in fact, a superset of it); the other implementation supports a variety of traditional dynamic routers (deterministic and adaptive). The target router type may be selected by providing '`-m dyn`' or '`-m numesh`' to the COP compiler.

7.1 Scheduled-Routing Backend

The scheduled-routing backend of the compiler is parameterized to compile the input language to a scheduled-routing substrate with an arbitrary, user-specifiable number of pipelines, schedules, VFSMs, and interface addresses. The compiler assumes the NuMesh hardware's constraints (discussed below), and generates NuMesh code as its output format for scheduled routing; however, if the specified architectural parameters exceed the limits of the current NuMesh hardware (and simulator) the compiler will emit a warning message and mark the generated program as non-executable.

7.1.1 The NuMesh Hardware

NuMesh [58, 59] is the canonical example of a reprogrammable scheduled router. This section adds implementation-specific detail to the high-level description presented in Section 1.4.

Pipeline Stages

The high-level discussion implicitly assumed two pipeline stages, with each VFSM reading on one cycle, then writing on the next. In practice, additional cycles are needed to handle reading the schedule and reading the VFSM 'annotations' that determine the source and destination of the data associated with the VFSM. The pipelining is designed such that the longest delay in any stage of the router is the longer of a small RAM read or a inter-node transfer. A schematic of the architecture for one of the routers is included in Figure 7-1; it shows only one of the pipelines, for clarity.

The first stage of the pipeline is the *schedule stage*. In this stage, a small memory that holds the scheduling information is read. Each schedule RAM entry holds a VFSM number

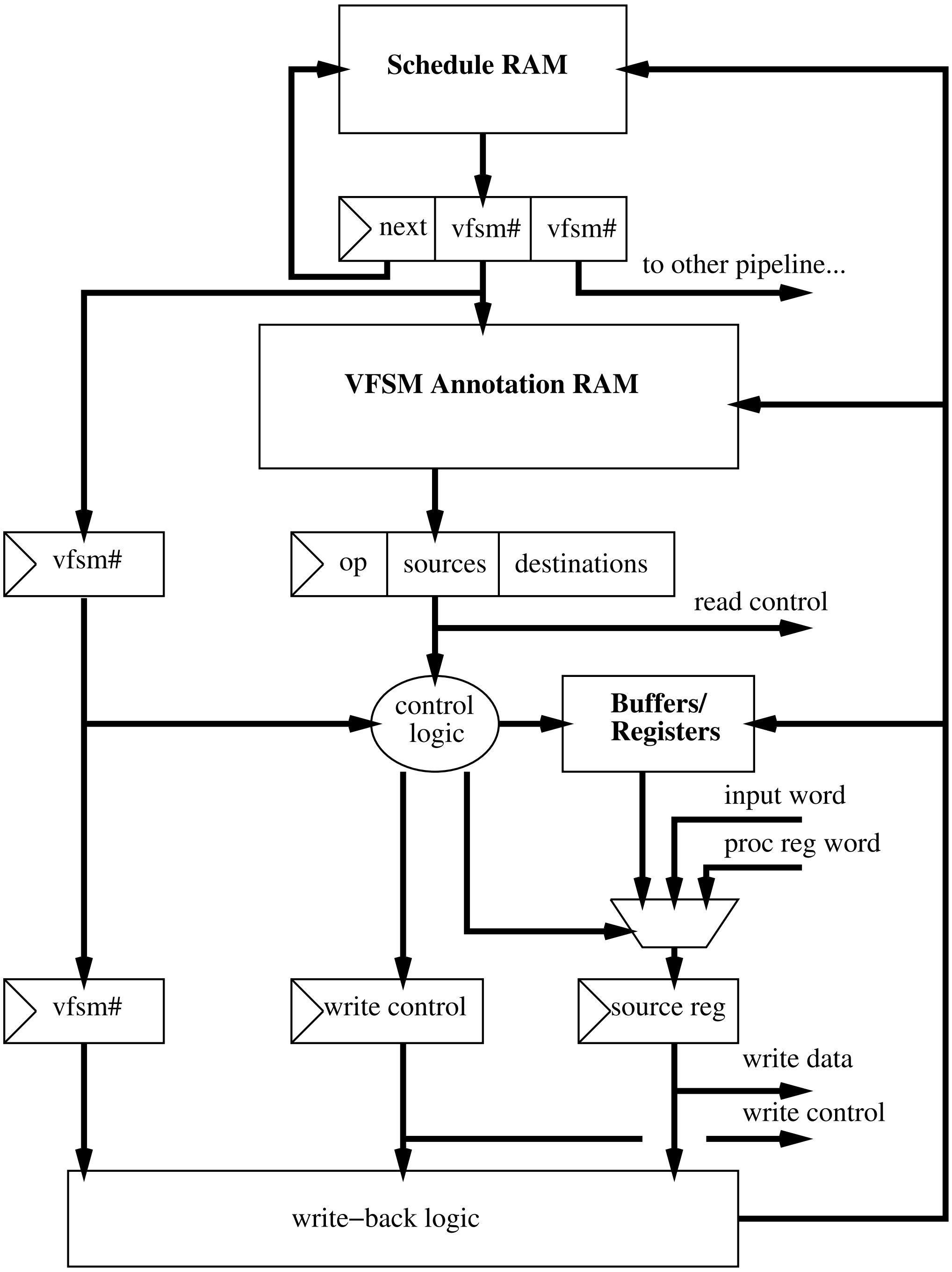


Figure 7-1: NuMesh hardware architecture

for each pipeline, and a 'next schedule pointer' value that tells the router which schedule RAM entry to read on the following cycle. Effectively, the schedule RAM is configured as a linked list. This allows a wide range of possibilities in how to handle allocating space in the schedule RAM for multiple schedules; in the COP compiler, the RAM is simply divided into a set of equal-sized blocks and the link pointers are arranged to iterate sequentially within each block. The schedule RAM holds 128 entries in the current implementation.

The second stage of the pipeline is the *instruction stage*. This stage, and the following ones, are separate for each pipeline in the router. Given the VFSM index passed from the previous stage, the VFSM annotation RAM is read at that index to find the annotations for the VFSM. The VFSM annotation has three fields: one for the type of data move and one each for source and destination. The data-move type used by COP is always a flow-controlled move; other options are a blind move (no flow control), and a 'copy move', discussed in detail in [56]. The possible sources and destinations are given in Table 7.1; some are not used by COP and are not discussed here. The current implementation supports up to 32 concurrent VFSMs in each pipeline.

Value	Source/Destination
0–5	Neighbor nodes $(-x, +x, -y, +y, -z, +z)$
6	MEMADR destination; JTAG TDO bit source (see [56])
7	special buffer handling (see [56])
8	NOP (invalid source; unsuccessful destination)
13	NULL (always-valid source; bit-bucket destination)
16–31	processor interface registers
32–63	VFSM buffer registers

Table 7.1: NuMesh VFSM sources and destinations

The third, or *read* stage, decodes the source and fetches the appropriate input word. Finally, the fourth, or *write* stage, decodes the destination and attempts to write the data word appropriately. The data path for the architecture is shown in Figure 7-2.

Processor Interface

Interfacing to the processor is managed by an interface register set, which supports flow control (if desired) in both directions. Any pipeline can read or write to the interface registers on each cycle, and the processor can do likewise. Access from the router side is single cycle; the current implementation for the processor uses the SPARC Local Graphics Bus, which is capable of single-cycle writes (after a single cycle of startup), and two-cycle reads. The interface has 16 registers.

The important restriction on the current implementation's processor interface is that each pipeline can access the interface at most once on each cycle. Therefore, a VFSM that reads from the interface cannot be scheduled after a VFSM that writes to the interface: the write in the fourth stage and the read in the third stage would, illegally, occur at the same time.

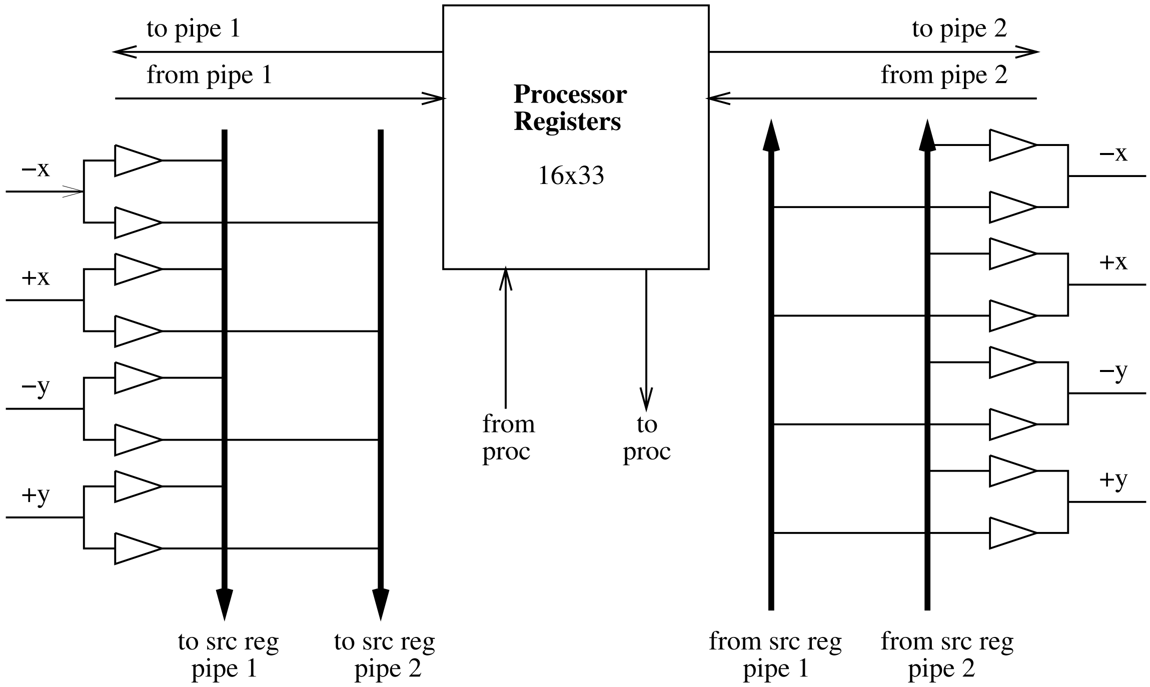


Figure 7-2: NuMesh datapath

VFSM Scheduling

A limitation of the NuMesh hardware is that you can not schedule a VFSM back-to-back, achieving 100% bandwidth. This is due to a fundamental information-flow problem: at 100% bandwidth, a chain of VFSMs would need to be able to propagate an invalid accept signal all the way to the source of the stream in a single cycle, or else nodes would end up unable to buffer data correctly. This problem could be brute-forced in hardware, by increasing the buffering (at noticeable cost in speed and area). Instead, two interleaved streams are used to achieve 100% bandwidth.

Reprogramming

The NuMesh is capable of being reprogrammed by specifying a special destination address, MEMADR, which allows updating most of the memory in the router. Specifically, a MEMADR write can update the schedule memory or the value of the schedule pointer (when issued from pipeline zero), the VFSM annotation memory for the pipeline the write is performed in, and a variety of other system services. In COP, one interface register per pipeline is dedicated to a VFSM that reads from that register and writes anything it finds to the MEMADR destination.

The reprogramming VFSM is scheduled as frequently as possible given constraints of other data-transfer VFSMs. For a given phase, the number of times it can be scheduled corresponds to the ‘reprogramming bandwidth’ B_r defined in Section 6.2.5. Since this value is only known after stream routing has been performed, and can vary from as low as 1 to once every few cycles (commonly) to as high as only once per schedule iteration, it is approximated in the NuMesh backend as a small constant.

Data-Transfer Restrictions

Some of the data transfer techniques used by NuMesh are slightly idiosyncratic.

Flow-control Wire Aliasing. To save on pins and wires, the NuMesh uses the same wire for the *valid* and *accept* flow-control signals. In normal use, this is a sensible overloading; there is always a read and a write at opposite ends of the wire, since the VFSMs are globally scheduled. However, during a phase change, it is important to account for this: if two neighbor nodes both schedule a read from each other at the same time, they may both assert their *accept* signal, then both think they see a *valid* signal from the neighbor, and both will consider the undriven data lines to hold a valid data word.

Scheduling Delays. When delaying a stream on a given node, two VFSMs are scheduled. The earlier VFSM can write its data, flow-controlled, to its own buffer register; the later VFSM can then read from the first VFSM's buffer register directly. When the second VFSM has a successful read, it clears the first VFSM's buffer, making it ready to accept more data. However, if multiple VFSMs need to collect data into a single outgoing stream, they must instead all write their data to the buffer of the outgoing stream's VFSM. Writing to the other VFSM is almost the same as reading from the other VFSM, but due to hardware constraints a VFSM can't write to the buffer of the VFSM scheduled immediately after it.

Data Forking. This is handled by scheduling two VFSMs back-to-back. The first VFSM writes the first destination (which must be a neighboring node) in the usual manner. The second VFSM then specifies its source not as the real source of the data, but as the first VFSM's destination. This special case is recognized in the router, and the success of the first VFSM's transfer becomes conditional on the ability of both the 'real' destination and the forking VFSM to accept the data. Forks of more than two destinations are handled by scheduling multiple sequential VFSMs, with data passing down pairwise between VFSMs.

NuMesh Topology

The NuMesh architecture is suitable for a variety of mesh topologies. The instruction-set architecture allows for up to six neighbors, thus allowing a three-dimensional Cartesian mesh arrangement. The current implementation of the chip has a four-neighbor pinout, allowing either a two-dimensional Cartesian mesh or a three-dimensional diamond lattice (as mentioned in Section 1.2.6). The COP compiler targets the 2D and 3D Cartesian meshes rather than the diamond lattice.

7.1.2 NuMesh Phase-Changing Restrictions

There are several implementation issues to be aware of when handling phase changes for NuMesh.

One initial observation worth mentioning is *how* the router can be instructed to change phases. The current NuMesh architecture supports direct writes to the schedule pointer, via the MEMADR destination. This technique allows for a phase change to occur at only one point in the schedule, thus adding noticeable latency to the phase change operation on average. A simple extension to the router takes advantage of the fact that COP allocates schedules consecutively in RAM by allowing a delta to be added to the schedule pointer instead (or possibly the previous cycle's schedule pointer, to avoid adding latency to the critical path). This extension means that phase changes can be issued at any point in the schedule, and allows MEMADR-updating VFSMs to be scheduled anywhere without worrying about inter-node synchronization issues.

When changing phases, care must be taken after the phase-change command has been issued. If a naïve return to processor code is performed immediately, a write could be issued to an interface register and then read by one of the previous phase's VFSMs still in the pipeline, thus losing data. This is particularly true since a write to the router's update interface register may not be read immediately, increasing the window of vulnerability. It is thus necessary to either arrange such that no interface registers are shared between phases for writing; or, since interface registers are a fairly scarce resource, the phase-changing code can simply wait for the router to consume the phase-change command, stall a few more cycles to drain the router's pipeline, and return to the main application code. (This stall is required in any case for global barrier phase changes.)

There is also a more subtle interaction between the interface registers and the phase change process. If a VFSM is writing the processor interface at time t, just as the phase change completes, a VFSM from the next phase may end up reading the processor interface at time $t + 1$. This creates a conflict in the processor registers. However, as long as one access is required always to succeed, or both accesses always to fail, this transient condition will be resolved in the following pass through the schedule.

7.1.3 The NuMesh Stream Router

The stream router for the NuMesh backend is Tadpole [45]. Tadpole handles single-source, multiple-destination streams, and allows a fixed bandwidth to be requested for each stream. It is linked into the COP compiler as a C library.

Since Tadpole does not currently support optimizing via stream rerouting, the streams are provided in order of reverse difficulty, so that Tadpole will have the best chance of routing them optimally. 'Total schedule entries' is used as the metric of difficulty; a stream with high bandwidth, significant distance between source and destination, or multiple destinations requires more schedule entries, and is thus listed early in the list of streams to route.

Ideally, Tadpole should support 'relative priorities' for streams. However, the current implementation requires that fixed bandwidth values be provided for each stream. Accordingly, the simple cost-metric numbers derived in Section 6.2.6 are used to set bandwidths. Because these numbers may be too optimistic given the constraints of timeslot allocation, and Tadpole regards the provided bandwidths as absolute, Tadpole may fail to allocate at a given bandwidth. When that happens, a binary search is performed, multiplying the streams' values by some scale factor < 1, until the highest routable scale factor is found.

7.1.4 Cost Function Constraints for NuMesh

Processor-Interface Access

A significant constraint in determining feasibility for routes is the fact that the processor interface can only handle a single access per pipeline at a time. Thus, a VFSM that reads the interface (which happens in pipeline stage 3) can't be scheduled after a VFSM that writes the interface (which happens in pipeline stage 4). Normally the router avoids this situation, but reprogramming requires extra care.

The `C_ALLSRC` stream flag (attached to a stream by the relevant operator implementation) means that *any* VFSM, not just the first VFSM of the stream, could potentially read the processor interface. For example, one of the runtime-source broadcast implementations changes a VFSM to read from the interface register. Without a COP-specific cost function to forbid this, the router might schedule a stream destination immediately before this VFSM, thinking it was safe to use the processor-interface registers at that point. A similar flag, `C_MESHSRC`, can be specified to say that the source of the stream will *never* read the interface register. Accordingly, the cost function checks these flags and ensures that the stream is legal to be routed.

MEMADR Access

Another significant constraint on the stream router's choice of schedule slots and pipelines is COP's requirement to be able to reprogram the router. Thus, for example, suppose the stream router tries to route streams from an operator that is using a reprogramming implementation into a particular pipeline. The compiler must ensure that the operator does not use the last schedule slots in the pipeline, since, if it does, there will be no room to add a reprogramming slot to the schedule. Similarly, it is necessary to make sure that there is a suitable pipeline available to issue phase-change commands to the mesh.

Determining whether a given pipeline can schedule a MEMADR update VFSM requires walking the schedule for that pipeline and examining the streams that are currently routed there. To allow a MEMADR VFSM to be scheduled, there must be an empty slot that is not preceded by a VFSM that writes to the interface registers, since an interface-register read can not follow an interface-register write. As discussed above, the stream router must be alert to the `C_MESHDEST` and `C_ALLDEST` flags when determining whether the VFSM scheduled before a vacant slot may cause trouble for a potential MEMADR VFSM there.

The first thing checked is that the given target pipeline is capable of supporting the extra stream. If the stream is part of an operator with a reprogramming implementation, the stream may be marked for reprogramming; other already-routed streams in the pipeline may also be marked for reprogramming. Either way, the pipeline must have room for a MEMADR update VFSM to be scheduled, or else it will not be possible to actually perform the reprogramming at run time.

If the pipeline has room to schedule the reprogramming VFSM, no action is required; by contrast, if there is no room but some operator in the pipe requires reprogramming, the cost function must refuse the stream router to use that schedule slot. If the pipeline has no slots for the reprogramming VFSM, but nothing in the pipe requires reprogramming, there is still another constraint: the end-of-phase reprogramming step that is required for every phase. In the current compiler implementation, pipeline zero is always used for the phase-transfer reprogram. However, since pipelines can be relocated (as discussed in Section 5.4.2), as long as some relocatable pipeline has a MEMADR slot in it the additional schedule slot usage is allowable. (Recall that pipelines may become non-relocatable when a continuing operator is scheduled in them.)

As is true for barriers (Section 5.2), any phase that has no successors is privileged in that it does not need a reprogramming VFSM to be scheduled in, and accordingly may carry slightly more bandwidth. Notice that this is computed on a node-by-node basis, unlike the barriers, since there are no global issues involved in determining whether or not to schedule a MEMADR update VFSM.

Co-Scheduling Streams

The final piece of functionality provided by the cost function allows implementations to request that certain streams be scheduled into the same pipeline when they appear on a node together.

Thus, for example, in the `collect` implementation discussed above, data at the NuMesh level must be transferred from one stream's destination to another's source. NuMesh only provides for this operation within a pipeline; accordingly, each child's destination pipeline must be the same as the parent's source pipeline. This does not require that the same pipeline be used on all nodes, simply that each node choose a single, consistent pipeline to use.

In this case another NuMesh constraint must be watched for: a stream may not write into the buffer of a stream that is scheduled immediately after it. The cost function is used (for suitably-flagged streams) to ensure that a new destination is never scheduled one slot before an existing source, nor is a source ever scheduled one slot before any existing destination.

A similar use for this functionality is when one stream needs to reprogram another. For example, the in-router barrier function allows a child to reprogram the destination of the next child (or the source of the parent, if it is the last child). Again, the NuMesh only allows VFSM-reprogramming to be performed from within the same pipeline, so the same restrictions must be applied to the stream router's path choices here as well. In this case, however, the proximity cost mentioned in the previous paragraph is irrelevant.

7.1.5 NuMesh Caching Restrictions

Each schedule slot in NuMesh is taken to be a block of s consecutive addresses (where s is the schedule length). While NuMesh has the ability to arbitrarily interleave schedules' data in the memory, with fixed-size schedules this is unnecessary, and in fact would make it impossible to use the add-to-schedule-pointer feature that is used to transfer from phase to phase.

While (as mentioned in Section 5.4.1) no attempt is made to do on-the-fly aliasing of VFSMs, much of the benefit of VFSM aliasing is gained by making VFSMs zero and one in each pipeline correspond to a NOP (annotated as NOP-to-NOP) and a reprogram (annotated as interface-to-MEMADR) respectively, and aliasing those together in all phases. An empty schedule slot in any phase is normally represented by a reference to the NOP VFSM.

After the pipeline and VFSM cache mapping is computed, the schedule is examined and as many of the scheduled NOPs as possible are converted to reprogram references. All NOPs are converted except those following VFSMs that write to the interface registers, since the pipeline delay means that the reprogramming interface-read would occur at the same time as the previous interface-write. The more reprogramming VFSMs that can be scheduled into a phase, the lower the latency will be for phase changes out of that phase, or any necessary runtime reprogramming.

The hardware requirement that pipeline zero's reprogramming VFSM be used to update the schedule pointer can constrain the set of possible pipeline permutations are considered while computing the VFSM cache mapping. Specifically, no permutation can be used which maps pipeline i to pipeline 0 if the pipeline has no reprogramming VFSM scheduled.

At boot time, a simple *loader phase* is loaded in the router, identical on all the nodes in the mesh. This phase uses the first schedule slot for a schedule in which every cycle runs the same pair of VFSMs (the reprogramming VFSM in each pipeline), used to load both pipelines depending on which interface register is written to. Currently, this schedule is locked down and not eligible for eviction. It consumes one slot in the schedule space, and effectively consumes no additional VFSM resources. A future version of the compiler could consider evicting and later reloading the loader phase, but for now it is left hardwired, and used to do high-speed transfers to the router as necessary.

7.2 Dynamic-Routing Backend

COP contains more compile-time information on communications than is required for a standard dynamic routing substrate. However, configuring COP to generate code for this target allows a fair comparison of scheduled router performance against dynamic router performance.

The implementation of a given operator can be quite straightforward for a dynamic-routed substrate. For example, a `stream` operator is supported equally easily for runtime source and destination as it is for compile-time arguments. In either case, the implementation generates the message by emitting an appropriate header word, then issuing all the words of the message, marking the last word with an end-of-message bit. Reads are handled by reading the message header word (if provided by the routing substrate), then reading all the words of the message.

Under simulation, the resulting compiled application can be run in several environments. The simulator handles standard Cartesian and diamond-lattice dynamic routing methods, as well as a precomputed shortest-path technique that can be used for arbitrary networks. Routing protocols currently supported include standard E-cube (and variants for diamond-lattice and arbitrary networks) and the dynamic adaptive routing protocol discussed in [20].

7.2.1 Disambiguating Messages

The implementation of COP for the chosen dynamic router model allows different messages to be delivered to different interface addresses. This technique provides a more level playing ground for comparing dynamic routing to scheduled routing.

The scheduled routing model assumes that all messages arrive with a *tag*: specifically, they are routed to a given interface address. This allows the application to read from one interface address at a time, with a guarantee that it will get only the data that it is currently interested in at that address. Other data may be pending arrival, but will be blocking on a different interface address. Other messages are left buffered in the network; in a scheduled network this will not effect other messages, it will just block the writer, if the message in question is long enough. Blocking the writer is taken to be an appropriate response if the writer is so far ahead of the reader that the writer is generating messages for which the reader is not ready.

By contrast, the traditional dynamic-routing solution is for the router to provide untagged messages to the processor for further handling. The processor is then responsible for examining the message and determining what to do with it: making it available directly to a user program polling for that message or allocating memory to queue it, for example. The user program may in turn have to examine messages to determine the type of message, then buffer and/or skip over messages the application is not yet ready for. This buffering can both block forward progress on the reader node and require additional memory traffic for writing to the buffer and later reading back from the buffer.

To avoid this problem, online messages are tagged in the same way that is done for offline-schedule messages. Specifically, a processor interface address is encoded in the header of the message. Messages will then block on a specific address until they are read by the processor. This trades off low-latency access to the next message desired by the application against the possibility of increased contention in the network from messages that are blocking on interface addresses that have not yet been read. However, since most of the applications considered in this thesis have a large percentage of collective operations, for which the message would be consumed upon arrival by the waiting application, this was deemed an appropriate tradeoff. Additionally, using tagging on messages simplifies the application code. Queuing in the router must be allowed for each interface address, to avoid blocking in the case that a message arrives early for a later operator.

7.2.2 Dependency Analysis for Interface Address Allocation

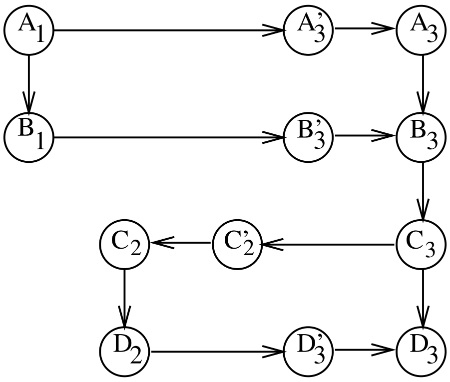
To determine the tag used on each message, a distinct tag would ideally be used for each type of message (*i.e.*, each stream starting or ending on the local node). However, to stay within the limited number of interface addresses available, dependency analysis is performed to determine when interface addresses can be reused at each node. Address zero is hardwired for outbound traffic, so addresses one and higher are available as destinations for inbound traffic.

The key question for reusing an address for a new operator (say operator C) is to examine other operators (without loss of generality, A and B) assigned to that address, and determine whether A and B can be guaranteed to have consumed all their expected messages before the arrival of any messages for C. This can be done using a dependency analysis technique similar to that presented in Section 5.3.4. This technique can also be used for online routing under scheduled routing, for similar reasons, as mentioned in Section 4.2.1.

When using dependency analysis on dynamic streams, it is important to ignore streams with a runtime source or destination, since such streams do not allow for predictable dependency analysis; a given operator can not rely on the presence of a communications arc in that case. A possible future extension of this algorithm would be to do dependency checking by checking all possible runtime arcs; if the dependency is present for *every* possible set of runtime values, it can be treated as a valid dependency.

Having computed the dependency graph for a program (a sample is given in Figure 7-3), the closure of the graph is then computed, for each arrival vertex N'_n on node n finding all the vertices that can be reached from N'_n. During the graph search any edges that lead back to a node associated with operator N are not followed, thus avoiding carrying dependencies more than once around any loops. (If any such connections are found, N_n is marked as being in a loop.) A list is made of all the arrival vertices V'_n on node n whose I/O vertex V_n could be reached by N'_n, but which could not themselves be reached. This list thus contains operators that may be be receiving data at the same time as N on node n.

To determine interface address usage, a graph holding all the arrival vertices is created. For




```
(sequential
  (stream 'A 1 3)
  (stream 'B 1 3)
  (stream 'C 3 2)
  (stream 'D 2 3))
```

Figure 7-3: Dependency graph for simple COP fragment

each arrival vertex N'_n, an undirected edge is added to all the arrival vertices on the list that was derived in the previous step. The corresponding graph from the example in Figure 7-3 above is in Figure 7-4. The final step is an N-coloring of the graph to assign interface addresses to arrival nodes.[1] For this example, an appropriate address allocation on node 3 might then be $A = 1, B = 2, D = 1$.

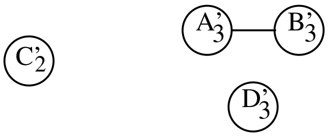


Figure 7-4: Address allocation graph for simple COP fragment

An additional dependency constraint can be added in some cases (although this is not implemented in the current compiler). When two streams have the same source and destination, data is guaranteed to arrive sequentially first from one, and then from the other. An appropriate edge can therefore be added to the dependency graph in this case. However, this only holds when the routing is deterministic; if adaptive routing is used, the ordering of messages in transit between the two nodes can't be guaranteed. Figure 7-5 shows the extra edge, as well as the structure of the graph if the top-level `sequential` construct were replaced with

[1]A simple greedy algorithm is used in the current implementation of the compiler to compute the coloring problem.

a `loop`. When computing the address allocation graph for this dependency graph, using the deterministic-routing assumption, the edge between D'_3 and A'_3 suggests an address allocation of $A = 1, B = 1, D = 2$ on node 3.

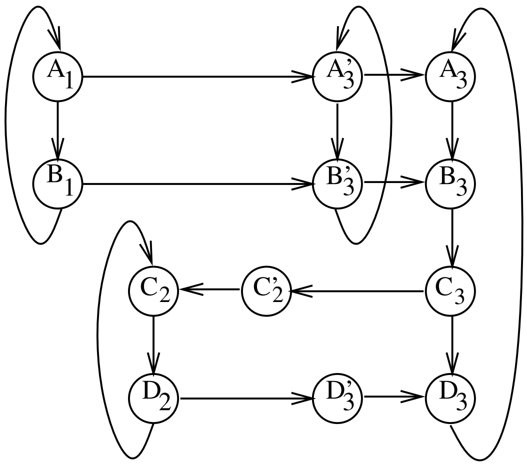


Figure 7-5: Dependency graph for simple COP loop with deterministic routing

The compiler does not currently try to allocate multiple interface addresses for writer nodes. Doing so could potentially ensure that remote-read-delays on one long outbound message would not prevent messages behind it from getting through to their destination. However, if the message would be blocking because the data ahead of it has not been read yet, the odds are that the previous write would have blocked anyway in this situation. Accordingly, the interface addresses are currently used only for differentiating operators at read time. Investigating the trade-offs in allowing multiple write addresses might be interesting future work.

7.2.3 Supporting Virtual Interface Addresses

If there are a large number of operators running in parallel, the dependency analysis may show that more interface addresses are needed than are, in fact, available. In this case, the highest-bandwidth operators can be allocated to the physical interface addresses (in the current implementation this would allow up to fourteen distinct read addresses). The remaining operators can all be allocated to a multiplex interface address.[2]

Operators that use the multiplex address treat it somewhat differently from the regular interface addresses. The writer adds an extra header word which holds the virtual interface address. On the reader side, queues of delivered messages are maintained (similar to the techniques used for online routing on a scheduled router; see Section 4.2). When a multiplexed operator's read function is called, it checks the queue corresponding to its virtual interface address.

[2]The material in this section is currently unimplemented, as sixteen addresses have been enough for the per-phase demands in applications considered to date.

If a message is present in the queue, it returns that message directly. Otherwise, it reads the multiplex address until the message it is interested in arrives. While waiting, if other messages arrive, it reads them and queues them to the appropriate queue based on their virtual interface address. (Messages will not be interleaved, since the dynamic router will lock the interface address message-by-message, just as is done for writing to neighboring nodes.)

Some dynamic-router interfaces are incapable of supporting multiple delivery addresses. The notion of a ‘multiplex address’ allows such hardware to be supported from COP. For example, an interface consisting of paired FIFOs for reading and writing the network could be handled by providing the ‘-P 2’ argument to COP so that it codes for a single output and a single input address, multiplexing all inputs through the virtual interface address queues.

Chapter 8

Results

This chapter discusses some of the results obtained from simulating the output of the COP compiler. An overview of the experimental methodology is presented, then a series of experimental results generated by the system, giving results and analysis. The experiments include a variety of different communication patterns in a range of applications and network sizes.

8.1 Experimental Methodology

8.1.1 Compilation

The experiments discussed in this chapter are the results of simulating the output of the COP compiler. Application code was written in C, with communications specified in COP.[1] All application and COP code was hand-generated; the results are equally applicable to a system with a high-level language compiler emitting C/COP, however, since the C/COP environment is an adequately high-level one, and reasonable to program in. The COP code is compiled into C, then the C application code (which `#includes` the COP output) is compiled. The scheduled-routing and dynamic-routing platforms are tested using the same C and COP code.

8.1.2 Simulation

The compiled output from the C/COP environment was run on the the NuMesh simulator, `nsim` [50]. The simulator is event-driven, with a modular mechanism for attaching routing elements, CPU elements, and interface elements between the two. As well as being extensible with modules written with C linkage conventions, it is directly user-programmable in Tcl, and includes an X interface for displaying runtime status and viewing the mesh being simulated. The simulator gives cycle-by-cycle results for data motion in the router and interface modules.

The simulation of the processing time is crude. The simulator uses a simple threads-based model for running compiled code on multiple virtual nodes; control is passed back to the simulator on I/O events and when delays are explicitly requested. Delays have been inserted

[1] COP code is embedded in the C code by making the first line of the file '`;/*`', followed immediately by the COP code.

into the experimental applications to approximate processing time, but a more sophisticated processor-simulator module, or well-instrumented hardware, would be an appropriate step for further quantitative analysis.

Unless otherwise mentioned, the processor interface is assumed to be capable of doing flow-controlled reads and writes at one per cycle. If a less aggressive interface were assumed, it would tend to benefit scheduled routing more than dynamic routing, since the interleaving of streams in the scheduled router would then be largely hidden from the analysis.

8.1.3 Data Collection

COP output includes a hook for writing timing data to a file if requested by an appropriate environment variable. The file contains records that provide information on how time is spent during execution on each processor. A post-processing tool then sweeps the file and produces a per-processor and total breakdown of cycle usage during the application's execution. The data includes time spent in each of the categories in Table 8.1; the categories are discussed in more detail here.

Type	Description
cpu	Computation
write	Writing to the interface
read	Reading from the interface
psync	Phase synchronization
router	Loading complete schedules
reprog	Reprogramming the router

Table 8.1: Instrumentation categories for elapsed time

cpu. This category takes into account time spent processing. Given the simple processing model in the simulator, this is only an approximation, but it should be roughly accurate. It will be the same in both router-hardware models, except when online-routing on NuMesh causes processing cycles to be used by the online routing interrupt handler.

write, read. These categories give time spent waiting on an interface register to become ready to handle a read or write. The first write typically takes only a cycle, since the interface register is empty. Subsequent writes may take longer, if the interface register has not been cleared by the router. Reads give the time spent blocking waiting for data to become available from the router.

The three categories so far cover all the timing for the dynamic router; the remaining three categories are scheduled-routing specific.

psync. This reflects generic phase-synchronization time. For all phase changes, this includes time spent providing the synchronization word, as well as time spent blocking to allow the change to reach the read stage of the pipeline. For phases with other phase-synchronization delay needs (as discussed in Sections 5.2 and 5.3), that time is included here.

router. This is time spent doing basic router loading, as is done on each cache 'miss' (as discussed in Section 5.4). This does not include any later run-time tweaking, but just the baseline time to load a complete schedule with appropriate VFSM annotations.

reprog. This last category includes any time taken performing router reprogramming for run-time implementations, such as time taken by a node to switch itself from a broadcaster to a non-broadcaster.

While some figures present raw cycle counts, others present results in units of wall-clock time, attempting to factor in the difference in cycle speeds between two equivalent implementations. As discussed in Section 1.2.1, and in more detail in [56], a 1:2 cycle speed difference is used between scheduled and deterministic dynamic routers. No additional speed penalty is accrued to adaptive routers, though [56] suggests that the speed penalty would be greater than 1:2 for such routers. While this ratio is admittedly purely theoretical, it seems likely to reflect the approximate difference in possible speeds likely with the two architectures. In fact, scheduled routing can easily be extended to reduce the inter-node latency with pipelining, thus causing a potentially even more dramatic difference in achievable cycle speed. For the sake of fixed units, this chapter assumes a dynamic router at 1 GHz (as is currently the case, for example, for SCI [26]), and a scheduled router at 2 GHz. Where CPU time is factored in against communication time, a processor capable of roughly 1 Gop/sec is assumed, such as a 2-way 500MHz or 3-way 333MHz processor.

8.2 Communication Kernels

The first set of experiments is designed to assess several of the assumptions made in the Introduction. There, it was claimed that not just lower communication time, but lower cycle counts as well, could be achieved. In this section, those claims are quantified.

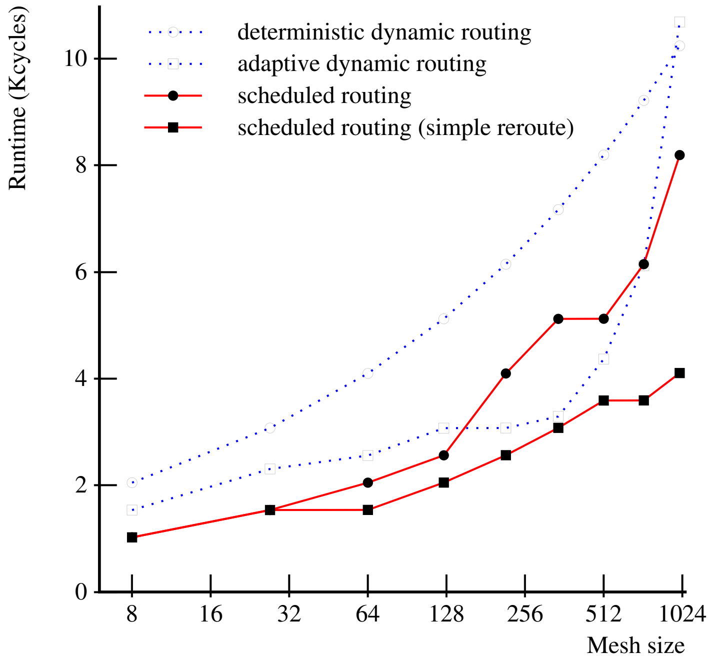
For this section, only worst-case read/write time is presented for both target routers; that is, a single number is presented which corresponds to the elapsed time from initiating the operation on all nodes, until the time the final node completes the operation. The initial load time for the scheduled-routing substrate is omitted so that the I/O time can be seen in isolation; the other components of scheduled routing are returned to in later sections.

8.2.1 Hotspot Avoidance: Results

In Section 1.2.2, it was claimed that scheduled routing can make better use of a network's bandwidth than dynamic algorithms could. In this section, that claim is examined, using two common communications patterns (3D transpose and bitreverse) as model communication kernels.

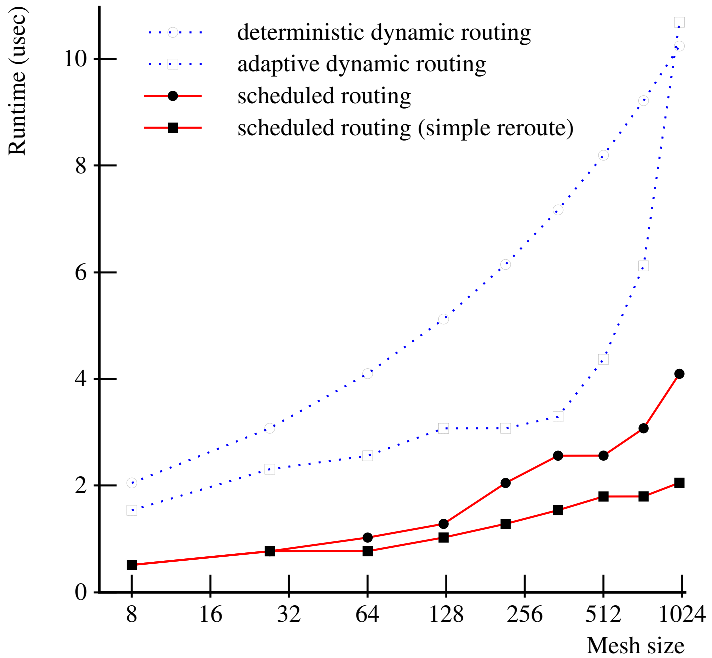
Transpose

This experiment compares transpose performance rotating a 3-dimensional array from (x, y, z) to (y, z, x). The scheduled-routing backend is contrasted to two dynamic routing algorithms—both the traditional oblivious algorithm, as well as a somewhat more sophisticated adaptive algorithm (Dally's adaptive dimension-reversal router) [21]. A small benchmark case, with 2KB of data on each node, is examined. The dynamic router uses 8 words (32 bytes) of buffering on each link, with five virtual channels on each link for the adaptive algorithm. Results are presented in Figure 8-1, with a comparative figure showing wall-clock times in Figure 8-2.



`(transpose 't (make-1d 'x) (make-1d 'z))`

Figure 8-1: Comparative performance for transpose (cycles)



`(transpose 't (make-1d 'x) (make-1d 'z))`

Figure 8-2: Comparative performance for transpose (wall-clock time)

The compiler's current stream-router technology, Tadpole, uses a somewhat primitive route-selection algorithm; each stream is scheduled once and for all as it is encountered, with no rerouting performed. In particular, the jump in execution time around the 200-node-mesh mark is due to Tadpole's difficulty in choosing the best routes. Accordingly, for these results a 'simple reroute' line is included which gives results for a simple rerouting module; however, this router does not allocate timeslots, just bandwidth. This line serves as an indication of potential performance on these datasets when Tadpole is improved or replaced with a more sophisticated stream router.

The oblivious online algorithm has extremely low throughput on these applications, since the paths chosen by the algorithm create bottlenecks on certain links in the mesh. The adaptive algorithm performs better, but still not as well as scheduled routing. For moderate-sized meshes, the performance of the adaptive router can be improved somewhat by using many small adaptively-routed messages instead of sending each chunk of data as a single message. However, this technique requires that the receiving end be prepared to handle out-of-order packets, since under high-load conditions packets may arrive out of order. This does not happen with the oblivious dynamic or the scheduled models, since they only use a single path to transmit data. Using many small packets also can cause the adaptive router to generate excessive misroutes in large meshes, eventually falling back to the oblivious channels, and thus resulting in even longer times to completion than the simple oblivious router.

Bitreverse

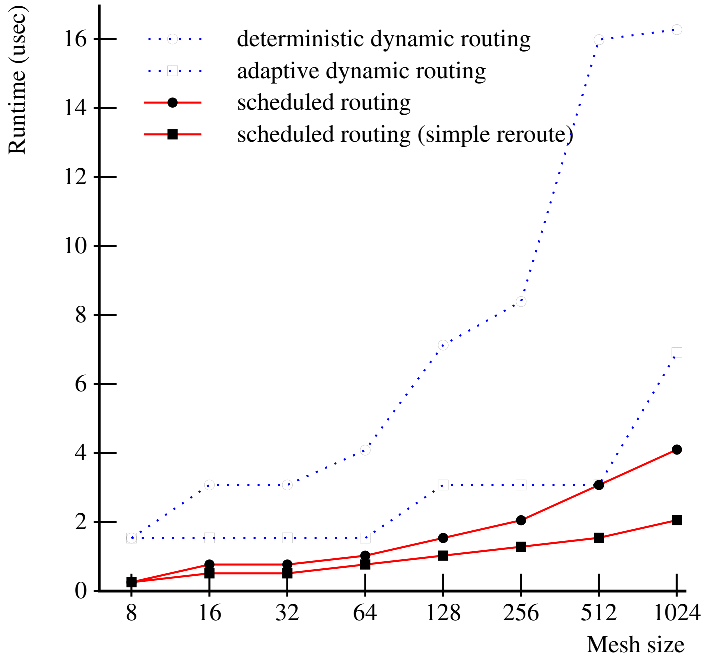
The second experiment in this section is bitreverse, a kernel found (for example) in FFT. The data from node i, with binary address $i_0 i_1 \dots i_k$, is sent to address $i_k i_{k-1} \dots i_0$. The experimental framework is the same as for transpose; wall-clock time results are shown in Figure 8-3. (The equivalent figure for raw clock cycles is shown in Figure 1-3, p. 16.)

As can be seen in the figure, the various routing algorithms result in similar timings as for the transpose case; that is, scheduled routing dramatically outperforms deterministic dynamic routing. The adaptive dynamic router keeps up more closely with the scheduled router here than for transpose, but still has worse performance.

8.2.2 In-Router Algorithms: Results

In Section 1.2.3, it was asserted that scheduled routing allows for clever algorithms to improve the overall cycle times of certain algorithms. In this section, the parallel prefix kernel is examined to support this claim. This kernel is both a powerful primitive as well as a good example of in-router algorithms.

Parallel prefix leverages the programmability of the communications FSM to improve the end-to-end latency of a parallel prefix. To perform parallel prefix, a breadth-first spanning tree is generated over the mesh; each physical node in the mesh is associated with one internal node as well as one leftmost leaf node (considered to hold the initial x_i value) attached to the internal node. The leaf values are propagated upwards, with the $\otimes$ operation applied to them at each step. Partial values are passed back downwards to the leaf nodes as the algorithm runs, with values passed in from above broadcast to all the children as well.



(bitreverse 'b)

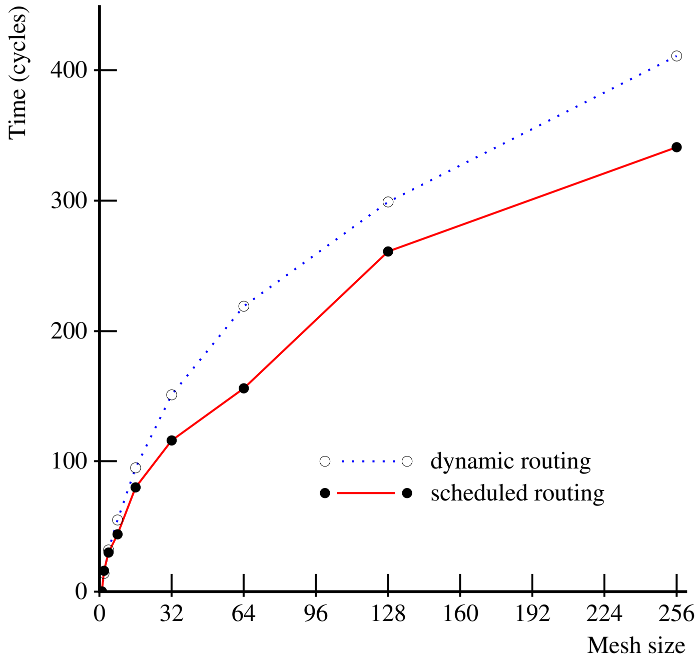
Figure 8-3: Comparative performance for bitreverse (wall-clock time)

Using standard online routing, during the downward phase of data motion, data is sent from a parent node in the tree to its children one by one; those children read the data, then forward it to their own children one at a time. With scheduled routing, the router can accept a single stream of data from the processor, which it duplicates internally to generate the multiple streams of data to each of the children. This removes the potentially time-consuming overhead of passing data through the processor/router interface multiple times, thus reducing the total latency to the processor itself.

Figure 8-4 gives the results, in cycle times, for a 32-bit integer arithmetic prefix. Recall that wall-clock times will show a much greater differential. Cycles are not accrued to the $\otimes$ operator, so as to make the I/O differences clearer; modeling the delay would simply add a constant amount to both hardware types for a given mesh size.

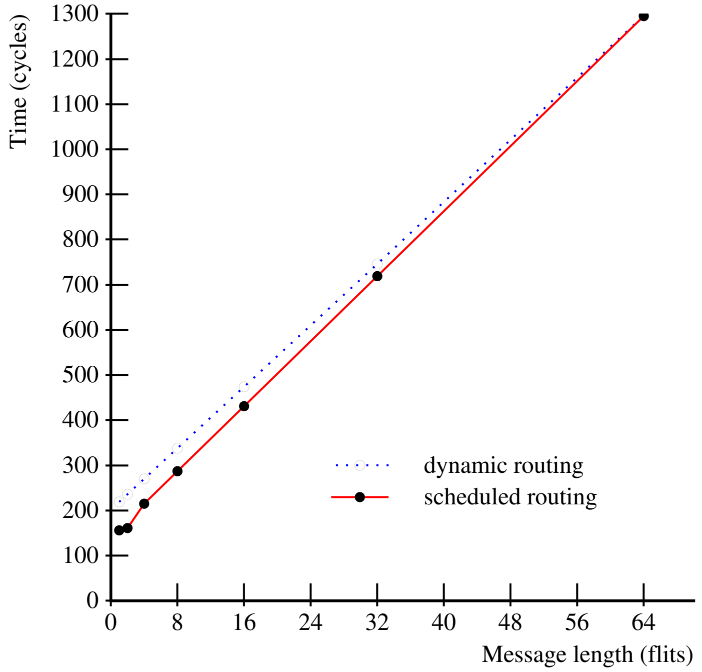
As can be seen in the figure, the scheduled router requires fewer cycles to perform the operation; for a 256-node mesh, the dynamic router is already 20% slower and falling further behind.

Figure 8-5 presents results for a parallel prefix on an 8×8 mesh as the message length is increased from 1 word up to 64 words. Unsurprisingly, as the prefix argument size is increased, the cycle time difference decreases, approaching the bandwidth limits of the processor interface. Overall, however, these results demonstrate that there are improvements to be had from moving communications algorithms partly into the router.



(prefix 'p 'myfunc :m 1)

Figure 8-4: Prefix times (one-word prefix, cycles)



(prefix 'p 'myfunc :m 64)

Figure 8-5: Prefix times (mesh size 64, cycles)

8.3 Data-Dependent Communications Techniques

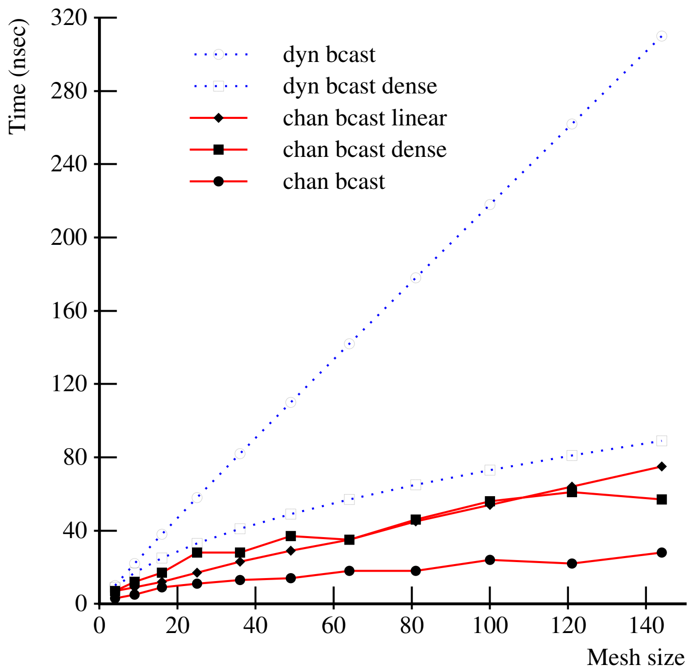
The next round of experiments examines more closely the claim that scheduled routing is capable of good performance even with data-dependent communications. To demonstrate this, comparisons of the I/O time of single operators are again made. '`broadcast`' is used as the canonical operator, and its performance is examined with both compile-time and run-time operand values.

The results serve not only as a comparison between data-dependent implementations for dynamic and scheduled routing, but also to illuminate performance differences between different implementations on the same hardware, highlighting the contention that it is important for the compiler to be able to choose implementations to match the mesh size and predicted message traffic volume.

This section also examines the hypothesis that virtual online routing using a scheduled routing substrate can be used as a final fallback when no other implementation is available, and without too many orders of magnitude difference in performance.

8.3.1 Latency

The first round of experiments compares a range of 2D meshes, with a one-word broadcast from node zero to all the other nodes. The times are wall clock times, from when the broadcast is initiated until the last node reads the last word of the broadcast. As for the results in the previous section, and for the same reasons, the scheduled routing curves include only the read+write portions of the overall timings. Figure 8-6 shows the timing results.



`(broadcast 'b (runtime) :m 1 :impl` *implementation*`)`

Figure 8-6: Broadcast implementations (one word, wall-clock time)

Two dynamic-router implementations are shown in the figure. The dyn_bcast implementation sends the message sequentially to all the destinations; this is clearly an inefficient technique in this case, but is the only applicable implementation in those cases with a runtime source and a sparse subset. The dyn_bcast_dense implementation performs better, using a flood-fill technique; it is the implementation of choice on dense grids, with either a compile-time or run-time source. The dyn_bcast_tree implementation, not shown, uses a tree to reach the destinations; it has similar performance to dyn_bcast_dense in this case (consistently around $1.2\times$ cycles), and is thus not shown in an attempt to reduce clutter. It is best used as an efficient way to reach a sparse subset from a compile-time source.

The remaining curves in the figure correspond to scheduled-routing implementations. The bottom curve, str_bcast, is the standard multicast implementation used for constant-source routing. The other two curves are two alternative run-time source implementations. str_bcast_linear is a low-overhead implementation whose router code is modified at operator load time to reflect the current broadcast source; it is clearly the run-time implementation of choice for meshes of less than about 60 nodes. However, its low constant is balanced against a linear implementation, since the implementation relies on sending data along snake-like paths that cover the entire mesh node by node. The str_bcast_dense implementation is the scheduled equivalent to dyn_bcast_dense, and shows similar performance, though less smooth, due to synchronizations between the processors and their routers at I/O time. Despite the relatively high constant associated with str_bcast_dense's runtime, it is $\Theta(\sqrt{N})$ (or $\Theta(\sqrt[3]{N})$ on a 3D mesh), and begins to outperform the efficient str_bcast_linear implementation around the 100 node mark.

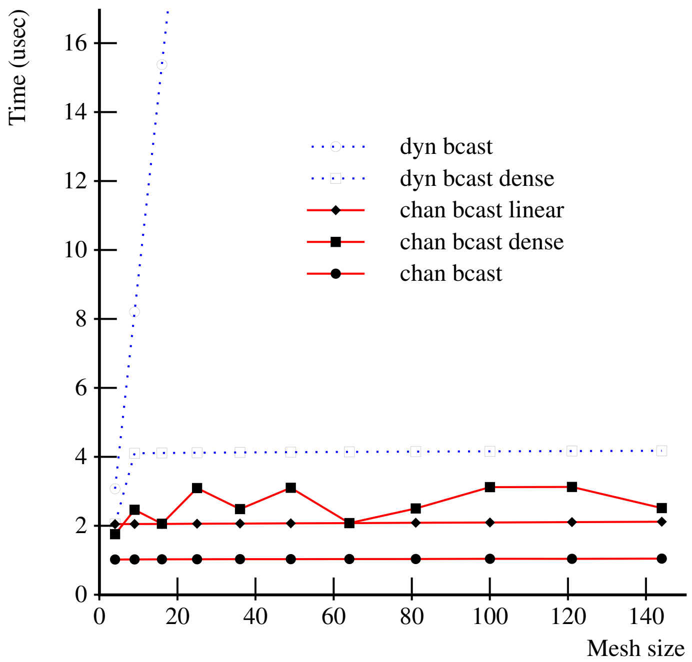
The graph shows that constant-source broadcasts are dramatically lower-latency with scheduled routing. Runtime-source broadcasts are better across the board (and, in fact, up to 60 nodes even take fewer raw communication cycles). A dynamic router with hardware multicast support would have the potential to run at similarly low latency, but the additional hardware would be likely to increase even further the critical path through the router that determines its cycle time.

8.3.2 Bandwidth

The other performance metric of interest is bandwidth. Figure 8-7 shows the time taken for a broadcast of 1024 words (4KBytes) using the same set of implementations as above; times are given in thousands of cycles.

The dyn_chan implementation, iterating over each destination, takes dramatically longer than the rest, going rapidly off the scale. The other dynamic implementation, dyn_bcast_dense, runs faster, at about 4 ns/word. The dynamic router substrate is limited by the fact that only a single outgoing message at a time can be composed; as a result, the dyn_bcast_dense implementation reads in a message and simultaneously forwards to its y neighbor, and then sends a second complete message to its x neighbor.

The three scheduled-routing implementations show a range of performance. The compile-time source implementation runs in 1 ns/word (2 cycles/word) using multicast; the routing bottleneck is the 'fork' in the router, which requires two cycles per word. The str_bcast_linear implementation does 2 ns/word as a result of the NuMesh restriction that the interface registers may only be accessed once per cycle from each pipeline. Finally, the str_bcast_dense operation performs roughly comparably to the equivalent dynamic operator on cycle-count, but with



`(broadcast 'b 0 :m 1024 :impl` *implementation*`)`

Figure 8-7: Broadcast implementations (1024 words, wall-clock time)

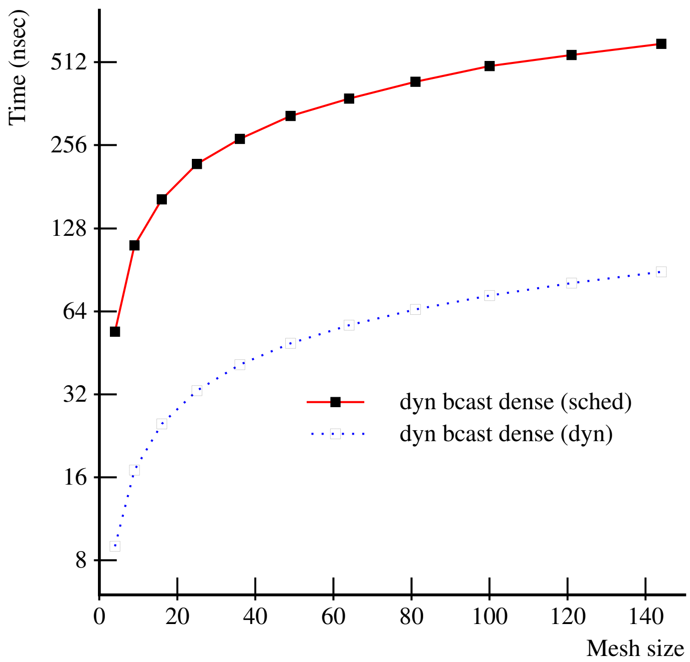
fluctuations in performance which result from stream scheduling requirements.

Again, scheduled routing is shown to perform at least as well as dynamic routing for a run-time source, and dramatically better for a compile-time source.

8.3.3 Online Scheduled Routing

The compiler must be able to fall back to online routing on the scheduled router substrate when no scheduled-routing implementations are available. In Section 4.2 it was claimed that support could be provided for online routing; this section looks at the performance. The broadcast operator is used as an example again; the dyn_bcast_dense implementation running native on the dynamic routing hardware is compared with the same implementation running with online routing support for scheduled routing, as discussed in Section 4.2. The data is presented (for a one-word broadcast) in Figure 8-8.

The figure shows that the scheduled routing layer is on the order of five-fold slower than dynamic routing in hardware. The wall-clock times are shown on a log scale to make it clearer that the curves are different by a constant factor. To assume a relatively faster clock rate for scheduled routing would decrease this discrepancy, but adding simple hardware-level dynamic routing facilities to the scheduled-routing model would most likely be necessary to achieve similar results for the two platforms.



`(broadcast 'b 0 :m 1 :impl 'dyn_bcast_dense)`

Figure 8-8: Online broadcast with dynamic vs. scheduled routing, wall-clock time

8.4 Application Benchmarks

While previous sections have considered communication kernels, a contention throughout this thesis is that scheduled routing provides most applications with communications that take a comparable number of cycles, and thus, given efficient hardware, shorter wall-clock time. The experiments in this section show end-to-end application time on two larger applications, matrix multiplication and Gaussian elimination.

Neither application is likely to do substantially better with scheduled routing, since their communications are mostly fairly local, with few hotspots. However, given the data-dependent nature of the communications, these applications serve to demonstrate that reprogrammable scheduled routing is capable of performing communications at least as rapidly (if not, indeed more so) than traditional dynamic routing for at least some class of data-dependent applications.

As the experiments in this section are intended to look at more than just raw I/O time for communication operations, the results are presented in a histogram format giving the contributions to the overall execution time of CPU, reads, and writes; the remaining scheduled-routing contributions are bundled into a single remaining category, 'overhead', since typically these only account for a few hundred cycles apiece.

8.4.1 Matrix Multiplication

A simple parallel algorithm is used to perform matrix multiply. The algorithm tiles a 2D array of processors with the elements of the array, then a broadcast is done on each row of processors of the appropriate piece of the A matrix, followed by a multiply of the appropriate piece of the B matrix, then a shift of the B matrices up one row. After one pass over the mesh, the matrices have been multiplied, and the result is left tiled in the same manner as the original A matrix. The COP code used is shown in Figure 2-6, p. 38.

The problem size for these results is a 60×60 mesh, running on meshes of up to 10×10 in size. While this may seem like a small problem (particularly for a 100-processor mesh), this size allows for the communication effects to be visible, while still showing speed-up from increasing processors. For broadcast on the scheduled router, the compiler picked either the str_bcast_dense or str_bcast_linear implementations (as discussed in Section 8.2); the circular shift was implemented as constant streams. The dynamic routing implementation used the dyn_bcast_dense technique for the broadcast.

Figure 8-9 gives the wall-clock running time of the algorithm using scheduled routing and dynamic routing.

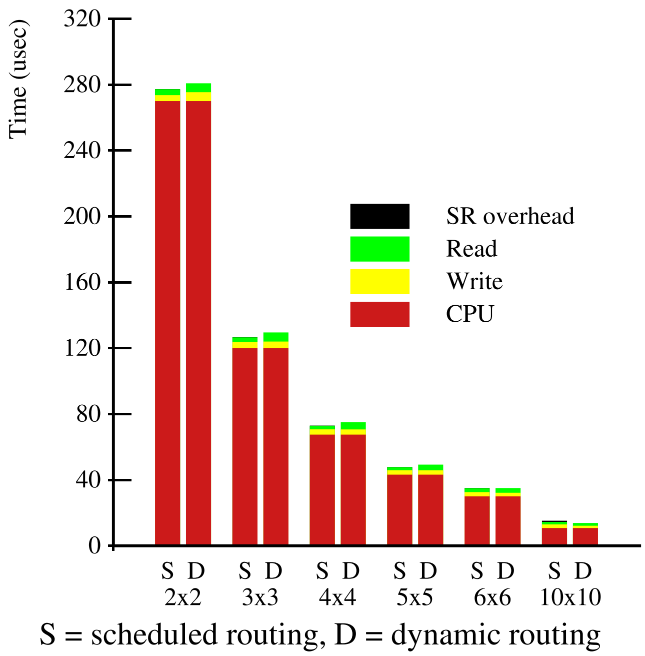


Figure 8-9: Matrix multiplication comparison, wall-clock time

For this example, despite the faster I/O time, there are in fact slightly more cycles spent doing I/O for the scheduled router. The extra cycles are accrued mostly as processor time spent waiting for writes to the interface. This is because the structure of both operators used here alternates reads and writes at a single-word granularity. For example, the broadcast reads a word from the source, then writes it to the destination. While the processor waits for the router to consume the last-written word from the interface, a new word is written from the source neighbor. When the processor completes the write, it finds the read data already waiting.

The reason for the slower cycles counts is, to a large extent, that Tadpole is unable to find good routes. Table 8.2 shows the bandwidths achieved for the two operators at each mesh size.

For the shift operator (in linear arrays of more than two nodes), Tadpole can't find a 1:2 packing for the circular shift. Performance could be improved here by improving Tadpole's

Size	broadcast	shift
2×2	0.500	0.500
3×3	0.458	0.333
4×4	0.458	0.333
5×5	0.458	0.333
6×6	0.250	0.333
10×10	0.250	0.333

Table 8.2: Bandwidths per operator in matrix multiply, wall-clock time

routing. Alternatively, and fairly easily, a simple schedule-generator implementation could be created that looked for circular linear shifts of this type and generated the correct schedules for them.

The broadcast operator's implementation also deteriorates with growing mesh size. This is due to Tadpole's inability to do stream rerouting. Another NuMesh router, `nusked`, which currently runs only in a stand-alone, static-stream configuration, is able to produce bandwidth-0.5 stream routes for both the 6×6 and the 10×10 configurations. Future work might include the ability to include multiple stream routers in the COP compiler, with known features and strong points (*e.g.*, `nusked` does not do multicast streams), and use the most appropriate one at stream-routing time.

The scheduled-routing overhead is found to consume a negligible fraction of the total runtime. Table 8.3 gives the complete cycle counts for the timing components for scheduled routing.

Size	CPU	Write	Read	PSync	Router	Reprog
2×2	270000	7205	7205	29	64	0
3×3	120000	7596	5472	51	69	0
4×4	67500	6388	4409	73	69	0
5×5	43200	5314	3686	227	69	0
6×6	30000	5191	4706	259	66	0
10×10	10800	4355	3052	1406	66	0

Table 8.3: Time breakdown for matrix multiply with scheduled routing

Router load time is only 65–70 cycles, since the data is downloaded once and cached for the remainder of the run time. The compiler chooses to put each operator in a separate phase to maximize operator bandwidth; both phases can synchronize by performing a message-synchronization pause, since the operators are sending enough data to be message-bound. Phase synchronization time is < 100 cycles for the smaller meshes, rising to about 1400 cycles for the 10×10 mesh (about 7% of the total execution time).

8.4.2 Gaussian Elimination

The final application tested here is Gaussian elimination, a technique for solving equations of the form $Ax = B$ with matrix manipulations. The algorithm is discussed in more detail in

Section 2.9, and the COP code is given in Figure 2-7, p. 39. A 210×210 array is used; the total time spent running this algorithm is thus much higher (on the order of millions of cycles) than for the matrix multiplication example.

For the scheduled-routing backend, the COP compiler generates a single phase for the initial reduction/broadcast step (since both are low-traffic operators), then one phase per operator after that. Timings are shown in Figure 8-10.[2]

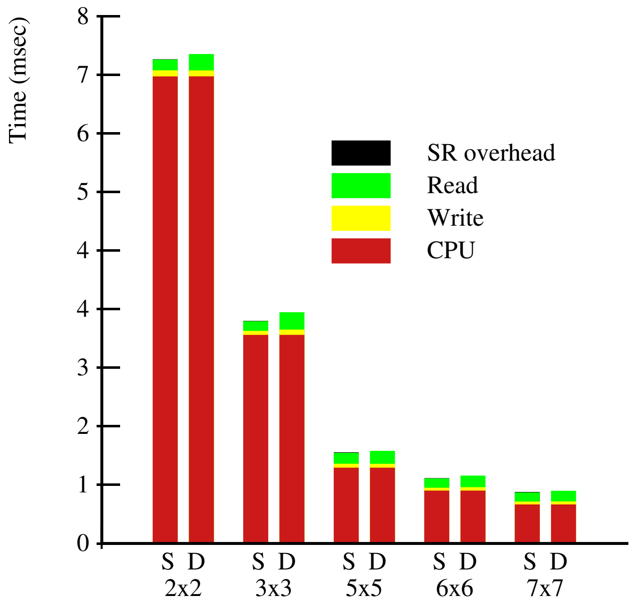


Figure 8-10: Gaussian elimination comparison, wall-clock time

These results are largely similar to the results for matrix multiply. Again, the I/O takes slightly more cycles, mostly accrued by the reads from the interface. The scheduled-routing overhead is still low, though it is just perceptible in the figure; a complete breakdown of the scheduled-routing cycle counts is given in Table 8.4.

Size	CPU	Write	Read	PSync	Router	Reprog
2×2	7177449	185754	332912	7958	144	0
3×3	3205323	123529	297307	8686	153	0
5×5	1164685	120804	335352	16145	149	0
6×6	812548	87413	285552	9386	148	0

Table 8.4: Time breakdown for matrix multiply with scheduled routing

A jump in read time and phase-synchronization time can be seen at the 5×5 mesh size; at this point, the mesh becomes larger than can be synchronized with message-bound operators for the phase with the column-zero broadcast (`b5`), and a barrier is introduced to coordinate the end of the phase. The phase-synchronize overhead is still only $< 1\%$ of the total runtime, however. No reprogramming time is accrued, since the compiler determined that dense broadcasts would be most appropriate for this context.

[2]Results for 7×7 and 10×10 scheduled-routing meshes are not currently available; Tadpole crashes.

Chapter 9

Conclusions

The goal of this thesis was to demonstrate that a reprogrammable scheduled router could be used for arbitrary applications, and show significant speedups over traditional routing architectures. A wide range of techniques for performing communications compilation for a reprogrammable scheduled router were shown, primarily involved with managing multiple communications phases and efficiently implementing data-dependent communications. As part of this effort, a communication language, COP, was designed and implemented, to keep computation compilation separate from communication compilation but still provide much of the necessary compile-time information on communication patterns to the communications compiler. The remainder of this chapter presents the conclusions of the thesis in more detail; the chapter ends with a brief look at possible future work.

9.1 Managing Multiple Phases

A primary focus of the thesis is the management of multiple communication phases in a scheduled router.

Continuing Operators

One important caveat on changing phases is that there may be some ‘continuing operators’ that do not change. In the normal case where the next phase includes all the nodes that an implementation uses to route, a required path is simply passed to the stream router, then the same VFSMs are reused for that stream in the new phase. If the next phase is fragmented into smaller subphases (for example, because there are disjoint groups), the router is instead forced to use the exact same schedule for that operator, requiring it to work around those schedule slots when creating the schedule for the new phase.

Changing Phases

Changing phases is not as simple as just issuing a command to the router. The compiler must determine when each node has finished moving all the data belonging to the terminating operators from the phase. This thesis presents a procedure for choosing methods to close a phase that make that guarantee:

- Using the synchronizing properties of long messages to guarantee that other nodes in the mesh are close to completing the phase;
- *Hijacking* an existing set of streams in order to create a long-message synchronization across nodes before changing phases;
- Using 'processor-bound' implementations to allow local phase changes (but restricting wire use in the next phase);
- or adding an explicit barrier operator to terminate the phase.

Determining which technique to apply requires finding out which operators in a phase are terminating, as well as which operators might run last—a non-trivial task, given the presence of optional operators and the possibility of multiple portions of a COP input hierarchy being present in a single phase. With that information, the message lengths of the operators determine whether long-message synchronizing is viable.

If not, a minimum spanning tree algorithm can be used to find a set of streams which can be hijacked for use as a synchronizing barrier; multicast streams which reach multiple nodes are preferred for their lower latency and data-buffering capacity, but any set of streams that can reach all the nodes in each partition of the mesh is acceptable. If necessary for a given operator implementation, a provided function can be called to undo any implementation-specific modifications to the stream, making it available for synchronization purposes.

Inter-Phase Wire Constraints

An important consideration is avoiding the re-use of inappropriate $\langle$*wire/timeslot*$\rangle$ pairs. If one phase is writing a particular direction at a particular time, the compiler must be careful to ensure that, after a phase change, data can not be transferred *between* phases to unrelated operators.

The thesis sets out the restrictions that are necessary to avoid this kind of problem. In general, a given $\langle$*wire/timeslot*$\rangle$ can only be safely reused if the node used to be reading from it, and the operator that was using it has terminated. However, when the previous phase ended with some form of global barrier, any $\langle$*wire/timeslot*$\rangle$ that was used for writing can be reused, since by the time the new phase starts the neighbor will have stopped reading on that $\langle$*wire/timeslot*$\rangle$. The effect of aliased valid/accept hardware lines is also considered for the case where two nodes might try to read simultaneously during a phase change.

The thesis also explores the notion of wire-use dependency when there is no global barrier. A dependency graph can be set up for each operator/node pair, with two nodes each, representing 'I/O' and 'data arrival'; dependency edges are then created between those nodes, and the resulting graph is used to reason about neighbor phase transitions based on the operators that are currently running. Once an operator has been shown to run only if the neighbor has changed phase, that operator can freely reuse edges as if the previous phase had ended with a global barrier.

Managing a Scheduled Router as a Cache

Since the router has a finite amount of memory for schedules and VFSM annotations, the compiler must attempt to partition and use that space in a way that optimizes the total run time. The router can be treated as a *cache* for routing data that is held in the processor's memory. At run time, before starting a new phase, the COP code checks whether the schedule and VFSM annotations for that phase are currently in the router; if not, it downloads the data from the processor.

Determining the allocation of schedules and VFSMs to *slots* in the schedule memory and VFSM annotation memory is similar to the way that the 'optimal' cache strategy for conventional direct-mapped caches is managed, but with several complicating factors. First of all, the application does not provide complete knowledge of the future; there are both optional operators to consider, and operators with run-time arguments that induce different phases to be downloaded. Second, VFSMs are cached independently, but must be all present at the time a phase begins; thus, they must be managed as a unit. Ideally, when one VFSM is evicted to make room for another, the compiler also attempts to evict other VFSMs from the same evicted phase to hold the remaining VFSMs from the current phase, to minimize VFSM reloads.

Finding Load Operators

Once all routing and caching decisions have been completed, the question remains of where to place the router load information in the application. The per-operator *load* function must be placed appropriately in the application code, and this function gives the COP compiler a hook on which to hang reprogramming information.

The thesis examines the question of which operator to use when a group of parallel and sequential operators are all included in a phase. Doing this requires taking into account that some nodes may be involved in the routing of some operators and not others, and thus that the load function for a given phase may vary from node to node. It also considers the case where some load function must be responsible for loading multiple phases of data: for example, a red/black style broadcast on alternate phases would require each node to load two phases' of routing data, running the appropriate barrier between the two so as to know when to switch.

Results

Two large applications were examined, both requiring multiple phases of communication for optimal run time. The two applications were matrix multiplication and Gaussian elimination. In both applications the compiler was able to find an optimal grouping of operators into phases, choose barrier methods, and correctly allow for wire reuse in following phases. Caching techniques were applied to find mappings of virtual schedules and VFSMs to the physical hardware. Running the application showed roughly similar cycle counts, and factoring in the predicted speedup of scheduled routing showed overall significantly faster communications for almost all of the simulation runs.

9.2 Managing Data Dependency

The other key focus of the thesis is how to support data-dependent operations on a scheduled router. This thesis presents a wide variety of approaches, including full dynamic routing.

Operator Implementations

With scheduled routing, there are essentially two high-level ways to implement an operator: either express it as a set of (potentially modifiable) streams, or generate a complete, global schedule. The first technique allows multiple operators to share a phase (if determined to be optimal, or if required by parallel grouping); the second technique allows for the use of implementations that are faster, that take advantage of the hardware more effectively, or that support run-time operators that can't be handled with modified streams.

There are many ways to implement data-dependent operators under reprogrammable scheduled routing, and while the compiler does not implement all of them, the thesis details a wide range of techniques to support such run-time values. The thesis presents a range of ways to use reprogramming on existing streams or to use run-time schedule generation, as well as a pair of *meta-implementations* that allow the generated COP code to choose which flavor of a phase, or which version of a stream, to use for communications.

Online Scheduled Routing

The thesis examines in more detail the techniques necessary to support online routing on a scheduled-routing substrate. Providing an arbitrary routing method is required if scheduled routing is to support arbitrary applications. In fact, when the communication sources and destinations change rapidly, relative to the message size, it may be more efficient than other implementations.

Support for online routing involves using the processor to forward messages over nearest-neighbor streams. The online routing streams are allocated based on the demands of the operator implementations using those streams, so varying bandwidths and numbers of interface addresses will be allocated throughout the mesh. Other, statically-routed streams may also co-exist with the online-routing streams, for operators that do not need the online-routing functionality. For simplicity, availability of data-arrival and data-departure interrupts is assumed, and the online router is structured as a (processor-level) interrupt handler.

Results

As shown in Chapter 8, testing of the compiler on the NuMesh platform showed that it allowed much faster implementations of certain communication primitives, both by cycle count and by wall-clock time. A representative operator (broadcast) with a data-dependent operand was examined, and scheduled routing was found capable of implementing the operator consistently faster than dynamic routing.

Online routing on a scheduled router was demonstrated to function perhaps five times more slowly than on a dynamic router, but the result gave a lower bound on the possible slowdown of any application running on a simple scheduled router. The addition of dynamic routing hardware to the scheduled router would bridge the gap for applications requiring online routing.

9.3 The Compiler Search Engine

The techniques and algorithms presented so far would not represent a complete compiler without a structure for making decisions among the various implementations and phase breakpoints. This thesis presents a notion of per-operator 'extents' which allow the compiler to pick when to allocate separate threads of control to sets of operators, as well as a search-based algorithm for finding optimal groups of implementations.

Multiple Threads of Control

A desirable feature of scheduled routing is allowing for disjoint groups of operators to execute independently. These disjoint groups are not explicitly specified to the compiler, but rather derived from parallel groups of sequentially-grouped operators.

With scheduled routing, it is critical to determine which nodes support the routing for an operator. One of the powerful features of scheduled routing is its ability to find good paths for messages at compile-time, avoiding hotspots in the mesh. This need is balanced against the desire to allow multiple threads of control by using *spatial extents* to control disjoint groups. Each extent corresponds to a subset of the mesh that the compiler can use to route that operator, if it is sharing the mesh with disjoint groups.

Using extents to support disjoint operator groups also vastly simplifies the communication compilation process, since doing so avoids overloading multiple communication phases on particular nodes located between two disjoint groups. The need to reserve bandwidth for multiple phases (spatially and/or temporally distinct) would quickly lead to diminishing returns; the extent technique avoids exploring that part of the communications implementation space, but provides the benefits of allowing disjoint communications groups in a straightforward manner.

Searching the Implementation Space

This thesis presents a structure for searching the implementation space jointly for phase breakpoints and operator implementations, using a cost function to predict performance. A branch-and-bound search algorithm using this framework is also presented.

Disjoint groups are handled during the search, converting the leading and trailing phase in each disjoint phase-group to a single phase at either end used to allow correct handling of operators that span the initial disjoint operators, but terminate in their first phase. The disjoint management technique also allows simplification of the compiler structure, since it results in only a single phase being *active* at any point during the search process.

Closing a phase during the search induces a determination of the appropriate barrier to use for the phase (as discussed above), followed by a computation of the predicted time to execute the phase. The timing can be broken down into three components: the sum of the I/O time predicted for all the operators, the time taken by the phase change, and the time to perform any router programming operations. Summing these three values yields the overall predicted time for the phase.

Phases composed of routable streams require extra computation to get a predicted timing from; a stream-alias implementation has no explicit bandwidth associated with each stream, since bandwidth on each stream is determined jointly by all the streams' demands. An algorithm is presented to make a fast estimate of the bandwidth that will be allocated to each stream, based on a notion of *resources* (node I/O or bisection bandwidth) that are shared jointly by all streams. A single bandwidth value is associated with all streams of an individual operator; dynamic-routing streams for an operator are specially handled to reflect the fact that they are not all present in the mesh at once (unlike for scheduled routing).

Stream Routing

COP relies on a *stream router* to provide static-stream, single-phase routing services for scheduled routing. It invokes the stream router multiple times, once for each phase (or more than once for phases with multiple partitions that perform independent barriers). A key interaction between the stream router and COP is the *cost function* used by COP to disallow the use of certain wires at certain times (or to provide an incentive to use another wire if possible). This cost function is used to avoid implementation-specific restrictions for particular scheduled-routing substrates, as well as for supporting the wire-reuse restrictions discussed above.

Back Ends

The NuMesh backend requires some specific, idiosyncratic management by the compiler. In particular, with NuMesh reprogramming is done with a separate scheduled VFSM that reads address/data pairs from the interface and applies them to memories in the router; this requires support from the stream router cost function (to ensure that the reprogramming VFSM can be scheduled). The other major restriction in NuMesh is that the processor interface can only be read or written by each pipeline in each cycle, not both. Thus, avoiding schedules that might cause this problem requires careful handling, again implemented through the stream router cost function. Techniques to handle other minor NuMesh idiosyncrasies are also presented in the thesis.

The dynamic-routing backend requires relatively little additional support. The most significant work done to handle this platform is to generate legal interface addresses on each node. Unlike scheduled routing, where interface addresses are bound to a particular phase, with dynamic routing interface addresses can't be trivially allocated to operators. Instead, the dependency analysis introduced to handle $\langle$*wire/timeslot*$\rangle$ reuse in scheduled routing is used, and extended to perform an n-coloring of the interface addresses. An (unimplemented) technique for *virtual registers* is also proposed, to use when the dependency analysis fails to limit the number of distinct addresses to the number provided in hardware.

9.4 COP

The design of a language to express communications in a form suitable for deriving router schedules is a significant part of the results. The language expresses much of the structure of an application's communication, though naturally some information is lost. The language serves as an excellent research vehicle for scheduled routing, and appears to include most of the necessary information to generate good communications.

Communication Languages

One might think that the only feasible approach to managing a reprogrammable scheduled router would be a top-to-bottom compiler for a high-level parallel language. In fact, this thesis demonstrates a way to separate the demands of the communication substrate from the high-level demands of the application, such as partitioning, placement, and processor-code optimization. Using a separate communication language allows a communication substrate that is a non-trivial compilation target to be *separately* compiled, thus increasing the ease with which independent high-level languages can be targeted to the substrate.

The interface to the communications compiler should be a clean one. The COP interface effectively converts each communications operator specification into a set of functions for I/O, plus a function used as a hook for router management functionality. Using the returned functions instead of explicit communications operations, the high-level language compiler can ignore the details of communication for a given hardware substrate.

Communication languages in the past have typically had very limited expressibility. They typically only have basic functionality for parameterization, and little other higher-level means of expressing communication structure compactly. By making COP a Lisp-based language, the full power of a Turing-complete language is available to specify the communications patterns for an application. While a high-level language compiler may not use such features in great depth, they are nonetheless useful, particularly for users who choose to write communications code by hand, *e.g.*, using the C/COP compilation model demonstrated in the Results section.

High-Level Operators

The great majority of existing communication languages only allow simple streams to be expressed, *i.e.*, single point-to-point connections. This is problematic at several levels:

1. Some communications inherently do not have this characteristic; a broadcast (or multicast) is just the most obvious.

2. More importantly, specifying communications at the stream-by-stream level defeats the goal of allowing the communications compiler to optimize communications to match the hardware.

3. Finally, for runtime communication patterns, they can rarely be expressed at such a low-level as run-time stream sources and destinations; rather, they must be expressed as run-time broadcast sources and the like.

COP includes a wide range of high-level operators, as well as the facility to define additional implementations of existing operators.

In addition to defining communications at the operator level, COP includes a notion of operator *annotations*, specifying characteristics of the operator that go beyond the type and the sources and destinations of the communication. The user can specify, *e.g.*, the message size and predicted number of messages, whether the operator will be used in a pipelined or round-trip manner, or a maximum bandwidth for an infrequently-used operator.

Abstract Operator Sequencing

A critical feature of communication languages is to express a notion of time-varying communications. Languages without any such feature can only support the simplest and most regular of communications patterns, such as pipelined DSP computations.

Existing languages with support for time-varying communication support mostly just user-specified phases of communication. While this is a step in the right direction, it still represents an attempt by the user (or high-level language compiler) to out-guess the communications compiler.

In COP, by contrast, operators are grouped hierarchically into sequential and parallel groups only—parallel constructs may use any operator in any sequence, whereas sequential constructs require the application to use the operators in the specified sequence. The compiler then takes the specified relationships among the operators and chooses an optimal grouping of the operators into phases. (The user may force phase groupings on the compiler with the `(phase)` construct if desired.)

Along with expressing hierarchical grouping comes the potential for operators that may be active in multiple phases of the computation. Such operators are important to handle, and COP correctly handles keeping data flowing through them in multiple phases; it even allows their bandwidth to vary to match different traffic demands from other operators in each phase, where possible.

When specifying operator sequencing, it must also be possible to express groups of operators that have no well-defined sequencing, because they execute on different parts of the mesh. COP handles this by transforming parallel and sequential groupings into *disjoint* groupings where possible, then handling disjoint groups of operators completely independently.

Run-Time Values

One of the most important features of a general-purpose communication language is that it allow data-dependent communications to be expressed. Existing communication languages are either fully compile-time or fully run-time. Since exclusively run-time communications comes with a significant overhead under scheduled routing, it is critically important to express run-time values in a limited and well-understood manner which can be capitalized on at compile time.

COP allows this by allowing the user to substitute a `(runtime)` construct in lieu of operator operands. Implementations that can handle such "values" can then create the necessary run-time structure to get the value from the application at run-time and handle it appropriately. The high-level language does not need to know whether the COP compiler handles the value by generating router code at compile-time, at run-time, or some of both.

To allow the compiler to choose the best possible options, the user can specify optional arguments specifying additional information about the run-time value. For example, the user can specify that the argument is not just a 'legal node value' (the default assumption for node operands), but that is restricted to a given list of possible nodes. Similarly, the user can instruct the compiler as to how often the value is likely to change at run-time, so that the compiler can weigh the tradeoffs (for example) of better bandwidth *vs.* higher overhead.

9.5 Future Work

As is no doubt typical with compiler projects, the amount of future work depends only on the creativity of the compiler architect. In this section, a few interesting directions for future work are outlined.

Dynamic-Routing Support

An intriguing issue is the extent to which hardware dynamic routing support is beneficial for certain applications. Adding a hardware dynamic router to the scheduled router would allow for applications with short-lived dynamic communications to exhibit about the same level of performance as with a traditional dynamic router.

Schedule Length Selection

Currently the schedule length (or periodicity) is given by the user, either as a compiler switch or in the COP input. Although this simplifies the code generation, it also limits somewhat the flexibility of the compiler. A future extension to the compiler might consider a variety of criteria in an attempt to pick a good schedule size for an application:

- the presence of schedule-generators that require a schedule to be an even multiple of their base schedule length;
- the number of phases in the inner loop of an application, as compared to the total schedule memory size of the target;
- or the likelihood of decreasing performance with decreasing schedule size, based on the number of streams passing through each node.

Though not easy, schedules of varying sizes might also by supported. For example, a phase with a large number of streams might benefit from a larger schedule, whereas the rest of the application might do fine with smaller schedules, leaving a smaller footprint in the router. Trying to transfer to a phase with a longer schedule would require real synchrony from the nodes, rather than integer-multiple synchrony such as is now used. The phase change could consist of a reduction followed by a broadcast that switched the router's schedule without processor intervention; this would require the schedule to be at the same address on all nodes. Switching back to a phase with a length that evenly divided the current schedule's length would not require any special handling.

Operators and Implementations

There are a wide range of interesting implementations to test. For example, currently a permutation with a run-time argument must be managed via online routing. As mentioned in Appendix B, schedules for a run-time permutation could be built at run-time, thus improving latency dramatically, assuming the operator was used enough to justify the time spent deriving a schedule for it.

Meta-implementations are discussed in Section 4.1.3. Future work should include an implementation of this concept, and testing to compare it against existing implementations.

It might be worthwhile to make operators easier to add; for example, an 'operator object' could be defined that knew about the kinds of operands it took, and so forth. Then it would be as easy to add a new operator to the compiler as it is currently to add a new implementation.

Language Features

The definition of `(runtime)` could be extended to allow it to apply to the optional arguments presented to operators; for example, an array of streams could be specified with `(runtime)` maximum bandwidths, then at load time set some stream's bandwidths to zero to give more bandwidth to streams that will actually be used.

Taking that one step further suggests the ultimate run-time feature: eval. The entire compiler could be linked in with the application, and pass text strings holding operators to the eval function, which would return a tuple of function pointers corresponding to the necessary I/O functions and load function.

Appendix A

COP

This appendix provides additional details on COP's syntax and semantics.

A.1 Primitive COP Operators

```
(reduce LABEL FUNC DEST)
(allreduce LABEL FUNC)
(prefix LABEL FUNC)
```

functional. FUNC is a function that is called internally by the COP-generated code to 'sum' two data items. The function should be suitable for a reduction operator; it is passed values in the same manner as they are passed to the regular I/O functions. For array operands, the function should take three arguments (two inputs and an output); for non-array operands, the function takes two inputs and returns the output. DEST is the node that receives the answer. It can be constant, runtime, or runtime distributed. For plain `reduce`, all the nodes but the specified destination will read some invalid result, which they should discard.

```
(broadcast LABEL SRC)
```

directional. This operator sends data to all the nodes in the selected set from the specified SRC node. It may be constant, runtime, runtime distributed, or runtime dynamic.

```
(collect LABEL DEST)
```

directional. This operation (in some sense the opposite of broadcast) allows any of the nodes in the subset to write the operator, and the destination is the only node that can read it. The operator is only guaranteed to work correctly if one node at a time writes to it, although it may be that in some circumstances multiple nodes can successfully write to this operator. The DEST can be constant, runtime, or runtime distributed.

```
(barrier LABEL)
```

functional. All nodes in the subset will pause until all nodes have called the function.

```
(cshift LABEL DISTANCE)
(eoshift LABEL DISTANCE)
```

functional. These operator types shift data within the selected subset. `cshift` ("circular") applies shifts modulo the size of the subset, whereas `eoshift` ("end-off") discards any shifts that leave the subset. The distance argument may be constant or runtime.

```
(permute LABEL PERM)
```

functional. This operator specifies a permutation of the subset. For compile-time arguments, PERM must be a list of nodes the same length as the subset size; the nth node in the list is the node to which node n sends its data. The argument may also be runtime (in which case all nodes must pass the permutation to the load operator) or runtime distributed (in which case each node only knows the node to which it is writing).

```
(stream LABEL SRC DEST)
```

directional. This operator connects the SRC to the DEST. Either of the nodes may be runtime; additionally, the DEST may be runtime dynamic.

```
(spread LABEL SRC)
(gather LABEL DEST)
```

directional. With the `spread` operator, the SRC node writes once to each node in the subset (excluding itself); the other nodes read once to get the appropriate value from the SRC node. For the `gather` operator all the nodes in the subset (except DEST) write once, and the DEST node performs $n - 1$ reads to get all the values in order.

A.2 Compound COP Operators

Unless otherwise indicated, these operators do not support run-time values.

```
(grid '(LABEL ...) LEN OFFSET)
```

This operator type creates a grid of communicators. The list of LABELs must be twice the number of physical dimensions in length. The LEN specifies how far each grid link goes, and OFFSET specifies how frequently the grid links are spaced. (Thus, OFFSET less than LEN means overlapping grid links.) OFFSET and LEN may be lists of size equal to the dimensionality of the mesh to make their values different in different dimensions.

```
(transpose LABEL SUBSET SUBSET)
```

The `transpose` operation allows the user to specify two subsets to transpose data between them. Normally, the two SUBSET arguments are the same as for a `subset` argument, and the operator is converted into a `permute`. Alternately, one subset or the other may replace all the subset lists with single nodes, resulting in either a `gather` or `spread` as appropriate.

```
(bitreverse LABEL)
```

This operator creates a permutation connecting each node in the subset to the node that has the address corresponding to the bit-reversal of the source node's address.

```
(bistream LABEL1 LABEL2 NODE1 NODE2)
```

This operator creates streams going each way between the two nodes. The same label may be used for both streams, if it is otherwise legal to do so. Run-time values are acceptable as arguments.

A.3 Optional Operator Arguments

Every operator can take optional named arguments providing information on message length, message count, message type, and so forth. This section lists the complete set of optional arguments.

`:type TYPE` (default `int32*`). The type argument sets the interface for arguments and return values for this function; this is only relevant if the tuples returned to the frontend are in a typed language (*e.g.*, C). Legal integral types are `int32`, `float32`, `int64`, and `float64`. Appending the `*` character to the end of the type name implies an array of the given type.

`:m MSGLEN` (default 1 or 2 to match `:type`). The message length is the amount of data transferred in a single operation. For example, a 64-bit prefix operator would specify "`:m 2`"; a permutation of 1024 double floats would specify "`:m 2048`". The message length must be exact for each operator; if multiple message lengths are desired, multiple operators with otherwise identical parameters should be used.

`:c MSGCOUNT` (default 1). The message count is the number of times the operator will be invoked (*i.e.*, read or written) within the inner-most enclosing loop. Taken together with the message length and the enclosing loop counts, this provides the total 'traffic count' through the operator. Like the loop counts, this only needs to be an approximate value, but it is very useful in attempting to tune the choice of an implementation.

:pipelined BOOL (default false). Normally when the compiler is computing approximate costs to determine how long a given implementation of an operator will take, it looks at the total time the operator will require to run, from first data send to last data receive. If the :pipelined flag is set, it indicates that the application uses the operator in a pipelined manner; *i.e.*, it writes or reads the operator back-to-back without any implied synchronization.

:b MAXBANDWIDTH (default 1.0). Normally bandwidth is calculated for an operator by using its traffic count; comparing this value with that of other operators for each resource that the operators share allows the relative bandwidths for all the operators to be determined. However, the :b argument can specify a maximum bandwidth to use, regardless of any relative traffic values discovered by the compiler. The special value *min* can be used, which means that the operator need not be invoked more than once in the scheduler loop regardless of how long the loop is.

:impl IMPLEMENTATION (default none). If the user wishes to force a particular implementation of an operator for some reason, this can be specified with this argument.

A.4 (runtime)

Some optional arguments may be provided to a (runtime) clause.

:distributed t means that the value of the argument is not fully known on all nodes, but rather that each node only knows the part of the argument relevant to itself. The default is false.

:block N allows the language to specify that a given runtime value will only change approximately once every N times through the innermost enclosing loop. By default the compiler assumes it will change each time through the loop.

:values LIST specifies the possible values that the argument can take on (which must, obviously, be a subset of the permissible values for that argument). Omitting this argument is equivalent to specifying all possible values.

:dynamic t is a stronger version of :distributed. This argument specifies that the writing node of a directional operator may call the *load* function additional times with altered values of the specified operand, without assuming that other nodes will do likewise. Currently this argument is only legal in a few places, including a runtime stream destination or a distributed broadcast source.

For operators with run-time values, the *load* function takes argument(s) corresponding to the run-time values. The values are provided in the same order that they appear in the COP operator, in machine-native format. Integer or floating-point values are passed in the obvious way. Functions are passed by name at compile time but by pointer at run time. Arrays of objects are passed as a pointer to a contiguous array of the objects terminated by an illegal value, such as -1 for node numbers.

A.5 COP Grouping Constructs

There are a variety of constructs for grouping operators and providing semantic information about them. OP is used throughout to indicate either operators or further hierarchical grouping constructs.

`(sequential OP ...)` Indicate operators will be run sequentially.

`(loop LABEL COUNT OP ...)` Indicate operators will be run sequentially in a loop. The LABEL provides a label for a *load* function returned by the compiler. The COUNT is an approximate number of times through the loop.

`(parallel OP ...)` Indicate operators will be run in parallel.

`(doall ((VAR VAL) ...) OP ...)` This constructs a parallel group containing multiple versions of the listed operators; all the operators are placed in the group once for each different combination of the specified variables bound to the specified values.

`(optional OP ...)` Indicate operators may or may not be loaded and used at run time.

`(phase OP ...)` This sequential construct forces the compiler to place no phase breaks after each operator in the list except the last, after which a phase break is forced.

A.6 Predefined COP Variables and Functions

COP defines a number of variables to indicate the context of a COP program. They are listed in Table A.1.

Name	Description
`*nodes*`	Number of nodes in current subset
`*xphys*`	Mesh x dimension size (y, z similar)
`*xsize*`	Current subset x dimension size (y, z similar)
`*subset*`	Current subset
`*mode*`	Compilation target (*e.g.*, `dyn` or `numesh`)
`*schedlen*`	Schedule length (for numesh mode)

Table A.1: Pre-defined COP variables

The first four variables in the table describe the mesh geometry. They can be updated to force a given mesh size for compilation (overriding the command-line arguments). The `(reset-mesh-size X Y Z)` function resets all the geometry variables at once. The `*mode*` variable can be set to force compilation to a given target. Similarly, the `*schedlen*` variable may be set to force a particular schedule length; this variable is only predefined when the compiler is generating scheduled routing code.

A number of functions are available in the Lisp environment beyond those normally available in standard Lisp. Table A.2 lists some of those functions.

Function	Description
`(all` n`)`	Return a list $(0, 1, ..., n-1)$
`(count` i_s s`)`	Return a list $(i_s, i_s+1, ..., i_s+s-1)$
`(step` s i_s i_e`)`	Return a list $(i_s, i_s+s, ..., i_e)$
`(coord` x y z`)`	Return node index for coordinates x, y, z
`(coords` n`)`	Return list of coordinates for node index n
`(make-1d` dim`)`	Return subset set with coords varying in dimension dim

Table A.2: Pre-defined COP functions

A.7 COP Extensibility

As discussed in Section 2.8, it is possible to extend the language in several ways.

Lisp Implementations

These implementations are generally fairly simple. The `bistream`, `grid`, `transpose`, and `bitreverse` operators are all implemented entirely in Lisp. As the simplest example, here is the definition of the `bistream` operator:

```
(define-op (bistream label1 label2 src dest)
  (parallel
    (call-op stream label1 src dest)
    (call-op stream label2 dest src)))
```

The `define-op` macro sets up an appropriate lexical scope for the newly-defined operators, allowing the various optional arguments (`:m`, `:type`, *etc.*) to have their default values. The `call-op` macro invokes the given functions, passing the optional arguments to them in a form of limited dynamic scoping. The resulting operator returns a set of streams accessible using the passed-in labels.

C Implementations

Creating a new C implementation is straightforward. The user simply writes a standalone module, including `<cop.h>`, and compiles and links it as a shared library. Then, at compile-time, the user specifies the '`-Ifilename:definition`' switch, passing the filename of the shared library and the C name of the implementation definition structure. The compiler will add that implementation to its existing set and use it as appropriate in the generated code.

For all C implementations of functional operators, the *func* functions, when passed the same argument by reference for both input and output, must correctly reuse the pointed-to buffer.

A.8 Sample COP Idioms

This section presents a few snippets of COP code to demonstrate some useful idioms.

A set of streams can be created such that each node can write to and read from the `xor` operator to communicate with the node whose address is the same as its except for the low-order binary bit.

```
(doall ((i (all *nodes*)))
  (stream 'xor i (logxor i 1)))
```

Broadcasting a value from the diagonal of a matrix to all the rows, in parallel, can be expressed tersely as follows.

```
(doall ((i (all *ysize*)))
  (subset (list (count (* i *xsize*) *xsize*))
    (broadcast 'b i)))
```

As a final example, Figure A-1 is the COP code used to create nearest-neighbor connections on a 3D mesh. Alternately, the connectivity of a torus could be expressed by removing the `if` statements, and adding an extra `mod` function around the increment statements. Only six operators are defined in this example, one for each direction: `x-plus`, `z-minus`, and so forth.

```
(doall ((x (all *xsize*)) (y (all *ysize*)) (z (all *zsize*)))
    (if (< x (- *xsize* 1))
        (bistream 'x_plus 'x_minus
            (coord x y z) (coord (+ 1 x) y z)))
    (if (< y (- *ysize* 1))
        (bistream 'y_plus 'y_minus
            (coord x y z) (coord x (+ 1 y) z)))
    (if (< z (- *zsize* 1))
        (bistream 'z_plus 'z_minus
            (coord x y z) (coord x y (+ 1 z)))))
```

Figure A-1: COP code for specifying nearest-neighbor connections

Appendix B

Operator Implementations

This appendix lists implementations of the operators. Implementations that are not yet implemented in the compiler are marked with a dagger (†). Each implementation is listed with its 'canonical name' (as would be used as an argument to `:impl`), followed by any restrictions on its arguments, and then some descriptive text. Restrictions are typically either that an operand must be constant, or that it must be non-distributed (*i.e.*, a runtime value that must be known on all nodes). Some schedule-generators also require that the subset be connected (*i.e.*, all nodes in the subset can be reached from each other without leaving the subset). Some of the support code used by the implementations is discussed briefly at the end of the appendix.

B.1 Implementations

`broadcast`

str_bcast: *constant source*. A stream-alias multicast stream implements this directly.
str_bcast_bind: *constant source*. Like str_bcast, but the source reads its own output, thus making it processor-bound.
str_bcast_linear: Two linear snakelike multicasts through the mesh are created. One multicast starts at one end and delivers to all the nodes; the other starts at the other end of the subset. The *load* function rewrites the router to either accept data from the interface register (when the node is the source), or forward the data through. Higher-dimensional versions of this implementation might be feasible, *i.e.*, a 2D version that multicasts along dimensions and at load time makes streams forward to each other by rewriting. This implementation supports a dynamic broadcast source only for messages of length one, since otherwise there is no guarantee that messages will not be fragmented.
str_bcast_tree†: Do a constant collect to a known central root and then a constant broadcast. The implementation rewrites the stream router output to automatically forward the incoming streams at the collect destination to the outgoing broadcast streams.
str_bcast_dense: *non-distributed source, convex connected subset*. Streams are allocated from all nodes to their neighbors (using a single interface register for reads); a broadcast is then done by reading from the 'read' register (if a reader) and writing to the appropriate neighbors based on where the source of the broadcast is known to be.

sch_bcast_dense†: *non-distributed source; connected subset.* All the nodes store a few schedules based on where the source might be relative to them, and at load time download the appropriate schedule.
dyn_bcast: Send the message one at a time to the other nodes in the subset. Best for very small subsets, or a dynamic source.
dyn_bcast_dense: *non-distributed source, convex connected subset.* As for str_bcast_dense, using dynamic routing.
dyn_bcast_tree: *constant source.* Create a reduction tree, with the root at the source, and forward the data throughout the mesh using the tree.

`collect`

str_collect: *constant destination.* A reduction tree is created to get a set of streams that are used to pass the data up the tree. All the junctures on intermediate nodes are rewritten after the stream router runs to deliver data directly to the outbound stream. (For small subsets, an alternative is to have N streams direct to the target.)
str_collect_runtime. As for bcast_linear, but the compiler also arranges at load-time to only deliver the data to the interface of the destination node. Other nodes simply discard the data. A better solution is to use simple streams, and rewrite the stream to deliver to the destination as necessary, but this is pending a fix to Tadpole to guarantee stream paths.
str_collect_dim†: *non-distributed destination, convex connected subset.* Arrange for streams along each dimension of the mesh, going each way. At load time, nodes who have coordinate components equal to the dest rewrite their node code to forward all incoming traffic toward the dest. The writer injects the word into the appropriate stream (rewriting the router to get it to read the interface temporarily if necessary).
str_collect_tree†. Like str_bcast_tree, but after the stream router runs the broadcast component is rewritten to throw away the data on each node. At load time, the compiler ensures that only the chosen destination is getting the data.
sch_collect_dense†: *non-distributed destination; connected subset.* Like sch_bcast_dense, but in reverse.
sch_collect_dim†: *connected subset.* Set up merging stream patterns in each necessary direction (like str_bcast_dim in spirit). The destination rewrites the passthroughs at load time to deliver the data to him.

`reduce`, `prefix`

str_reduce: *constant destination.* A reduction tree is used to send data to the destination at the root. (It might be faster to embed a tree with higher-degree nodes and non-local paths, to trade off bandwidth against computation time.)
dyn_reduce: *constant destination.* The same streams and algorithms are used, but with dynamic routing.
repl_runtime_reduce. The operator is converted to a sequential group with a constant `reduce` followed by a runtime-destination `stream` taking the result to the desired destination.
repl_allreduce. The operator is converted to a sequential group with a constant `reduce` followed by a `broadcast` taking the result to all the nodes.

sch_prefix: *connected subset*. A schedule generator is used to create an appropriate prefix. The reduction tree routine is used, forcing distance to one for all connections, and setting the maximum tree degree to three to match the restrictions in the schedule generator. A more detailed discussion of this is available in an internal NuMesh memo, [51].
str_prefix. The same tree is created as for sch_prefix, but without the restrictions on neighbor distance. The broadcast is then performed by processor forwarding, and all the streams run in the same phase.
dyn_prefix. As for str_prefix, but with dynamic routing.

`barrier`

repl_allreduce. The allreduce implementation also works for barriers. Reductions with a one-word message-size and no function are optimized to use a single address for all messages, reducing interface address pressure.
str_barrier. This entirely in-router implementation is structured like repl_allreduce, but the reduction initializes the stream endpoints to NOP. A node enters the broadcast by resetting its first child's destination to MEMADR, and each child successively enables the next child to write to MEMADR, then turns its own MEMADR access off. The last child enables the parent's upwards write before turning its access off. The parent then resets its own upward stream to NOP and waits for the broadcast to indicate the barrier is complete. This solution is good for slow processor interfaces, and uses fewer registers than repl_allreduce.
repl_barrier_bcast†. `(broadcast 'l N :m L)` is used, from some central node N, using a message length longer than the amount of buffering present on the path to the farthest node in the mesh. This may be faster than repl_barrier, but it depends on how slow it is to read words from the interface.

`cshift`, `eoshift`, `permute`

str_shift: *constant distance*. The necessary streams are created for `cshift` or `eoshift`.
dyn_shift. The data is exchanged at runtime using dynamic routing.
str_permute: *constant permutation*. Streams as for str_shift are created and used.
dyn_permute. As for dyn_shift.
sch_permute_runtime†. The values are used to build a schedule for moving the data; the relevant portion of the schedule is then downloaded to the router.
sch_shift_runtime†. Hand off to sch_permute_runtime.

`stream`, `seqread`, `seqwrite`

dyn_stream. Route the data using dynamic routing.
str_stream: *constant source and destination*. Use a stream.
str_stream_rtdest: *constant source, non-dynamic destination*. This is set up as for str_bcast, but the data is discarded on all nodes except for the desired destination. This is more efficient than putting a lot of streams in the schedule, but will be less than just choosing the right stream at runtime (if possible).
str_stream_rtsrc: *constant destination*. Similar to str_collect.

str_stream_runtime: *non-dynamic destination.* As for bcast_linear, but the data is discarded on nodes except for the desired destination.
str_stream_dyn†: *constant source.* A multicast tree to the destinations is created, then low-bandwidth reprogramming streams are added to each decision point in the tree. The source node reprograms the tree then sends the data to the desired destination.
str_stream_tree†: *non-dynamic destination.* Two stream sets are set up, one going up a reduction tree and one going down. At load time all nodes rewrite their routers to forward data across the junctures as appropriate or to read to or write from the interface.
str_seqread†. Streams are created from the last source to the previous, *etc.*, and finally from the first source to the destination. Each source writes to the stream, and then edits its router to connect the incoming source to the outgoing.
str_seqwrite†. Streams are created from the source to the first destination, then between each destination. Each node reads, then updates its router to forward data to the next node.
repl_seq_runtime†. Runtime arguments are allowed for the single source or destination of a `seqread` or `seqwrite` by replacing the operator with a sequential group that includes a constant `seqread` or `seqwrite` as well as a stream with one runtime end to move the data to or from the desired runtime node.

B.2 Generating Reduction Trees

A convenient algorithm for many implementations is some way to specify a reduction tree. The `build_tree()` function is used by some of the broadcast, collect, prefix, and reduce implementations, among others. The routine takes a subset, a Boolean array of which nodes are valid, a start index into the subset array, an upper bound on the maximum number of children per tree node, and the maximum mesh distance between a node and its parent.

The basic pattern is to start with the given start node, then walk backwards and forwards through the subset, attaching nodes to the growing tree at places that preserve its inorder (prefix-style) characteristic. The algorithm finds the suitable node in the tree that is closest to the new node, breaking ties by choosing nodes farther from the root to keep the branching factor from becoming too high.

When a Boolean valid array is given, this lists the nodes that are participating actively in the operator (see Section 5.2.4). Before returning the tree, any nodes that are invalid and all of whose descendants are invalid are pruned. Otherwise it would be necessary to specify an 'identity' value along with the specified $\otimes$ function in reductions or prefixes, so that nodes in the subset but not `:valid` could pass the data that way. This approach allows the compiler to simply forward the data opaquely on invalid nodes with valid children, and ignore altogether invalid nodes with invalid children.

The two upper bounds allow for implementations with hard-coded limits. For example, the schedule-generator prefix implementation requires all nodes to be physically adjacent to their parent and children nodes (*i.e.*, maximum distance 1), and can only handle up to three children at any node. If no such tree configuration is possible, `build_tree()` returns NULL.

Bibliography

[1] Anant Agarwal et al. The MIT Alewife machine: Architecture and performance. In *Annual International Symposium on Computer Architecture*, 1995.

[2] Kazuhiro Aoyama and Andrew A. Chien. The cost of adaptivity and virtual lanes in a wormhole router. *Journal of VLSI Design*, 2(4):315–333, May 1993.

[3] Jonathan Babb et al. The RAW benchmark suite: Computation structures for general purpose computing. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, April 1997.

[4] Jonathan Babb, Matthew Frank, and Anant Agarwal. Solving graph problems with dynamic computation structures. In *SPIE's Intl. Symp. on Voice, Video and Data Communications*, November 1996.

[5] John Beetem, Monty Denneau, and Don Weingarten. The GF11 supercomputer. In *The 12th Annual International Symposium on Computer Architecture*, pages 108–115, May 1985.

[6] Francine Berman. Experience with an automatic solution to the mapping problem. In *The Characteristics of Parallel Algorithms*, pages 307–334. The MIT Press, 1987.

[7] Ronald P. Bianchini, Jr. and John Paul Shen. Interprocessor traffic scheduling algorithm for multiple-processor networks. *IEEE Transactions on Computers*, C-36(4):396–409, April 1987.

[8] Jeffrey C. Bier, Edward A. Lee, et al. Gabriel: A design environment for DSP. *IEEE Micro*, pages 28–45, October 1990.

[9] 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.

[10] Shekhar Borkar, Robert Cohn, George Cox, et al. Supporting systolic and memory communication in iWarp. Technical Report 90-197, CMU-CS, September 1990. See also Proceedings 17th SIGARCH.

[11] Shekhar Borkar et al. iWarp: An integrated solution to high-speed parallel computing. In *Proceedings of Supercomputing '88*, November 1988.

[12] Joseph Buck, Soonhoi Ha, Edward A. Lee, and David G. Messerschmitt. Multirate signal processing in Ptolemy. In *Proceedings of ICASSP*, April 1991. Toronto, Canada.

[13] Joseph Buck, Soonhoi Ha, Edward A. Lee, and David G. Messerschmitt. Ptolemy: A platform for heterogeneous simulation and prototyping. In *Proceedings of 1991 European Simulation Conference*, June 1991. Copenhagen, Denmark.

[14] Nicholas Carriero and David Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, April 1989.

[15] Marina C. Chen. A design methodology for synthesizing parallel algorithms and architectures. *Journal of Parallel and Distributed Computing*, pages 461–491, 1986.

[16] D. E. Culler, A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumeta, and T. von Eicken. Introduction to Split-C. In *Proceedings of Supercomputing*, 1993.

[17] E. Denning Dahl. Mapping and compiled communication on the Connection Machine system. In *The Fifth Distributed Memory Computing Conference*, pages 756–766, April 1990.

[18] W. J. Dally and C. L. Seitz. Deadlock-free message routing in multiprocessor interconnection networks. *IEEE Transactions on Computers*, C-36(5):547–553, May 1987.

[19] William J. Dally. Express cubes: Improving the performance of k-ary n-cube interconnection networks. *IEEE Transactions on Computers*, 40(9):1016–1023, September 1991.

[20] William J. Dally. Virtual-channel flow control. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):194–205, March 1991.

[21] William J. Dally and Hiromichi Aoki. Deadlock-free adaptive routing in multicomputer networks using virtual channels. *IEEE Transactions on Parallel and Distributed Systems*, 4(4), April 1993.

[22] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole, Jr. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the Fifteenth Symposium on Operating System Principles*, 1995.

[23] J. Flower and A. Kolawa. Express is not just a message-passing system: Current and future directions in Express. *Parallel Computing*, 20(4):497–614, April 1994.

[24] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. *Solving Problems on Concurrent Processors*, volume 1. Prentice Hall, 1988.

[25] T. Gross, A. Hasegawa, S. Hinrichs, D. O'Hallaron, and T. Stricker. The impact of communication style on machine resource usage for the iWarp parallel processor. Technical Report 92-215, CMU-CS, November 1992.

[26] David B. Gustavson. The Scalable Coherent Interface and related standards projects. *IEEE Micro*, pages 10–22, February 1992.

[27] Soonhoi Ha and Edward Ashford Lee. Compile-time scheduling and assignment of dataflow program graphs with data-dependent iteration. *IEEE Transactions on Computers*, 40(11), November 1991.

[28] Susan Hinrichs. Simplifying connection-based communication. *IEEE Parallel and Distributed Technology*, 3(1):25–36, Spring 1995.

[29] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[30] Yuan-Shin Hwang, Raja Das, Joel Saltz, Bernard Brooks, and Milan Hodoščček. Parallelizing molecular dynamics programs for distributed memory machines: an application of the CHAOS runtime support library. Technical Report CS-TR-3374, University of Maryland, November 1994.

[31] Kirk Johnson. *High-Performance All-Software Distributed Shared Memory*. PhD thesis, MIT, November 1995.

[32] 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.

[33] Simon M. Kaplan and Gail E. Kaiser. Garp: graph abstractions for concurrent programming. In *The 2nd European Symposium on Programming*, pages 191–205, March 1988.

[34] Philip Klein, Ajit Agrawal, R. Ravi, and Satish Rao. Approximation through multicommodity flow. In *31st Symposium on the Foundations of Computer Science*, volume II, pages 726–737, October 1990.

[35] Philip Klein, Clifford Stein, and Éva Tardos. Leighton-Rao might be practical: faster approximation algorithms for concurrent flow with uniform capacities. In *22nd ACM Symposium on Theory of Computing*, pages 310–321, May 1990.

[36] R. K. Koeninger, M. Furtney, and M. Walker. A shared MPP from Cray Research. *Digital Technical Journal*, 6(2):8–21, spring 1994.

[37] A. Kolawa and S. W. Otto. Performance of the Mark II and Intel hypercubes. In *Hypercube Multiprocessors*, 1986.

[38] David A. Kranz, Robert H. Halstead, Jr., and Eric Mohr. Mul-T: a high-performance parallel Lisp. In *Conference on Programming Language Design and Implementation*, pages 81–90, June 1989.

[39] Edward Ashford Lee and Jeffery C. Bier. Architectures for statically scheduled dataflow. *Journal of Parallel and Distributed Computing*, 10:333–348, 1990.

[40] Tom Leighton, Fillia Makedon, Serge Plotkin, Clifford Stein, Éva Tardos, and Spyros Tragoudas. Fast approximation algorithms for multicommodity flow problems. In *23rd ACM Symposium on Theory of Computing*, pages 101–111, May 1991.

[41] Tom Leighton and Satish Rao. An approximate max-flow min-cut theorem for uniform multicommodity flow problems with applications to approximation algorithms. In *29th Symposium on the Foundations of Computer Science*, pages 422–431, October 1988.

[42] Virginia M. Lo, Sanjay Rajopadhye, et al. LaRCS: A language for describing parallel computations. Department of Computer and Information Science, Univesity of Oregon.

[43] Virginia M. Lo, Sanjay Rajopadhye, et al. OREGAMI: Software tools for mapping parallel computations to parallel architectures. Technical Report 89-18, Department of Computer and Information Science, University of Oregon, 1989.

[44] Virginia M. Lo, Sanjay Rajopadhye, Samik Gupta, et al. OREGAMI: tools for mapping parallel computations to parallel architectures. *International Journal of Parallel Programming*, 20(3):237–270, June 1991.

[45] Pat LoPresti. Tadpole: An off-line router for the NuMesh system. Master's thesis, MIT, February 1997. NuMesh group, MIT LCS.

[46] David B. Loveman. High Performance Fortran. *IEEE Parallel and Distributed Technology*, 1(1):25–42, February 1993.

[47] Kenneth Mackenzie, John Kubiatowicz, Anant Agarwal, and Frans Kaashoek. FUGU: Implementing translation and protection in a multiuser, multimodel multiprocessor. Technical Report LCS/TM-503, MIT, 1994.

[48] Jeff Magee, Jeff Kramer, and Morris Sloman. Constructing distributed systems in Conic. *IEEE Transactions on Software Engineering*, 15(6):663–675, June 1989.

[49] Bruce M. Maggs. A critical look at three of parallel computing's maxims. In *International Symposium on Parallel Architectures, Algorithms and Networks*, pages 1–7, 1996.

[50] Chris Metcalf. The NuMesh simulator, nsim. MIT LCS NuMesh memo 24, ftp://cag.lcs.mit.edu/pub/numesh/memos/nsim.ps.Z, January 1994.

[51] Chris Metcalf. Parallel prefix on the NuMesh. MIT LCS NuMesh memo 25, ftp://cag.lcs.mit.edu/pub/numesh/memos/parpref.ps.Z, October 1995.

[52] John Nguyen, John Pezaris, Gill Pratt, and Steve Ward. Three-dimensional network topologies. In *Proceedings of the First International Parallel Computer Routing and Communication Workshop*, May 1994.

[53] David R. O'Hallaron. The Assign parallel program generator. In *The 6th Distributed Memory Computing Conference*, pages 178–185, April 1991.

[54] Craig Peterson, James Sutton, and Paul Wiley. iWarp: A 100-MOPS, LIW microprocessor for multicomputers. *IEEE Micro*, pages 26–29, 81–87, June 1991.

[55] Ravi Ponnusamy, Yuan-Shin Hwang, Raja Das, Joel Saltz, Alok Choudhary, and Geoffrey Fox. Supporting irregular distributions in FORTRAN 90D/HPF compilers. Technical Report CS-TR-3258.1, University of Maryland, November 1994.

[56] David Shoemaker. *An Optimized Hardware Architecture and Communication Protocol for Scheduled Communication*. PhD thesis, MIT, May 1997.

[57] David Shoemaker, Frank Honoré, Pat LoPresti, Chris Metcalf, and Steve Ward. A unified system for scheduled communication. In *International Conference on Parallel and Distributed Processing Techniques and Applications*, July 1997.

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

[59] David Shoemaker, Chris Metcalf, and Steve Ward. NuMesh: A communication architecture for static routing. In *International Conference on Parallel and Distributed Processing Techniques and Applications*, November 1995.

[60] 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.

[61] B. J. Smith. Architecture and applications of the HEP multiprocessor computer system. In *Real-Time Signal Processing IV*, pages 241–248, August 1981.

[62] V. S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4):315–339, December 1990.

[63] The Supercomputing Research Center. A five year review, March 1991.

[64] Alan Sussman, Gagan Agrawal, and Joel Saltz. A manual for the multiblock PARTI runtime primitives. Technical Report CS-TR-3070.1, University of Maryland, January 1994.

[65] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(9):103–111, August 1990.

[66] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active Messages: a mechanism for integrated communication and computation. Technical Report 92/675, UCB/CSD, March 1992.

[67] Elliot Waingold et al. Baring it all to software: The Raw machine. Technical Report LCS TR-709, MIT, March 1997.

[68] S. Ward, K. Abdalla, R. Dujari, M. Fetterman, F. Honoré, R. Jenez, P. Laffont, K. Mackenzie, C. Metcalf, Milan Minsky, J. Nguyen, J. Pezaris, G. Pratt, and R. Tessier. The NuMesh: A modular, scalable communications substrate. In *Proceedings of the International Conference on Supercomputing*, July 1993.