

**Supporting Multiple Function Blocks
on a Single NuMesh Node**

by

Gregory G. Spurrier

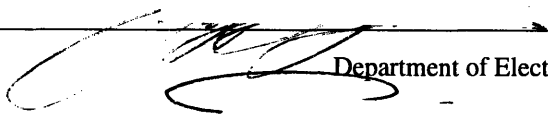
Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degrees of
Bachelor of Science in Computer Science and Engineering
and Master of Engineering in Electrical Engineering and Computer Science
at the Massachusetts Institute of Technology

May 24, 1996

Copyright 1996 Gregory G. Spurrier. All rights reserved.

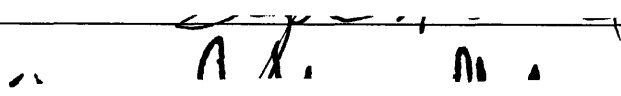
The author hereby grants to M.I.T. permission to reproduce
distribute publicly paper and electronic copies of this thesis
and to grant others the right to do so.

Author _____


Department of Electrical Engineering and Computer Science

May 24, 1996

Certified by _____


Stephen A. Ward
Thesis Supervisor

Accepted by _____


F. R. Morgenthaler

Chairman, Department Committee on Graduate Theses

MASSACHUSETTS INSTITUTE
OF TECHNOLOGY

JUN 11 1996

LIBRARIES

Eng.

Supporting Multiple Function Blocks on a Single NuMesh Node

by

Gregory G. Spurrier

Submitted to the

Department of Electrical Engineering and Computer Science

May 24, 1996

In Partial Fulfillment of the Requirements for the Degree of

Bachelor of Science in Computer Science and Engineering

and Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

The Lilypad system was developed as part of a new generation of software tools which allows programmers to specify NuMesh applications as hierarchical block diagrams rather than traditional textually specified programs. The role of Lilypad is to automatically generate control procedures for NuMesh nodes which allow an arbitrary number of primitive function blocks to reside on a single node. A set of conventions is developed for primitive function block implementations and the code generation process for the control procedures is presented.

Thesis Supervisor: Stephan A. Ward

Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

Thanks to David Shoemaker for giving me my start in the Department of Electrical Engineering and Computer Science as my 6.001 TA, and for helping me get to the end by introducing me to the NuMesh group and encouraging me along the way.

Thanks to Chris Metcalf for all his help and his patience while teaching me how to use his NuMesh simulator.

And a special thanks to Coach Greg Barringer and the Novice Lightweight Crew team for keeping me sane and fit as the academic pressures of MIT and impending thesis deadlines tried to do otherwise.

*For my Mother,
Who has always encouraged me to dream impossible dreams,
and then helped to make them come true.*

May 24, 1996

Table of Contents

1. Introduction	7
1.1. NuMesh Background	7
1.2. Developing a NuMesh Application—The Current Approach	7
1.3. High-level NuMesh Programming—A New Approach	8
2. Project Overview.....	11
2.1. Motivation.....	11
2.2. Goals.....	12
2.2.1. Efficiency	13
2.2.2. Abstract Treatment of Streams	13
2.2.3. Object Code for Primitive Function Block Instances	13
3. Primitive Function Block Implementation Conventions	14
3.1. Invocation Semantics	14
3.2. Stream Interface.....	15
3.3. Local State.....	16
3.4. Procedure Calling Conventions	17
3.5. Describing Function Blocks to Lilypad.....	17
3.6. Making Bullfrog Cooperate	19
4. Lilypad Design Decisions	19
4.1. Dealing with C Data Types	19
4.1.1. Stream Connection Type-checking.....	20
4.1.2. Stream Manipulation Code	20
4.1.3. The Decision on Types	21
4.2. Stream Representation	21
5. From Bullfrog Networks to NuMesh Code	23
5.1. Graph Building.....	23
5.2. Graph Partitioning.....	25
5.2.1. Motivation	26
5.2.2. OR-firing Blocks and Graph Partitioning.....	27
5.2.3. Demand/Tolerance Set Partitioning	28
5.3. Controller Code Generation.....	29
5.3.1. Include Files	31
5.3.2. Function Prototypes	31
5.3.3. Stream Initialization.....	31
5.3.4. State Block Initialization	32
5.3.5. Main Loop.....	33
5.4. Auxiliary Code Generation	36
5.4.1. Tadpole Input File.....	36
5.4.2. typesizegen.c.....	37
5.4.3. Makefile	38
5.5. Support for Simulation.....	38
5.5.1. Simulating an Application with nsim.....	39
5.5.2. The Need for block()	39
5.5.3. Startup Code for nsim.....	40
5.5.4. Makefile	41
5.5.5. Nsim Configuration File	41

- 6. Conclusion41
 - 6.1. Extending Lilypad42
 - 6.2. Extending the Tool Set.....43
 - 6.3. Closing Comments.....43

- 7. Appendix A -- Lilypad Definition File Format43

- 8. Appendix B – Lilypad User’s Manual.....46

1. Introduction

1.1. NuMesh Background

NuMesh [5, 6] is a communication network for heterogeneous processing elements that has been developed by members of the Computer Architecture Group of the MIT Laboratory for Computer Science. Each NuMesh node may be connected to up to four other nodes, resulting in a communications substrate for a parallel processing computer architecture.

Each node contains a programmable communication finite state machine (CFSM) and a processor. The CFSM manages the flow of information between the node and its immediate neighbors in the mesh. In addition, it also provides the interface through which the processor may exchange data with other processor elements in the mesh.

Rather than performing dynamic routing decisions as the NuMesh operates, the inter-node communication requirements of NuMesh applications are analyzed off-line to determine the communications paths which will be required at run-time. The CFSMs are then programmed to implement these communications paths and provide virtual channels of communication between the processor elements in the mesh. The result of this static analysis is that the run-time computation requirements of the CFSMs are very small, enabling them to run at a high clock speed and thus provide a high bandwidth of communication through the mesh.

1.2. Developing a NuMesh Application—The Current Approach

The NuMesh application development cycle is a tedious one that requires the programmer to perform many time consuming—and potentially error-prone—tasks manually

without the aid of a sophisticated software development tool set. For a typical application, the programmer must perform the following tasks [3]:

- Decompose the application into modules which can each reside on separate nodes in the mesh.
- Devise a mapping between the conceptual software modules and the physical nodes in the mesh.
- Determine communication routes through the mesh to link each module to any other modules with which it must communicate.
- Program each node's CFSM to implement these communication patterns (depending on the communications patterns of the application, this may require a unique program for each CFSM).
- Implement and independently test the software for each processor in the mesh (again, depending on the application, each processor may require unique software).
- Use the NuMesh simulator to debug the application as a whole.
- Debug the application on physical NuMesh hardware.

Currently, the software tools available to NuMesh application programmers are limited to the minimum required to complete the task: an assembler for CFSM code, a C compiler for the processor code, and a simulator for testing and debugging applications on a Unix workstation.

1.3. High-level NuMesh Programming—A New Approach

While the development process described above has been used to successfully develop a number of NuMesh applications, it is quite primitive and not well suited for use outside of a

research environment. In order for NuMesh to attain wide-spread use, a simpler development process, and the tools to support it, must be developed. The NuMesh group recognizes this need and is currently devoting significant research effort to this shortcoming.

The first of a new generation of software tools being developed to aid NuMesh programmers is Tadpole [2], a communications stream router that is being developed by Patrick LoPresti. Once completed, Tadpole will free programmers from the tedious and error-prone task of determining communication paths through a mesh by hand and programming the CFSMs to implement them. Instead, NuMesh programmers will specify the communication network of their application to Tadpole in terms of virtual streams which provide end-to-end communication between arbitrarily placed nodes in the mesh. Given such a specification, Tadpole will analyze the routing and bandwidth requirements of the application and automatically generate CFSM code to provide an efficient implementation of these streams.

By allowing the programmer to deal with NuMesh communications at a level of abstraction that is much more convenient than low-level CFSM code, Tadpole takes a significant first step towards relieving programmers from the tedium inherent in the current NuMesh development cycle.

In his master's thesis [1], Michael Connell presents a high level approach to developing NuMesh applications which could prove to be for processor code what Tadpole is for CFSM code. Rather than writing C code by hand for each processor in a mesh, the programmer will construct NuMesh applications by linking together function blocks with data streams, similar to the dataflow approach of programming. A set of primitive function blocks will be provided that may be combined and abstracted over to generate new function blocks. For example, a programmer will be able to derive a block which raises its input to the fourth

power from already existing blocks which compute the square of their inputs simply by linking two of them together and abstracting over the combination (see *Figure 1*).

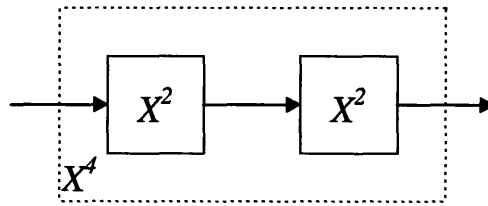


Figure 1—A composite block which computes the value of its input raised to the fourth power. The square blocks may either be primitive function blocks or composite blocks which were derived from more primitive units in the same manner.

Bullfrog is a system which was developed by Connell to reduce the hierarchical computation network descriptions generated by a NuMesh application developer to a flat network containing only primitive function blocks. The output of the Bullfrog system is a set of primitive function block instances, along with any run-time parameters that they require, and a set of streams which specify in Tadpole stream format the interconnection between these block instances.

As will be discussed in the following sections, the output from Bullfrog is not sufficient to directly drive the compilation process of the programs for the nodes in the mesh. Instead, Lilypad¹, the subject of this thesis, acts as a final stage between Bullfrog and Tadpole and the C compiler. The end result of the combination of Bullfrog, Lilypad, Tadpole and a front-end graphical user interface will be a development system (see *Figure 2*) that allows users to specify NuMesh applications as block a diagram of function units which will be compiled down to code for a physical mesh .

¹ The frog motif in naming the new generation of software development tools for NuMesh applications began as a revolt against the prevalence of acronyms in computer systems. I saw no reason to stray from this convention, and Lilypad seemed appropriate since its purpose is to provide residence space for the products of the Bullfrog system.

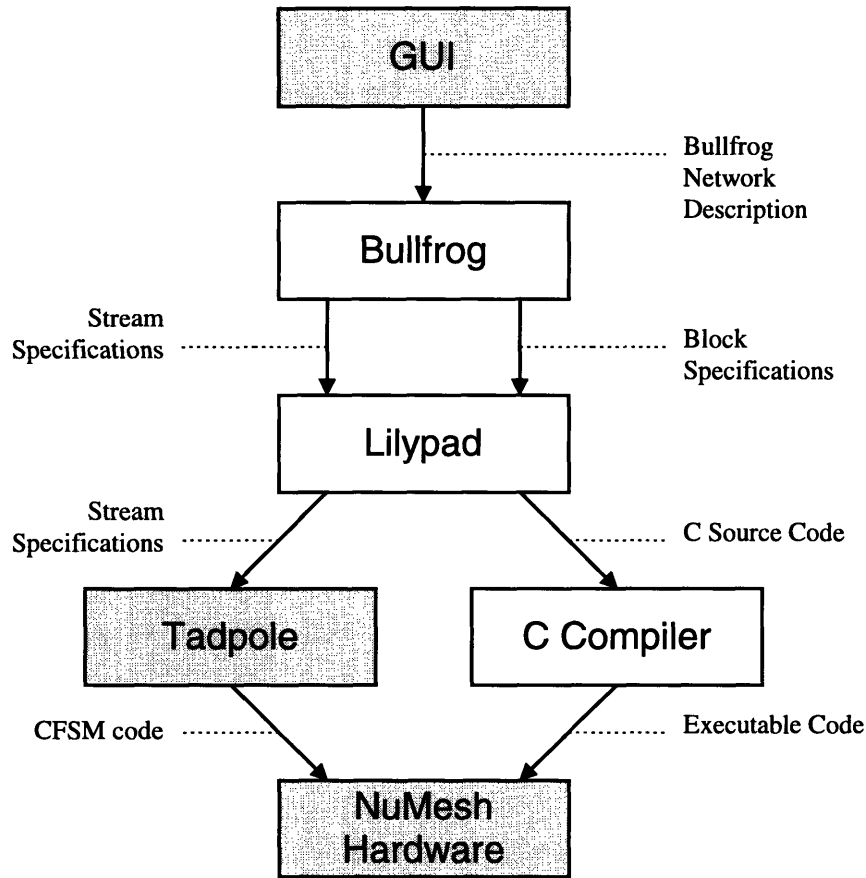


Figure 2—The compilation process which produces low-level code for the NuMesh hardware from a high-level block-diagram specification of a NuMesh application. Blocks which are shaded in gray have not yet available for use, although Tadpole and the NuMesh hardware can be simulated with *nsim*.

2. Project Overview

2.1. Motivation

The programming paradigm presented by Connell (hereafter referred to as the block-diagram approach) is an exciting one which may prove to revolutionize the way the majority of NuMesh applications are developed. Before this method may realistically be considered an alternative to the traditional NuMesh development cycle, however, two issues which stand in the way of the automatic code generation code from block diagrams must be resolved.

First, although Bullfrog performs its intended job of reducing hierarchical computation networks to networks consisting purely of primitive function blocks, its output is not suitable for the production of executable code for the processors of a mesh. This limitation comes from the fact that Bullfrog assumes a one-to-one mapping from primitive function block instances to physical nodes in the mesh. This imposes the very serious constraint that programs implemented using Connell's block diagram approach may never reduce to more primitive blocks than there are physical nodes in the mesh. Clearly, this situation must be resolved before the block-diagram approach will be accepted as a useful means of NuMesh application development.

Second, no specifications or conventions are provided to guide implementers of primitive function blocks. Without a set of constraints to which primitive function block implementations must adhere, automatic combination of these blocks into NuMesh applications would be almost impossible.

Resolving these two issues is the subject of the research behind the Lilypad system. First, a set of conventions were developed for primitive function block implementations. Next, the Lilypad system was developed to automatically generate control procedures that allow an arbitrary number of primitive blocks to reside on a single NuMesh node, thereby eliminating the one-to-one primitive to node constraint imposed by Bullfrog.

2.2. Goals

As the Lilypad research began, a number of goals were determined for the code that was to be generated by the Bullfrog/Lilypad combination. These goals served to shape both the calling conventions for primitive function blocks and the source code which is generated for their controlling procedures.

2.2.1. Efficiency

Just as a C compiler cannot be expected to produce as efficient object code as a human programmer writing in assembly language, it is unrealistic to expect the applications produced with the block-diagram approach to be as efficient as NuMesh applications which are developed by hand using the old development approach. This being said, it is still important for Lilypad to strive to minimize the inherent overhead of the block-diagram approach.

2.2.2. Abstract Treatment of Streams

A stream abstraction should be provided to primitive function block implementers and block-diagram application developers alike which shields them from the details of the stream's implementation on the physical NuMesh hardware. In particular, the programmers should not need to be concerned with (or even know) whether a stream is an inter-node stream connecting two function blocks which reside on different physical NuMesh nodes or an intra-node stream connecting two function blocks on the same node.

Additionally, the stream abstraction should free programmers from being concerned with the data word size used for transmission of data across the physical mesh. Instead, they should be able to create streams to carry data types of an arbitrary (although fixed for each stream instance) size. It is the responsibility of Lilypad to generate code to transparently pack and unpack these data types when necessary for transmission across physical node boundaries.

2.2.3. Object Code for Primitive Function Block Instances

In the interest of avoiding object code size explosion, all instances of a given primitive function block on a given node must be able to share the same object code. Although this introduces some complications for function blocks which require local state (see *Local State*, page 16), this constraint provides potentially significant savings in object code size as well as

removing the naming conflicts which could arise with multiple code instances generated from a single primitive block implementation.

3. Primitive Function Block Implementation Conventions

Before the Lilypad code generation system could be implemented to construct control procedures for the processors in a mesh, a set of conventions had to be developed to constrain the implementations of primitive function blocks. Without these conventions, Lilypad would not be able to reason about the parameters which should be passed to the block instances or about their behavior once they are invoked.

3.1. Invocation Semantics

The invocation semantics of a primitive function block specify what the state of its input and output streams must be before the block may be invoked, and what the resulting state of these streams will be afterwards. The default invocation semantics for Lilypad primitive function blocks are modeled after the dataflow architecture community's notion of a well-behaved function block [9].

Before a well-behaved function block may be invoked, all of its input streams must contain valid data (*i.e.* the AND of its input streams' valid bits must be true). Upon invocation, the block consumes the valid data element from each of its input streams and produces valid data elements for each output stream. The result is that all input streams are invalidated and all output streams are validated after function block invocation. These constraints are required by the dataflow graph partitioning algorithms that are used during the controller code generation to minimize the number of required stream validity checks (see *Graph Partitioning*, page 25).

It is conceivable that the dual of these requirements for block invocation (*i.e.* the OR of the input streams' valid bits) would be a useful additional invocation semantics for Lilypad to support. For example, if the block-diagram approach was being used to generate a simulation of combinational logic devices, the OR invocation semantics could be used to ensure that a logic unit's outputs would be updated whenever new data becomes available at its inputs rather than forcing the block to wait until all its inputs contain new data.. In the case of OR-firing primitive function blocks, Lilypad guarantees only that, upon invocation, *at least* one of the block's input streams contain valid data.

The introduction of primitive function blocks which use the OR-firing semantics for invocation reduces the effectiveness of the dataflow graph optimizations used during code generation (see *OR-firing Blocks and Graph Partitioning*, page 27), and therefore should be used with caution.

3.2. Stream Interface

It was initially proposed that primitive function block implementations be passed pointers to abstract data types representing their input and output streams. These stream pointers would then be used in combination with Lilypad-provided accessor and manipulator procedures to read from and write to the streams. For example, if the square block discussed in the introduction was implemented as a primitive block, its implementation would call a procedure to read from and invalidate its input stream, compute the square of the stream's data element, and finally call a procedure to write to and validate its output stream.

While this use of abstract data types is sound software engineering practice, the procedure calls which it requires introduce a potentially significant overhead for each primitive function block invocation. It is highly likely that the procedure calls in the square

example would outweigh the cost of the computation being performed by the block. It is therefore desirable to develop an interface to stream data which does not require procedure calls for access and manipulation, yet provides all the functionality expected from the abstract stream interface.

A careful look at the invocation semantics used by Lilypad primitive function blocks reveals that, in all cases, the valid bits of the input and output streams of a function block are treated uniformly: after block invocation, all input stream valid bits must be cleared and all output stream valid bits must be set. Because these conventions do not vary with or depend on the primitive function block implementation, they may be decoupled from the mechanism used to pass data elements into and out of the primitive function block implementations.

This decoupling of stream data elements from stream status information allows primitive function block implementations to be passed pointers to the actual data elements of the streams on which they operate. Now, the input data is available through simple pointer dereferences and output data is placed in the locations specified by the output data element pointers; all procedure calls for data element access have been eliminated.

3.3. Local State

Using a single copy of the object code for every instance of a particular function block on a processor eliminates code explosion but introduces problems for primitive function block implementations which require local state (*i.e.* utilize static local variables). When the object code is shared for these procedures, so are the references to these assumed private data areas, introducing the undesired effect of data sharing between separate instances of the block.

Local state is a necessity for some primitive function blocks (*e.g.* an FIR filter or a stream generator whose next output depends on its last), so its elimination is not an acceptable

solution. Instead, a primitive function block implementation may specify an amount of private storage which it requires and an initialization procedure for it. At run time, a private memory block will be set aside for each instance of the block and will be initialized with the specified procedure. When the block is invoked, it will be passed a pointer to the private data storage area which corresponds to the particular instance which is being invoked.

3.4. Procedure Calling Conventions

To facilitate the easy generation of code for primitive function block invocation, it is necessary to place rigid constraints on the form of the argument list for primitive block top-level procedures. The required order of arguments for Lilypad primitives is:

1. Pointers to input data elements.
2. Pointers to output data locations.
3. Run-time parameters.
4. A pointer to the instance's state block.

All of these arguments are optional, but, when present, must occur in their assigned order. Further, the input, output, and runtime parameters must occur in the same order that they appear in the block's Lilypad definition file (see *Describing Function Blocks to Lilypad*, below).

3.5. Describing Function Blocks to Lilypad

During the generation of control procedures for the processors in the mesh, Lilypad must be aware of a number of properties about each block that determine when and how each it is invoked. These include:

- The function block's top-level procedure name.

- The block's input and output streams and their types, along with the order in which they appear in the top-level procedure's argument list.
- Any run-time parameters required by the block and their types, along with the order in which they appear in the top-level procedure's argument list.
- The invocation semantics to be used for the function block.
- How many bytes, if any, need to be allocated for the block's private state area and which procedure, if any, is to be used to initialize it.

The values of these and other properties are specified in a Lilypad definition file for each primitive function block implementation. The syntax of this file is given in

Appendix A -- Lilypad Definition File Format.

3.6. Making Bullfrog Cooperate

As previously mentioned, Bullfrog assumes a one-to-one primitive function block to physical processor mapping. This creates the need for a mechanism other than Bullfrog's processor mappings to carry the processor assignments generated by the user (or, in the future, an automatic block-diagram partitioning system) through Bullfrog to Lilypad. Bullfrog's support for run-time parameters is used as a work around for this problem.

To be used with Lilypad, each primitive function block's Bullfrog function definition must include the special run-time parameter `NumMeshNode`. When Lilypad encounters this parameter in the output from Bullfrog, it treats the given value as the processor ID for the processor on which the primitive function block is to reside rather than as a normal parameter.

The source-code specification which Bullfrog requires as one of the arguments when defining a function block is used to hold the name of the Lilypad definition file which corresponds to this block. This information is used later by Lilypad as it is building its internal graph representation of the computation network.

4. Lilypad Design Decisions

The previous section described the design decisions which were made about the implementation of primitive function blocks. This section details the design decisions which were made while developing the Lilypad system and how they affect the code that it generates.

4.1. Dealing with C Data Types

A fundamental decision that had to be made during the design of Lilypad was to what extent the type information available in the Lilypad definition files should be used in the code

generation process. The possibilities for this ranged from simply ignoring the types and echoing them in the generated C source to recreating C's data type facilities internally to facilitate a much more sophisticated treatment of type information.

The final decision about this matter was made only after careful analysis of the requirements of Lilypad's two main uses for C type information: type-checking the stream connections and code generation for stream manipulation.

4.1.1. Stream Connection Type-checking

While Lilypad is constructing its internal graph of the primitive function blocks and their interconnections (see *Graph Building*, page 23) it is necessary to check that the source and destination of a stream are of the same type. For this purpose, a simple string comparison between the names of the types associated with the source and destination ports of the stream is sufficient.

4.1.2. Stream Manipulation Code

For stream allocation and data element marshaling, Lilypad must produce source code that is dependent on the size of the data elements in NuMesh words. For this, Lilypad could either rely on the compiler's `sizeof()` operator or undertake the task of determining data type representation sizes itself. Neither of these approaches is without flaw.

Use of the `sizeof()` operator requires that all decisions requiring type size knowledge be delayed until compile time. This complicates the matter of determining whether a stream that crosses a node boundary needs extra code to provide for marshaling of its data elements (see , page).

Recreating the `sizeof()` operator in Lilypad is quite problematic, however. Not only would Lilypad be required to parse C data structure descriptions and know about the

sizes of all primitive C data types, it would also need to know how the compiler packages data types for use on a given processor architecture. For example, on some compiler/processor combinations, the elements of a data structure may be padded so that the next element is always word-aligned. Building all this extra information into Lilypad could make it both compiler and processor dependent, and is therefore a very undesirable option, since NuMesh is not processor-specific.

4.1.3. The Decision on Types

Given the uses for which Lilypad needs type information, it seems reasonable to treat the type description contained within the definition files simply as strings which may be compared for equality against each other for stream type-checking purposes and may be used in conjunction with the `sizeof()` operator when the generated source code needs to know the size of a data type.

4.2. Stream Representation

The abstraction notion of a stream which can carry data elements of an arbitrary (although fixed) size is central to the block-diagram approach to development NuMesh applications. Inter-node streams will be effectively handled by Tadpole. The intra-node streams which are used for communication between two function blocks on the same processor, however, fall within the domain of Lilypad and their representation is another major design decision which had to be made as Lilypad was designed.

As an initial approximation, a stream can be thought of a pair containing a data element and a flag indicating its validity. When new data is written to a stream, the data element is updated and the flag is set to true. Likewise, when the data is read from a stream, the flag is set to false to indicate that the stream no longer contains valid data.

This description suggests representation of an intra-node stream by the following C data structure:

```
struct AMOST_STREAM
{
    DATA_TYPE data;
    int valid;
};
```

Here, DATA_TYPE is the data type of the elements which are to be carried on the particular instance of this stream.

Although this representation works fine for purely internal communication within a processor, it breaks down when called upon to interface between an internal function block and the code which is used to marshal its data elements for transmission across node boundaries. Because only a NuMesh word's worth of data may be transmitted from the node at any given time, the data structure must be augmented to keep track of what parts of the data element have been sent/received and which remain.

This is accomplished with the following updated representation for streams:

```
struct STREAM
{
    NUMESH_WORD *data;
    NUMESH_WORD *tail;
    int valid;
};
```

In this representation, NUMESH_WORD represents the data type of the unit of transmission of data between NuMesh nodes (currently this is a 32 bit word). data will point to enough NuMesh words to fully contain the data type which is being carried on the particular stream. For example, if a double-precision floating-point number is represented as a 64 bit quantity, data will be initialized to point to a block of two NuMesh words.

tail is used to keep track of how far through this block of NuMesh words data transmission has completed. This is only used by code which directly manages the packing

and unpacking of data types for transmission across physical node boundaries. When a stream is used for purely intra-node communication, this field is simply ignored.

5. From Bullfrog Networks to NuMesh Code

The following sections describe of the processes that Lilypad uses to convert the primitive function block networks that are output by Bullfrog into source code which is ready to be sent to Tadpole and a C compiler to generate CFSM and processor code for each node.

5.1. Graph Building

The first phase of operation for Lilypad is the generation of a graph for each processor which contains the primitive function blocks assigned to it along with the intra-node streams which connect them. The construction of these graphs is a relatively straightforward process of breaking down the output from Bullfrog into its individual processor components.

Bullfrog generates two files which are of use to Lilypad. The first of these to be processed contains a list of all the primitive function block instances which are present in the application, the name of the Lilypad definition file which corresponds to each of them, and any run-time parameters which they require (including the `NuMeshNode` parameter which is treated specially as a processor specifier for the block). This information is used to generate the set of block instances for each processor.

The other file produced by Bullfrog contains a set of stream descriptions in Tadpole format which specify the interconnections of the primitive function block instances described in the other file. As Lilypad processes this file and links the primitive function block instances which it previously generated, one of two possibilities will hold true for each stream: the source and destination blocks both reside on the same processor, or they are located on

different nodes. The first case is handled simply by forming an intra-node connection between the two blocks in the graph corresponding to their parent processor.

The second case, however, requires the introduction of an inter-node stream and is thus more complicated. First, the stream is added to a set of inter-node streams which is maintained for later use during the code generation phase. Next, an output node is added to the block set of the processor that contains the source block and is connected by an intra-node stream to the source block. Similarly, an input node is added to the destination processor and is connected by another intra-node stream to the destination block. These input and output nodes will be later used to generate the code required to pack and unpack the data elements carried on the stream for transmission across the physical NuMesh.

Figure 3 and Figure 4 illustrate this process on a portion of a hypothetical network, showing both the original Bullfrog and resulting Lilypad representations for the function block graph.

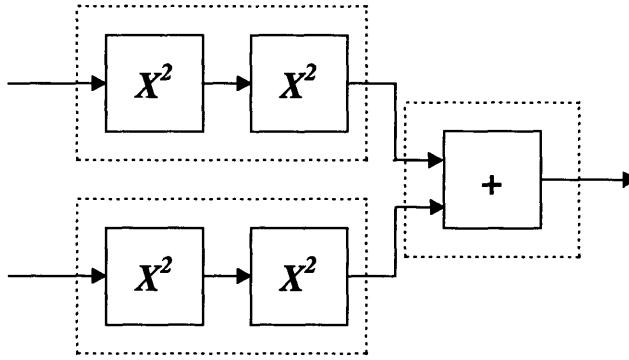


Figure 3—A portion of an example network which is generated as the output of Bullfrog. The dotted boxes indicate groupings of blocks by processor ID, as specified by the NumMeshNode parameter.

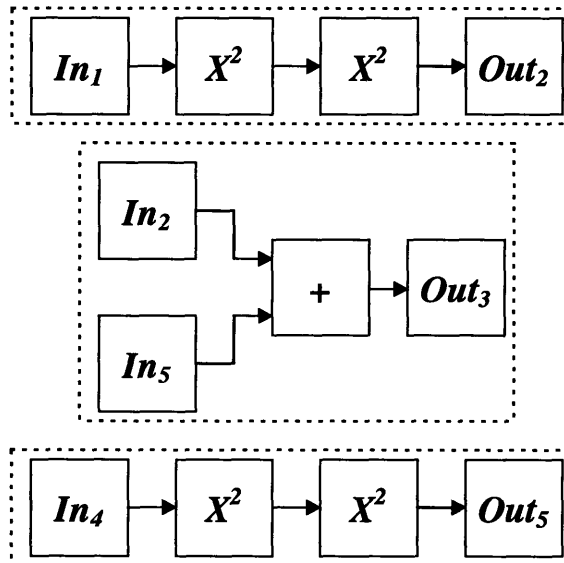


Figure 4—The same network in its internal Lilypad representation. Again, the dotted boxes indicate groups of blocks which reside on the same processor. Input and Output nodes which share the same subscript will be linked by Tadpole streams.

5.2. Graph Partitioning

After the function block graphs have been constructed for each processor, the next phase of operation for Lilypad is the optional partitioning of these graphs using algorithms borrowed from the dataflow architecture community [7, 8].

5.2.1. Motivation

The elimination of the need to pass streams as abstract data types that require procedure calls for access removed a significant amount of overhead from the code which is generated by Lilypad. A large source of the remaining overhead are the tests that the control procedure must perform for each block to determine whether it is ready to be invoked.

Algorithms have been developed as a result of dataflow architecture research to partition dataflow graphs into sets of blocks which can be proven to all be able fire once the input streams to the partition contain valid data [7, 8]. The original purpose of these algorithms was to produce the longest possible threads of execution for dataflow processors, however they can be applied equally well to the task of eliminating needless validity checks.

As a simple example of the usefulness of this application of dataflow graph partitioning, consider the code which will be generated for the top processor grouping in *Figure 4*. Using the graph as it is depicted, Lilypad would generate code that would perform the following validity checks during the course of moving one data element through the function graph:

- Before invoking the first square block instance, a test to ensure that the stream connecting *In* to the first square block is valid and that the stream connecting the two square blocks is invalid.
- Before invoking the second square block, a test to ensure that the stream connecting the first square block to the second is valid and that the stream connecting the second square block to *Out* is invalid.

A dataflow graph partitioning analysis of this graph will reveal that once data is available on the stream from the input port and the stream to the output port is ready to receive

a new data element, all of the blocks are guaranteed to be able to be invoked in order from right to left. Thus the number of stream validity checks has been reduced from four to two, producing a considerable savings.

5.2.2. OR-firing Blocks and Graph Partitioning

The graph partitioning algorithms used by Lilypad depend crucially on the well-behaved nature of the function blocks in the graph being partitioned. Since OR-firing blocks violate this property (they may be invoked before all inputs are valid), they pose a problem when used in conjunction with graph partitioning.

Fortunately, this is the same as the problem which arises in the dataflow community's research when non-strict functions appear in dataflow graphs, and the problem of how to incorporate them into the partitioning has been studied. The solution, adapted from an approach presented in [7], is to decouple the offending OR-firing block from the rest of the graph using artificially introduced input and output ports (hereafter referred to as inlets and outlets to avoid confusion with the input and output ports which interface between inter- and intra-node streams). Figure 5 shows an example of this process.

The inlets and outlets introduced are used only during the graph partitioning phase and do not surface in the final controller code. Thus, even though they appear to add overhead to the graph depicted in *Figure 5*, they do not.

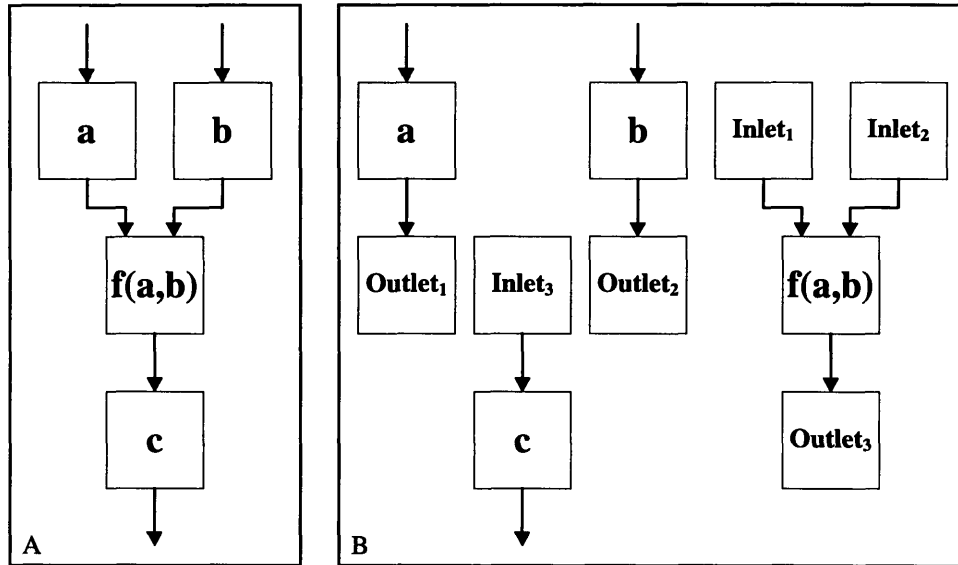


Figure 5—(A) shows a portion of a graph containing AND-firing function blocks a, b, and c and OR-firing block f(a,b). (B) shows the same graph portion after it has been transformed to allow for dataflow partitioning to be performed.

5.2.3. Demand/Tolerance Set Partitioning

Lilypad uses the simplified demand/tolerance set partitioning algorithm which is presented in [7]. A brief description of this approach is given below; for a detailed discussion including correctness arguments, please see the original reference.

The algorithm gets its name from the two sets which are calculated for each block repeatedly during the partitioning process. Informally, the demand set of a block is all of the output nodes which are affected by its output. The tolerance set of a block is given by all of the output blocks that an edge between the two can be added and not affect the input-output connectivity of the graph. That is, by adding an edge between the block and a given output, no inputs are able to reach the output that were not able to reach it before.

Formally, these sets are described by the following recursive equations:

$$\begin{aligned}
 \text{Demand}(o) &= \{o\} && [o \text{ is an output block}] \\
 \text{Demand}(b) &= \text{UNION}(\text{children}(b)) && [\text{otherwise}] \\
 \text{Tolerance}(i) &= \text{Demand}(i) && [i \text{ is an input block}] \\
 \text{Tolerance}(b) &= \text{INTERSECTION}(\text{parents}(b)) && [\text{otherwise}]
 \end{aligned}$$

To calculate the partitioning of a graph, the demand sets for the blocks are calculated and all blocks that share the same demand set are combined into a single partition. Next, the tolerance sets of the updated graph are calculated and blocks which share the same tolerance sets are combined into partitions. The process then begins again with the demand sets. This continues until the partitioning remains unchanged through the course of an iteration.

5.3. Controller Code Generation

Once the function block graphs have been assembled and partitioned for each processor, the source code for the processors' control procedures is generated. The code generation process consists primarily of filling in the details of the following control procedure template for each processor:

```

#include "typesize.h"
<INCLUDE FILES>
<FUNCTION PROTOTYPES>

struct STREAM
{
    long *data;
    long *tail;
    int valid;
};

void main(void)
{
    struct STREAM *internalStreams;
    char **stateBlocks;

    <STREAM INITIALIZATION>
    <STATE BLOCK INITIALIZATION>

    while(1)
    {
        if(<BLOCK-1 ENABLED CHECK>)
        {
            <BLOCK-1 FIRE>
        }
        :
        :
        if(<BLOCK-N ENABLED CHECK>)
        {
            <BLOCK-N FIRE>
        }
    }
}

```

The following sections describe the slots in this template and the code which is generated to fill them.

5.3.1. Include Files

Each primitive function block instance may require one or more files to be included by the C preprocessor to define the types which it uses in its top-level interface. These include files are specified as part of the Lilypad definition file for the block (see

Appendix A -- Lilypad Definition File Format, page 43). When the controller code is generated for a processor, the union of the include files required by all of the primitive function block instances residing on that processor is computed. This set is then iterated over to produce C preprocessor statements of the form:

```
#include <filename>
```

where `filename` is the name of the include file.

5.3.2. Function Prototypes

ANSI C requires that every function have a prototype, therefore one must be supplied for the top-level procedure of each type of primitive function block which is instantiated on the current processor. Rather than requiring a header file with the prototype for each block's top-level procedure, Lilypad automatically generates the necessary prototypes.

The rigid constraints which are placed on the format of the argument lists for primitive function block top-level procedures (see *Procedure Calling Conventions*, page 17) make generating the function prototypes from the information contained in a block's Lilypad definition file straightforward: the name of the top-level function is listed followed by the unique argument list which satisfies the constraints and the function block description. All primitive function block procedures are given `void` as a return type, since none return values. The function prototype for the primitive square block that has been discussed throughout this document, for example, would be:

```
void square(double *in, double *out);
```

5.3.3. Stream Initialization

Code must be generated to allocated enough space for the intra-node stream structures used by the control procedure. Once allocated, each of these streams must have memory

allocated for their data elements, and must be initialized as invalid. The following is an example of the code which is generated to initialize an intra-node stream:

```
/** Initialize stream s2$1 (double) */
(internalStreams+1)->data =
  (long *) malloc((sizeof(double)/sizeof(long)
    +(sizeof(double)%sizeof(long) ? 1 : 0)) *
    sizeof(long));
(internalStreams+1)->tail = (internalStreams+1)->data;
(internalStreams+1)->valid = 0;
```

The comment at the beginning reflects the name that was assigned to this stream by Bullfrog and the type of data elements carried on it. The complexity of the `malloc()` call is a result of the fact that the storage for data elements must be allocated in integral NuMesh word units (represented here by the type `long`). The formula within this call utilizes the `sizeof()` operator to calculate the minimum number of NuMesh words which are required to completely contain a data element of this stream's type.

After the memory allocation for the stream has been performed, the tail pointer is simply set to point to the beginning of the stream's data segment and the stream is marked as invalid.

5.3.4. State Block Initialization

The initialization process for local state areas is similar to that of the internal streams. First, memory is allocated for an array of pointers to the state blocks. Next, each of these pointers is initialized to point to a newly-allocated memory area and the state block's initialization function, if one was specified, is invoked. Below is sample code which was generated for the initialization of a state block:

```
*(stateBlocks+0) = (char *) malloc(8);
fgen_init(*(stateBlocks+0));
```

The size of the block and the initialization function are both specified in the primitive function block's Lilypad definition file.

5.3.5. Main Loop

After the streams and state blocks have been allocated and initialized, the control procedure enters its main loop that handles the routing of data through the intra-node streams and out to the external mesh. Each iteration of this loop checks each block in turn to determine whether it is able to be fired. The actual code which is generated for this test will depend on the invocation semantics which have been specified for the particular primitive function block. For AND-firing blocks:

```
(<INPUT-1>->valid && ... && <INPUT-N>->valid) &&  
(!<OUTPUT-1>->valid && ... && !<OUTPUT-M>->valid)
```

and for OR-firing blocks:

```
(<INPUT-1>->valid || ... || <INPUT-N>->valid) &&  
(!<OUTPUT-1>->valid && ... && !<OUTPUT-M>->valid)
```

Note that in both cases, no block will be fired when valid data exists on any of its output streams.

The code which is used to invoke the block and update the status of its input and output streams varies according to what type of block it is: an Input or Output block, a primitive function block, or a partition block generated by the graph partitioning phase. Different code is generated for each of these types.

5.3.5.1. Input and Output Block Invocation

Input and Output blocks are a construct of Lilypad which correspond to interface code between an intra-node stream connected to a primitive function block and a virtual NuMesh stream connected to another node in the mesh. Generating code to implement this interface is complicated by the fact that it is not known until compile time whether the data elements of a given stream will fit into a single NuMesh word. If so, no marshaling code is required and a

more efficient version of the inter/intra-node stream interface may be used. If not, data marshaling code, along with the overhead which it introduces, must be used.

The following is an example of the code which is generated for an output block:

```
#if SIZE_OF_double > SIZE_OF_long
  /*** Marshaling required because data size > NuMesh word
  ***/
  out(0, *((internalStreams+1)->tail));
  (internalStreams+1)->tail++;
  if(((char*)(internalStreams+1)->tail) -
      ((char*)(internalStreams+1)->data) >=
      sizeof(double))
  {
    (internalStreams+1)->valid = 0;
    (internalStreams+1)->tail = (internalStreams+1)->data;
  }
#else
  /*** No marshaling required ***/
  out(0, *((internalStreams+1)->data));
  (internalStreams+1)->valid = 0;
#endif
```

Note the use of preprocessor directives to select the appropriate code at compile time based on the relative sizes of the data elements and NuMesh words. Constants of the form `SIZE_OF_<type>` specify the size, in bytes, of each type that is carried on streams in this application and are defined in the `typesize.h` file which is included at the beginning of the control procedure template.

The strategy used when marshaling is required is to send one NuMesh word's worth of data at a time, from the beginning of the stream's data segment to the end. After each word is transmitted the tail pointer is updated to indicate the next word for transmission. Once the tail pointer runs off the end of the data segment, the stream is marked as invalid and the tail pointer is reset to beginning of the data segment.

The code which is generated for input blocks is similar enough to that of output blocks to not be reproduced here.

5.3.5.2. Primitive Function Block Invocation

The code generated for primitive function block invocations must first call the top-level procedure of the function block and then invalidate all its inputs and validate all its outputs. Below is an example of the code that would be produced to handle the invocation of an instance of the primitive square block:

```
/** Fire square$3 */
square((double*)((internalStreams+2)->data),
      (double*)((internalStreams+1)->data));
(internalStreams+2)->valid = 0;
(internalStreams+1)->valid = 1;
```

In this call, the data segments of the streams are type-cast as pointers to the appropriate data type for the primitive function block's input and output streams. Upon return from the computation, the input stream, `internalStream+2`, is invalidated and the output stream, `internalStream+1`, is marked as valid.

5.3.5.3. Partition Block Invocation

The invocation of a partition block is similar to that of a single primitive function block, except that multiple primitive function blocks are invoked and only the valid bits of the streams which cross partition boundaries must be tested and updated. The valid bits of the other streams can be safely ignored.

In order to guarantee correct results from the computation performed by a partition, its primitive function blocks must be topologically sorted so that a block is invoked only if the blocks on which its inputs depend have already been invoked. This ensures that blocks are never invoked with stale data.

The following is an example of code generated for the partition containing two square blocks which was discussed in the above section on graph partitioning (see *Graph*

Partitioning, page 25). For clarity, the partition invocation code is shown along with the validity checks which are performed before invocation.

```
if( (internalStreams+2)->valid &&
    !(internalStreams+1)->valid)
{
    /** Fire fsquare$4 */
    fsquare((double*)((internalStreams+2)->data),
            (double*)((internalStreams+0)->data));
    /** Fire fsquare$6 */
    fsquare((double*)((internalStreams+0)->data),
            (double*)((internalStreams+1)->data));

    (internalStreams+2)->valid=0;
    (internalStreams+1)->valid=1;
}
```

The important point to notice about this code with respect to the other types of block invocation code that were discussed above is the different treatment of stream validity bits. First, only the valid bits of the streams which cross partition boundaries (*i.e.* streams 1 and 2) must be tested before entering the invocation code for the partition. Second, the validity of the stream that connects the two square blocks is neither tested nor updated.

The net result of these differences is that two stream validity checks and two stream validity updates—one of each for the invocation of each square block—are removed. This yields a 50% reduction in both the number of tests and updates which are performed to during the invocation of these two blocks.

5.4. Auxiliary Code Generation

In addition to the control procedures for each processor, a number of other files must be generated by Lilypad before the complete code for the NuMesh can be compiled. This section describes the remaining files which are generated.

5.4.1. Tadpole Input File

Because the original Tadpole stream description file that was generate by Bullfrog assumes a one-to-one block to processor mapping, it is of no use other than specifying the

interconnections of the blocks. For the actual input to Tadpole, Lilypad must generate another file which specifies the inter-node streams which were introduced during the graph building phase.

Constructing this file is a simple matter of iterating over the processors and then over the inter-node streams. First a Tadpole node description is generated corresponding to each processor which is used in the application. Next, a Tadpole stream specification is produced which specifies the virtual ports on the source and destination nodes that the stream connects.

5.4.2. `typesizegen.c`

As discussed earlier, Lilypad relies on the C preprocessor to determine whether data marshaling code must be generated because a data type's representation will not fit in a single NuMesh word. The symbolic constants which are used in the preprocessor directives are all found in `typesize.h`.

Unfortunately, for the same reasons that it is needed in the first place, `typesize.h` cannot be directly generated by Lilypad; to do so requires type size information which can only be discovered through use of the `sizeof()` operator at compile time. The solution to this problem is to instead generate a C program which will in turn generate `typesize.h`.

The following template is used to construct `typesizegen.c`:

```
#include <stdio.h>

int main(void)
{
    FILE *outfile = fopen("typesize.h", "w");

    fprintf(outfile, "#define SIZE_OF_long %d\n",
            sizeof(long));
    fprintf(outfile, "#define SIZE_OF_<TYPE-1> %d\n",
            sizeof(<TYPE-1>));
        :
        :
    fprintf(outfile, "#define SIZE_OF_<TYPE-N> %d\n",
            sizeof(<TYPE-N>));

    fclose(outfile);
}
```

```
    return 0;
}
```

Here, TYPE-1 through TYPE-N represent the members of the set of types which are carried on streams in this application. SIZE_OF_long is always included in typesize.h because it is the data type which is used to represent a NuMesh data word.

5.4.3. Makefile

When Lilypad has finished executing, a set of C files (a control program for each processor and typesizegen.c) and a Tadpole file remain which must be compiled into low-level code for the NuMesh. This compilation process is a tedious one and it would be unreasonable to expect the user to perform these steps by hand.

For this reason, Lilypad also generates a Makefile which is used to automatically drive the compilation processes. This way, once Lilypad has finished with its code generation, all the user (or some higher-level program which itself invokes Lilypad) must do to complete the process is issue a command to start *make* with the Lilypad-generated Makefile.

5.5. Support for Simulation

Nsim [4] is a simulation and debugging package that allows programmers to test and debug their code under simulation on a Unix workstation before attempting to run it on the physical NuMesh hardware. Beyond its role in the debugging process, nsim is also valuable to the NuMesh application developer because of its support for hardware (*i.e.* the latest revision of the CFM) and software (*i.e.* Tadpole) which are not yet available for real-world use.

For these reasons it is important that Lilypad be able to target nsim as an alternative platform to the NuMesh hardware. Indeed, at this time the only way to test the code generated by Lilypad is with nsim.

The following sections describe the differences in Lilypad's behavior when it is generating code for use with `nsim`.

5.5.1. Simulating an Application with `nsim`

Rather than existing as a general-purpose stand alone program which can be used to simulate all NuMesh applications, `nsim` exists as a library which must be linked with the application's object code to produce a custom simulator for each application. The advantage of this approach is that `nsim` does not need to provide source code debugging support for the application; existing debuggers such as `gdb` instead are used on the entire application to accomplish this.

The drawback of this approach is that the object files which were destined for individual processors in the mesh must now be combined into one large application. This, at the very least, will generate name conflicts between the `main()` routines for each processor. Additionally, the application's code which was intended to run as separate processes on separate nodes are now packed into a single process on a single workstation. The remaining sections describe how these problems are dealt with.

5.5.2. The Need for `block()`

The current version of `nsim` uses non-preemptive multithreading to simulate multiple NuMesh nodes with a single workstation. Once the thread for a node's code becomes active it will remain so until it itself releases control. Control is yielded either implicitly by attempting a read or write to the mesh which will block or by explicitly calling the `block()` function.

The control programs that are generated by Lilypad are explicitly designed to never issue a read or a write which will block. This means that, because of the non-preemptive nature of `nsim`'s multithreading, once a processor control procedure has gained focus, it will

never release it. Therefore, if an application is targeted for simulation rather than use on the physical NuMesh hardware, the source code must be generated to include explicit `block()` statements to force yielding.

The `block()` statement in Lilypad generated code can also be used to determine a crude approximation of an application's performance. Each Lilypad definition file may include a specification for how many NuMesh cycles to block after its invocation. This is used to simulate the computation cost of the primitive function block. The code generated by Lilypad also includes calls to `block()` to reflect the relative cost of the data marshaling code and the checks for stream validity.

Currently, `block()` statements are inserted in the following locations when Lilypad is producing code for `nsim`:

- Before each check of the if statements which check for stream validity in the main loop.
- After each read and write to the CFSM.
- After the extra data marshaling code.
- After each primitive function block invocation.
- After the stream validity updates that occur after function block invocation.

5.5.3. Startup Code for `nsim`

`Nsim` provides its own `main()` procedure to initialize its internal state and call a procedure called `cpu()` for each processor in the mesh to start the simulation. To avoid naming conflicts with the `nsim` library, all Lilypad-generated processor `main()` routines are renamed with their processor id appended at the end. For example, the main routine for processor 37 would become `main37()`.

Lilypad also generates a file called `lpstart.c` which contains the `cpu()` procedure for this application. This procedure simply determines which processor's code it is supposed to be starting based on the processor ID which is passed to it and invokes the appropriate processor's renamed `main()` procedure.

5.5.4. Makefile

The Makefile that is generated when Lilypad is targeting an `nsim` application is updated to support the compilation process for `nsim`. Specifically, it adds `lpstart.c` to the list of source files which must be compiled and arranges for the `nsim` libraries to be linked in. After Lilypad has run to completion, the user is still able to simply invoke `make` to finish the process of creating an executable application.

5.5.5. Nsim Configuration File

`Nsim` also requires a configuration file to specify the nodes in the simulated mesh and their layout. This is generated by simply iterating over the processor set which Lilypad has been using. Because no layout information is given to Lilypad, a linear layout for the mesh has been chosen to simplify the generation of layout specifications. If the user wishes to test the application with a different topology, all that is required is to edit this file. The `nsim` documentation provides guidance on how to do this.

6. Conclusion

The goal behind the development effort for Lilypad was to build upon the work of Bullfrog to produce a system which is capable of realizing the block-diagram programming approach presented in Connell's thesis. Through the development of a set of guidelines for primitive function block implementations and the Lilypad system which allows an arbitrary

number of blocks conforming to these specifications to reside on a single NuMesh node, this goal has been realized.

The combination of Bullfrog and Lilypad has been used to successfully create NuMesh applications which run under simulation with nsim. The completion of a new revision of the CFSM and the Tadpole system are awaited so that these block-diagram applications may be tested on real hardware.

There still remain, however, a few additions which need to be made to these software tools before the block-diagram approach will be useful as a general purpose programming paradigm for the NuMesh. These are described in the sections below.

6.1. Extending Lilypad

During the initial design reviews, the need to support completely general primitive function blocks with arbitrary firing conditions and non-well-behaved semantics was discounted. Upon later reflection, however, it seems that the ability to support this generality would be useful. In particular, the constraint that blocks must produce valid output on each invocation seems to be a limiting factor when producing blocks which conditionally act on their inputs.

Extending Lilypad to handle these general blocks would require a mechanism of reporting after block execution which inputs have been consumed and which outputs have been produced. Although this conflicts with the current method of block invocation, it could be added by developing a special class of function blocks which generate this type of code, while the blocks which conform to the guidelines set in this thesis can use the already existing method of invocation.

6.2. Extending the Tool Set

Currently all of the components which were shown in the block-diagram programming system (see *Figure 2*, page 11) are either available for use, or are under active development, except for the graphical user interface. Without such an interface, application developers are required to enter Bullfrog networks by hand as text files. While this approach certainly works for generating applications, it would be much easier to use the system with a graphical tool for editing the networks.

Another tool which should be eventually added is a system which automatically determines block to processor mappings. Such a system would analyze the computation costs and communication requirements of the blocks in the application and split them up among the available processors in such a way as to maximize the performance of the mesh. Not only would this free the programmer from generating the mapping by hand, it would likely perform more sophisticated analysis than the human programmer, and thus achieve a better partitioning.

6.3. Closing Comments

The completion of Bullfrog and Lilypad marks an exciting time for the NuMesh group. Programmers are now finally able to construct NuMesh applications without writing code for each processor by hand. Once a graphical front end is available, the block-diagram approach to NuMesh application development promises to become the paradigm of choice for much of the new application development.

7. Appendix A – Lilypad Definition File Format

The Lilypad definition file is used to specify the information which Lilypad needs about a block in order to generate code to invoke and compile the block correctly. Below is a grammar specification for the Lilypad definition file format. Upper-case words represent values of the specified type, and bold-face words represent reserved tokens which the parser dispatches on.

```
definition := spec*

spec := (function STRING)
      | (inputs typed-name+)
      | (outputs typed-name+)
      | (params typed-name+)
      | (state-block NUMBER [STRING])
      | (fire-type fire-type-name)
      | (length NUMBER)
      | (files STRING+)
      | (includes STRING+)

typed-name := (STRING STRING)           ; Type followed by name
fire-type-name := and | or
```

A valid Lilypad definition file will include at most one of each specification type listed above. Other than **function**, all the specifications listed are optional and are used when they pertain to the specific primitive function block.

The (**function** STRING) construct is used to specify the name of the top-level procedure for the primitive function block's implementation. STRING must be a legal name for a C procedure.

A typed-name is a two-element list used to give information about a parameter to the primitive function block's top-level procedure. The first element is the type of the parameter; the second is the parameter's name. Typed-names are used in the declarations

of the inputs, outputs, and run-time parameters for the function block. In all cases, a parameter's name must be exactly the same as it appears in the Bullfrog definition file for the function block. This is necessary for correctly parsing Bullfrog's output.

The (**inputs** typed-name+), (**outputs** typed-name+), and (**params** typed-name+), constructs specify, respectively, the input streams, output streams, and run-time parameters to the function block. The typed-names must be listed in the same order in which they appear in the argument list for the top-level procedure.

The (**state-block** NUMBER [STRING]) specification indicates that the primitive function block requires a private state area of NUMBER bytes. If included, the STRING argument gives the name of the procedure which is invoked to initialize the state block. This procedure must be of type void and take one argument which is a char* to the block's private memory.

The (**fire-type** fire-type-name) construct specifies the invocation semantics which are to be used with this primitive function block. If a line of this form is not included in the definition file, the function block will default to AND-firing semantics (see *Invocation Semantics*, page 14).

The (**length** NUMBER) construct is used only when Lilypad is generating code for use with nsim. When this is the case, NUMBER specifies the amount of NuMesh cycles that nsim should block for after invoking this function block. This is useful for simulating a rough approximation of the relative computation cost for this block. When not specified, this parameter defaults to 3 NuMesh cycles.

The (**files** STRING+) construct is used to specify the one or more object files which must be linked with the application when using this function block. Each STRING must contain the path name of the object file relative to the Lilypad blocks directory.

Finally, the (**includes** STRING+) construct specifies any files which must be #included by the control procedure implementation. These are only required when the primitive block top-level procedure requires arguments which are of user-defined types.

8. Appendix B – Lilypad User’s Manual

Lilypad is invoked from the command line by executing `lilypad` at the command prompt. The following command line switches may be used to affect its behavior:

```
-nsim  
-outdir pathname  
-blocks filename  
-streams filename
```

The `-nsim` flag is used to specify that Lilypad should generate code targeted to run under simulation with `nsim` rather than for the physical NuMesh hardware. The specific differences in output that are entailed by using this switch are discussed in the *Support for Simulation* section, which begins on page 38.

Under normal operation, Lilypad places all of its generated files in the current working directory. If the desired location of its output is a different, the `-outdir` switch may be used in conjunction with a *pathname* to instruct Lilypad to place the files in the specified path.

As defaults, Lilypad assumes that the Bullfrog output files which specify the blocks and the streams of the bullfrog network are named with the Bullfrog defaults of `compiler.bfg` and `streams.bfg`, respectively. If the names differ, the `-blocks` and `-streams` switches may be used with the correct filenames to specify their location.

After Lilypad has successfully been run, a NuMesh application may be generated from its output by simply typing:

```
make -f Makefile.lp
```

References

- [1] Connell, Michael W. *Bullfrog: An Extensible, Modular Toolkit for the Construction of NuMesh Applications*, Master's Thesis. MIT Department of Electrical Engineering and Computer Science, February 1996.
- [2] LoPresti, Patrick J. *Tadpole: An Off-Line Router for the NuMesh System*, Master's Thesis (work in progress). MIT Department of Electrical Engineering and Computer Science, May 1990.
- [3] Metcalf, Chris. *Writing NuMesh Code*, NuMesh Systems Memo #16, March 4, 1996.
- [4] Metcalf, Chris. *The NuMesh Simulator, nsim v3*, NuMesh Systems Memo #24, December 19, 1995.
- [5] Shoemaker, David, Chris Metcalf, and Steve Ward. *NuMesh: A Communication Architecture for Static Routing*, publication? Date?
- [6] Ward, Steve, et al. *The NuMesh: A Modular, Scaleable Communications Substrate*, publication?. Date?
- [7] Coorg, Satyan R. *Partitioning Non-strict Functional Languages for Multi-threaded Code Generation*, Proceedings of Static Analysis Symposium '95, Glasgow, Scotland, UK, September 25-27, 1995.
- [8] Chery, Yonald. *Dataflow Graph Partitioning for the Id Compiler*, Masters Thesis. MIT Department of Electrical Engineering and Computer Science, May 1993.
- [9] Dennis, Jack B. *First Version of a Data Flow Procedure Language*, Proceedings of the Programming Symposium, Paris, Springer-Verlag LNCS 19, 1974.