The NuMesh: A Modular, Scalable Communications Substrate

Steve Ward,
Karim Abdalla, Rajeev Dujari, Michael Fetterman,
Frank Honoré, Ricardo Jenez, Philippe Laffont, Ken Mackenzie, Chris Metcalf,
Milan Minsky, John Nguyen, John Pezaris, Gill Pratt, Russell Tessier

MIT Laboratory for Computer Science
Cambridge, MA 02139
(617) 253-6036
numesh@cag.lcs.mit.edu

Abstract

Many standardized hardware communication interfaces offer run-time flexibility and configurability at the cost of efficiency. An alternate approach is the use of a highly-efficient, minimal communication element, with as much communication decision-making as possible done at compile time. NuMesh is a packaging and interconnect technology supporting high-bandwidth systolic communications on a 3D nearest-neighbor lattice; our goal is to combine Lego-like modularity with supercomputer performance. To date, the primary focus of the project has been the class of applications whose static communication patterns can be precompiled into independent and carefully choreographed finite state machines running on each node. Several extensions of the NuMesh to more general communication paradigms have been implemented, and the issues involved are under active exploration.

This paper presents an overview of our approach, as well as an introduction to our current-generation prototype. We also discuss our software environment and simulation technology, and enumerate some of the applications and programming models we have developed to make full use of the capabilities of the NuMesh.

Keywords: systolic array, multiprocessor, interconnect, network, reconfigurable architectures.

1 Introduction

Over the past two decades, the backplane bus has dominated computer architectures as the mechanism for intermodule communications. The reasons for this dominance are simple and remain compelling: a well-designed bus provides a simple, extensible communications substrate that allows modules performing a variety of computational tasks to be assembled into coherent systems. It induces a Tinkertoy modularity at the system configuration level, allowing system designers to construct systems without redesigning every component. However, despite the wide acceptance of buses, their technical limitations are well known and restrict their applicability in high-performance systems. Since they serialize all system-level communications, buses constitute a non-scalable communication bottleneck. Moreover, the achievable bus bandwidth is constrained both by the electrical length of the bus and by arbitration and other overheads.

The approach to this problem that we are exploring is a communications substrate that affords both modularity and high-performance communication. Our approach involves standardizing the mechanical, electrical, and logical interconnect among modules that are arranged in a three-dimensional mesh whose lowest-level communications follow largely precompiled systolic patterns. The attractiveness of this scheme derives from the separation of communication and processing components, and the standardization of the interface between them. By making the communications hardware as streamlined and minimal as possible, and requiring the compiler to do almost all the work for routing data within the mesh, we can maintain high-bandwidth, low-latency communications between the processing nodes distributed throughout the NuMesh.

Dependence on compile-time decisions regarding routing, deadlock avoidance, and resource allocation distinguishes the NuMesh communication substrate from more general dynamic networks such as those of Seitz [8, 25] or Dally [7]. One potential advantage of precompiled communication patterns is hardware simplicity, assuming run-time routing decisions can be eliminated altogether or handled infrequently with little dedicated hardware. More significant, however, is the amenability of such patterns to very high bandwidths because of their predictability. Every decision based on run-time data (such as port availability or message destination) constrains the minimum time of a flit through a network node. Elimination of such dependencies enables control circuitry to be

pipelined to arbitrary depth, allowing the data paths of the network to be switched at their maximal speed. Further, run-time collisions can be avoided by precompiling message schedules, allowing a higher level of communications traffic before saturation, as shown by Shukla and Agrawal [26].

Our goal is to develop hardware modules and supporting software that allow high-performance, special-purpose multiprocessors to be configured for particular applications by simply plugging together the appropriate NuMesh-based modules. Such modules replace the individually tuned communication paths of contemporary supercomputers by a regular mesh of replicated near-neighbor links whose mechanical and electrical characteristics are rigidly constrained and highly optimized. These constraints allow performance parameters for non-local communications that compete favorably with those of a specialized interconnect.

The interdependence of communication and computation functions in most modern parallel architectures both limits the evolution of each technology and discourages communication coherence within heterogeneous networks. We view the coupling of communication substrates with processor hardware to be roughly analogous to the processor-specific backplane buses that proliferated prior to the late 1970s, when the conceptual unbundling of communication from processing resources was evidenced by the emergence and acceptance of processor-independent buses. The NuMesh is a step in a similar revolution, providing a generic communication interface physically and logically separate from the processing elements it connects. Independent manufacturers could, for example, package and bond chips directly into an industry-standard preconstructed NuMesh package.

In the following sections we present our work: Section 2 gives our idealized system model, and our current prototype is explained in Section 3. The NuMesh software tools are discussed in Section 4, and applications and programming models are discussed in Section 5. Finally, Section 6 provides an overview of some related work.

2 System Model

The major thrust of the NuMesh project is to define a highly-efficient, generalized communication and interconnect substrate for modules of arbitrarily complex digital systems. Abstractly, a NuMesh consists of modules, or *nodes*, that may be connected together to populate a three-dimensional mesh. For example, Figure 1 shows a simplified view of a small mesh of our current prototype nodes. This figure depicts each module as a unit whose peripheral connectors provide signal and power contacts to four immediate neighbors.

Each node in the mesh constitutes a digital subsystem that communicates directly with each of its neighbors through dedicated signal lines. During each period of the globally-synchronous clock, one datum may be transferred between each pair of adjacent modules. Currently, our prototype runs at just over 1.2 Gbits/second per port; next-generation modules will be clocked faster, and we believe that our minimal hardware design will make it possible to achieve extremely high clock rates in future revisions of the hardware.

Although the NuMesh model features a rigid partition of 3-space into fixed-sized cells, it accommodates a number of variations within this discipline. A given mesh may be partially populated; one and two-dimensional configurations are specifically anticipated. Many node types may be deployed within a system, including nodes whose principal function is to provide power and cooling conduits while maintaining communication and structural integrity. Nodes may also occupy several adjacent cells; while internal communications within such nodes may follow formats foreign to NuMesh standards, NuMesh conventions are expected at their external boundaries.

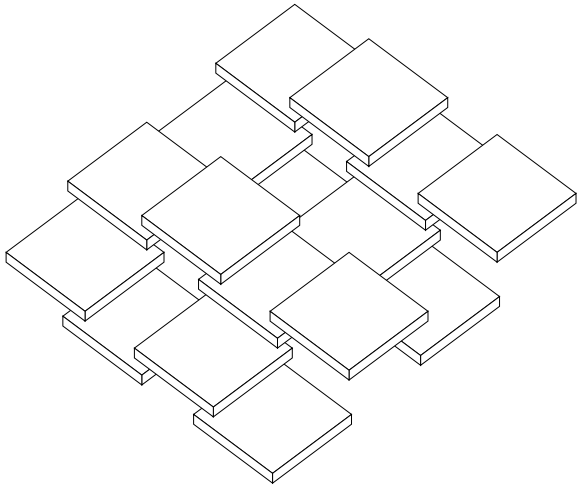

Figure 1: Simplified View of a Small NuMesh

2.1 Modules

An idealized NuMesh module is a roughly rectangular solid with edge dimension on the order of two inches. A node is logically partitioned into two parts: a *local processor* that implements the node's particular functionality, and a *communications finite state machine* (CFSM), replicated in each node, that controls low-level communications and interface functions. A node's local processor may consist of a CPU, I/O interface, memory system, or any of the other subsystems out of which traditionally-architected systems are constructed. A node's CFSM consists of a finite state machine, data paths for inter-node communication, and an *internal interface* to the local processor. A typical module is depicted schematically in Figure 2.

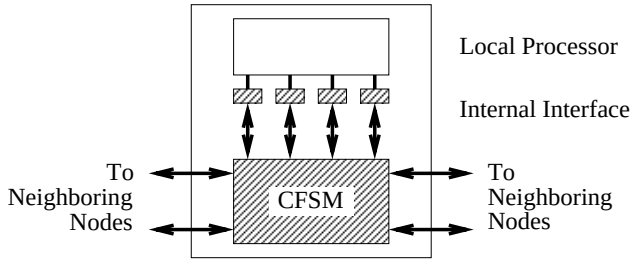

Figure 2: Abstract NuMesh Node

2.1.1 CFSM Structure

The core of the CFSM control path is a programmable finite state machine. The transition table, held in RAM, is programmed to control all aspects of routing data to other nodes and interfacing with the local processor. A small amount of additional hardware reduces the required number of states by providing special-purpose functionality such as looping counters.

The CFSM data path consists of a number of ports connected through a switching network allowing data from one port to be routed to another. Most of the ports are for communication to other CFSMs; however, one port supports CFSM-local processor transfers and allows the CFSM to move data between the processor and the mesh. This port may be wider than the network ports, and provide out-of-band signals in addition to the ordinary data path. Optimally, any combination of ports may be read or written on each clock cycle, but this flexibility may be constrained in a given implementation.

Each CFSM also contains an oscillator to generate the node's clock (the local processor has the option of an independent clock), and circuitry to control the phase of the oscillator. The clocks can be kept globally synchronized by any of a number of methods (see Section 3.1). The clock cycle time is constrained by the time necessary to transfer a data word between adjacent nodes, which can be made quite short because of the prescribed limited distances between nodes, the point-to-point nature of the links, and the use of synchronous communications.

2.1.2 CFSM Programming

The transition table of the CFSM is typically programmed to read inputs from various neighbors or the local processor into port registers and send outputs from various port registers to other neighbors or the local processor on each clock cycle. It may be thought of as a programmable pipelined switch. We consider a range of programming styles that support different communication models in Section 5.

The CFSMs in a mesh, operating synchronously at the frequency of the communication clock, follow a compiler-generated preprogrammed, systolic communication pattern. Eventually, we plan on extending our compiler technology to suggest selection, function, and mesh positions of the modules themselves. The aggregate CFSM circuitry constitutes a distributed switching network that is customized for each application; its programmability allows this customization to be highly optimized.

2.2 Statically Programmed Communication

In applications amenable to rigidly systolic communication patterns, each CFSM follows a periodic sequence of transactions with its immediate neighbors. For example, for module A to transfer a word to its neighbor B on cycle c_i of each period, A's CFSM is programmed to drive its lines to B on that cycle, while B is programmed to load data from A. By appropriate design of transition tables, arbitrary systolic communication patterns may be implemented. In some cases, words loaded by a module are destined to be read by that node's local processor. In other situations, they are routed (typically on the next cycle) to a neighboring node without local processor involvement.

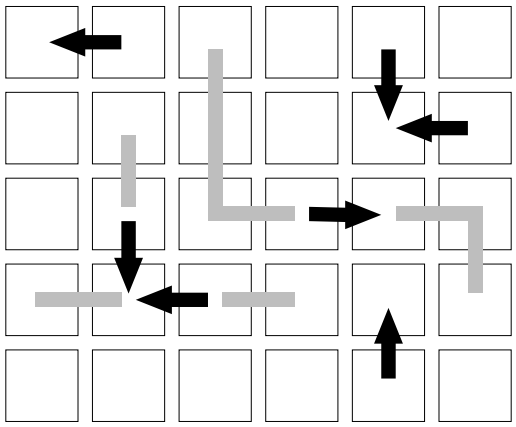

Figure 3: Communications Snapshot

Figure 3 shows a snapshot of communications within a two-dimensional NuMesh at a typical clock cycle. Heavy arrows depict local communications that complete during the current cycle; gray lines represent nonlocal transfers, formed by a sequence of local communication steps, that complete after some number of cycles. Although this pattern is static in the sense that it is compile-time determined, it is dynamic in that entirely different communication patterns may be used in each clock cycle, limited only by the amount of state memory available to each CFSM.

Certain algorithms may benefit from flow control, data-dependent CFSM execution, and other synchronization measures in their underlying communications. These may be superimposed on the data-independent primitives by software convention, allowing certain data words to contain control information. Additionally, *full–empty* status bits may be provided in each direction at each port (as is done in our current prototype); this allows a limited range of communication dynamics by indicating whether a word is available to be read or written during that cycle. The possible range of programming models can be seen as a continuum from fully static to fully dynamic; flow-control information allows us to use a more general regular-expression model to control the CFSM state.

2.3 Dynamic Communication

It is widely believed that efficient support for general models of computation (as opposed to the restricted class of systolic algorithms) requires the ability to route traffic dynamically. While we view the extent of this requirement to be an open research question,[1] we accept the value of some provision for dynamic routing if the NuMesh is to support general computation. However, we approach this need with a high-level bias toward replacing hardware functionality with compile-time analysis whenever possible.

Rather than a fully general dynamic routing mechanism capable of directing each communication to an arbitrary destination, we

[1] Intuitive counterarguments include the observation that printed circuit traces, point-to-point wiring, and backplane buses all represent rigidly static communication substrates that are commonly used to support general models of computation.

optimize for a communication mix in which destinations are largely compile-time predictable and whose unpredictable aspects (requiring run-time routing decisions) have low branching factors or occur infrequently. Our goal for the NuMesh communication architecture is to augment the static communication substrate with simple data-dependent branching that allows certain routing decisions to be based on run-time data. This will allow us to optimize applications with limited dynamic communications for the common case of packets whose path can be predicted at compile-time, using extra mechanism for dynamic routing only when necessary. A CFSM can be programmed, for example, to expect the header word for a dynamically-routed message to appear at its North port during clock number c_i; its value dictates the output port for the message, the body of which appears on subsequent clocks.

Interesting approaches to the support of such limited communication dynamics (1) compromise neither the simplicity nor the efficiency of static communications, (2) assume the number of bits of run-time decision per node to be small, and (3) avoid taking a position on issues that constitute controversial policy. Our approach is to resolve the majority of communication decisions—such as routing, arbitration, and deadlock avoidance—at compile time, and provide minimal run-time hooks to support those few cases that remain. Dynamic routing support will be provided off the critical path and only accessed explicitly by the CFSM, so that there is no slowdown for the static routing case.

Integrating dynamic routing and static routing is a difficult issue that is being studied actively by the group. A naïve way to combine the two is to allocate dynamic routing phases that are a fixed number of cycles in length, for which we can guarantee that packets injected at the beginning of the phase will reach their destination by the end. Such phases can then be intermixed as necessary with static routing. One way to implement such general dynamic communications is to have the local processor parse a header word associated with a message, returning to the CFSM the desired output port for the message (and optionally a new header word). The returned values would then be cached within the CFSM itself, allowing future messages with the same header to be routed without the intervention of the local processor. This would allow arbitrarily powerful dynamic routing models, particularly since the header words need only be recognizable by CFSMs along its path, not by every CFSM in the mesh.

Mixing static and dynamic routing in the same cycle is a harder problem. A simple initial solution is for the compiler to pre-allocate static timeslots per-link in which dynamic routing can occur. This has the obvious disadvantage of wasted bandwidth on slots that are idle on a given cycle, but more sophisticated compiler technology may be able to reclaim that bandwidth, at least partially. One simple method of producing such data-dependent routing is to pass a CFSM state pointer in the data stream, which the CFSM will be programmed to expect, and which it will use to vector to a new state that handles routing the following message. Thus a message destined for one of two possible destinations might be routed purely statically up to the node where the paths to the destinations diverge; that node would then examine the header word to determine which way to send the message.

2.4 Topology Considerations

Although a six-neighbor Cartesian mesh is intuitively appealing, other topologies may provide equivalent performance at lower cost. In the interest of post-manufacture scalability we restrict our attention to regular communication meshes whose per-node hardware requirements (*i.e.*, number of ports) are constant. One particularly interesting topology is the four-neighbor diamond lattice. Figure 1 showed a partially populated three-dimensional mesh whose connectivity is isomorphic to the structure of the diamond lattice.

The diamond interconnect has a number of useful properties, including low switch complexity, good latency and throughput performance, isotropy, and excellent mechanical and thermal properties. The number of links crossing the bisection of the diamond is less than for the equivalent Cartesian mesh; however, if we assume that nodes are pin-limited, the bisection bandwidth of the diamond is as good or better than the Cartesian mesh as a result of the 50% wider links made possible by the reduced number of ports. Further details on the diamond interconnect will be available shortly [24].

3 Prototype NuMesh

Early NuMesh prototypes with conservative performance parameters have been constructed using off-the-shelf TTL and CMOS chips. They plug together in a two-dimensional four-neighbor Cartesian mesh using standard 96-pin DIN connectors, or, with different placement of the connectors, in a three-dimensional four-neighbor diamond lattice. The nodes feature either 40 MHz SPARC processing elements or 40 MHz DSPs (Texas Instruments TMS320-C30). These prototypes are designed as an early exploration of the NuMesh communication scheme and as a development platform for prototype software. Their internal architecture compromises the goals described earlier (see Section 2.1.1) in a variety of ways; many of these compromises are lifted in next-generation prototypes currently under development.

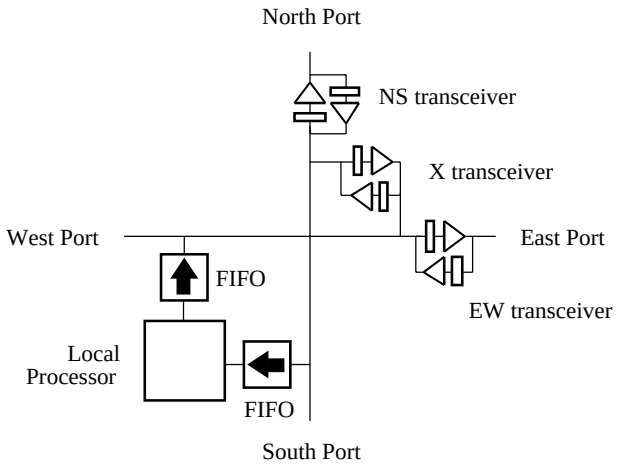

Figure 4: Prototype CFSM data paths

3.1 CFSM Architecture

Data paths for the prototype CFSMs consist of bidirectional registered transceivers that serve to isolate local buses of adjacent nodes [20]. The CFSM at each node contains North-South and East-West bus segments as shown in Figure 4, the two segments being linked by an interbus transceiver (called the X transceiver). Each node has a transceiver at the North and East ends of its respective bus segments; the opposite end of each segment connects to a similar transceiver of an adjacent node. Transitions between CFSM states are labeled with control bits that specify the actions taken by the components that make up the CFSM. The current state pointer (effectively the instruction pointer) may itself be loaded from a dedicated register on the East-West bus; additionally, the node can drive its state, augmented with its node ID and with a user-specifiable constant value, onto the North-South bus. The connection between data paths and CFSM state provides means for limited dynamic routing, as discussed in Section 2.3.

The CFSM data paths aggregate as shown in Figure5, which depicts a 5-by-3 configuration. The asymmetric placement of transceivers with respect to node boundaries allows data paths of adjacent CFSMs to be separated by a single registered transceiver and hence a single clock delay.[2] Data propagates along either orthogonal axis at a maximum rate of one node per CFSM clock cycle. A change in direction, *e.g.* routing a datum from the North port of a node to its East port, suffers an additional 1-clock latency due to the delay through the X transceiver.

Communication between a prototype CFSM and its local processor is through a pair of asynchronous FIFOs arranged so that CFSM-to-local processor data is sent from the North-South bus, and local processor-to-CFSM data is received onto the East-West bus. The attraction of FIFOs stems primarily from their inherent synchronization barrier, which allows the CFSMs and their node-specific logic to be clocked at independent frequencies. The current prototypes use processors clocked at 40 MHz and CFSMs clocked at 38 MHz. The FIFOs also make possible some simplifications in the compiler, since, for example, data does not have to be consumed on the exact cycle it is produced.

A scalable clock-synchronization technique, using only local communication between nodes, is used in the current prototypes. This synchronization technique has been proven to work for network connected as two-dimensional grids; extensive simulation has verified this result and suggests its effectiveness for grids of higher dimensions. For more information, see [23].

3.2 Host Interface and Bootstrap

A prototype NuMesh can connect either to a SPARCstation, via an SBus card connected to a prototype node's FIFO, or to a Macintosh, via a DMA NuBus card connected to a node's NuMesh port. The two interfaces support bidirectional transfers at backplane bus speeds—considerably slower than NuMesh communication bandwidths—and allow diagnosis, program loading, and application-dependent communication between the NuMesh and host.

[2]This may be thought of as a pipelined version of the data paths in the Gated Connection Network by Li and Stout as described in [18].

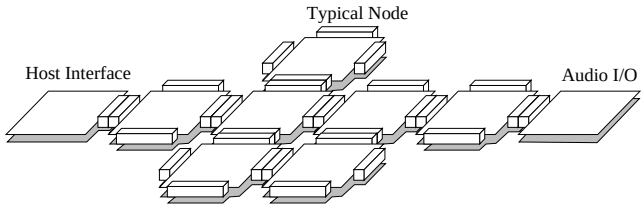

Figure 6: Typical Prototype NuMesh Configuration.

The CFSMs are designed to allow the communication substrate to be loaded and tested independently of the local processor. A mesh is initialized via a global reset, forcing each CFSM into an initial state from which it interprets incoming data from any direction as a new state transition table to be loaded. The host may explore the connected configuration by loading each successive node with CFSM code that probes its neighbors and provides a bridge connection to each responding neighbor. As a side effect of the exploration process, the host organizes the nodes into a spanning tree that governs the loading order for local processors. While it is possible to reconfigure CFSM static routing tables at run-time, applications that use this feature are still under development.

The preliminary NuMesh software interface allows the NuMesh to be used, largely transparently, as a backend processor for workstation application code. The user may naïvely launch what appears to be a conventional application, invoking an interface program that (1) reads a configuration file associated with the selected application, (2) explores the NuMesh hardware to determine whether the actual configuration is adequate to support the application, (3) loads each CFSM and local processor with code as specified in the configuration file, and (4) invokes a workstation program that communicates with both the NuMesh and the user to provide the illusion of a workstation-resident application.

4 Software Tools

The rapid prototyping and efficient deployment of ad-hoc multiprocessors depends on automating the design task: design of the network topology, allocation of computational tasks to processors, specification of timing and connectivity details for each module, and programming the local processor. While the first several steps are the most challenging, they are amenable to partial solutions—ones that involve interaction with the designer, and benefit enormously from our initial restriction to static algorithms with time- and space-bounded components.

Code generation requires an accurate, detailed model of local processor timing, optimally including processor cache latencies. However, hardware interface provisions (*e.g.*, locally shared memory with read/write synchronization flags) provide some timing latitude in processor-CFSM synchronization. Placement and routing aspects of system design—mapping a graph of time-bounded computations to a mesh of processors—can benefit from progress in adjacent domains of algorithm research.

Much of the thrust of the NuMesh project thus centers on compiler technology. Our goals include the fully automatic generation of NuMesh implementations of application-specific high-level pro-

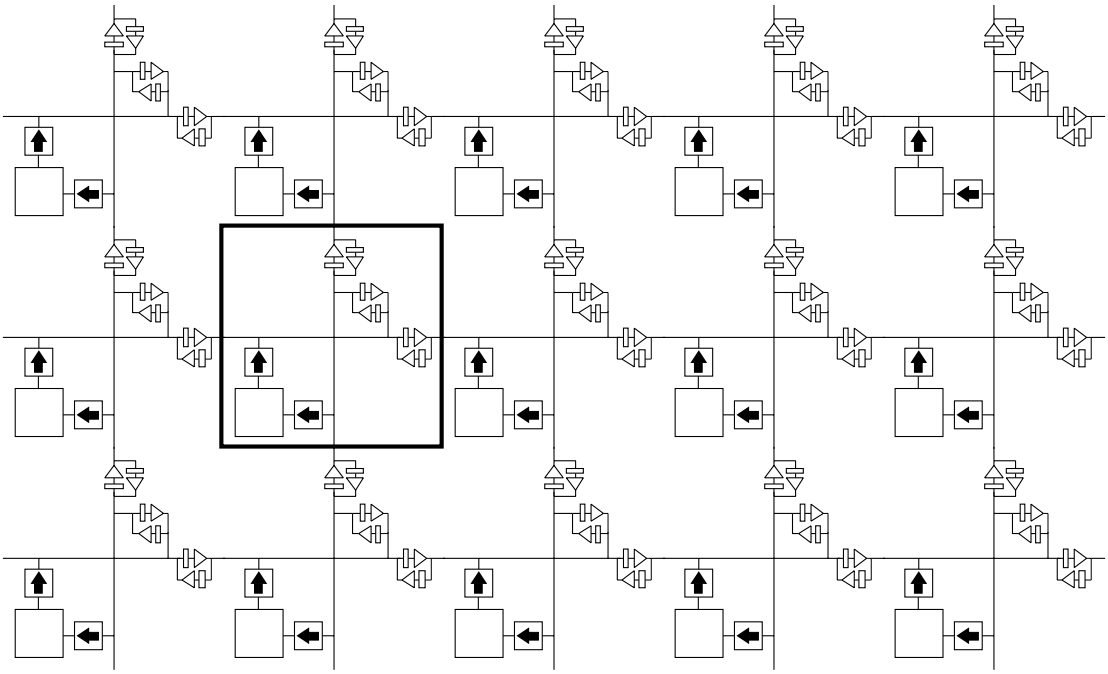

Figure 5: Aggregate prototype data paths, with one node highlighted.

gramming models that can be specified as real-time tasks; candidates include LaRCS [19], Gabriel [5, 17], and *CONSORT* [27]. For many static NuMesh applications, block diagrams, which are essentially hierarchical communication graphs, are an appropriate model; by augmenting diagrams with latency constraints the compiler can be made aware of hard real-time performance criteria that must be met by the resulting NuMesh implementation.

4.1 The NuMesh Simulator

Our general-purpose FSM-based routing simulator uses a simple register-transfer language to describe the routing actions taken in each state, and either a simple generic assembly code or compiled C code to describe the local processor programming. If compiled C code is used, it is linked with the simulator binary; this allows much faster execution for compute-bound simulations, but requires some user effort to specify delays explicitly in the time-critical sections of the code so the simulator can produce realistic results. We are currently working on adding a cycle-by-cycle SPARC simulator to the existing CFSM simulator so as to be able to generate precise simulations for our SPARC nodes.

The simulator environment allows us to experiment with different topologies, ranging from the basic Cartesian 2D and 3D meshes, to the diamond lattice configurations described in Section 2.4, to other experimental topologies. Using the simulator also allows experimentation with different CFSM architectures: for example, we can compare the code generated under the assumption that a full permutation of input data to output ports can be performed every cycle, as opposed to the current hardware where two internal buses drive the output ports. It can also be convenient to simulate much larger networks than we can currently build: for example, a multigrid [21] computation was tested on the simulator for sizes of over 16,000 nodes with 2,000 states in each CFSM; the processing code was written both in C and in generic assembler, and the CFSM code was generated by a dedicated "multigrid routing compiler."

The simulator has also proven useful when writing complex parallel code. When running native C code, the user's program, linked with the simulator, can be run under *dbx* or any other native debugger. The user can alternate between setting breakpoints or performing other C-language debugging actions and checking node states or watching data movement at the simulator's interactive debugging prompt. Furthermore, extensive task trace information is available from the simulator. For example, we can derive a parallelism profile showing when processors are active and when they are blocked waiting for I/O; Figure 7 shows one run of the multigrid code. As can be seen, the amount of parallel execution drops significantly during the coarser relaxation steps when fewer tasks are available to the nodes.

The simulator has been carefully written to maximize its speed; on a SPARCstation 2, for example, the multigrid application runs at about 53 kHz per simulated processor (in units of routing cycles). The simulator itself has also been parallelized for two separate environments. On an eight-processor Encore Multimax the same multigrid application runs at around 73 kHz per simulated processor. Additionally, System V compatible multiprocessors can take advantage of an implementation using semaphores and shared memory.

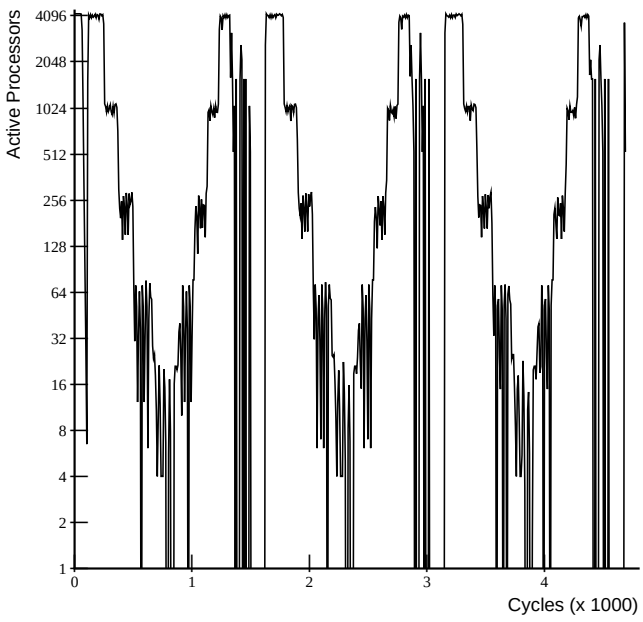

Figure 7: Multigrid parallelism profile

5 Applications

In order to develop insights into the static communication methodology, we have implemented applications using several programming models. These range from a rigid C-language interface for programs with purely static communication patterns to task-scheduling mechanisms that accommodate applications with more dynamic features. We present several different points along this spectrum.

5.1 Fully Static Routing

The most primitive programming model relies on a purely static, systolic communication scheme that uses no control-flow information. The complex scheduling issues that arise from such an approach demand either very dedicated programmers or a complete scheduling compiler such as that sketched by Lam [16]. One of our example programs uses a purely static communication scheme in implementing the multigrid algorithm for solving partial differential equations.

In the static multigrid implementation, the communication overhead is relatively low since, when possible, data is routed from source processors to their destinations before it is requested. Additionally, static NuMesh routing is very fast: to cross an 8-by-8 grid made up of our relatively slow prototype nodes takes less than 0.5 μs from source processor write to destination processor read. For all data to be routed purely statically, the code execution time must be derived by examining loop timings, and the FIFO transit time must be known precisely as well. The CFSMs can then route data exactly when it is predicted to be ready from the local processors—no time is spent waiting for flow-control information.

5.2 Flow-Controlled Static Routing

The next most dynamic model relaxes the purely static approach to allow utilization of flow-control information. In our current prototype, this model imposes a factor of two slowdown over fully static routing, since the full/empty bits are only available on the cycle following data being produced or consumed. A constrained C-language compilation environment has been developed to support a static message-passing communication paradigm. The programs written in this environment exhibit a cyclic control pattern and statically-determinable communication patterns. A scheduler generates the appropriate CFSM code from a profile of these patterns while local processor machine code is produced by a vendor-supplied C compiler. From the programmer's viewpoint, nodes run individual C programs interconnected by static streams. As a result, this scheme provides for very natural code organization and translation for certain applications. Early experiments include audio filters implemented as periodic computation modules interconnected to perform various transformations such as spectrogram display, pitch scaling, and auditory model computation, all of which are capable of using sampled audio data as inputs in addition to real-time data streams.

A similar approach is used to implement real-time video filters. These represent a set of applications with sufficiently high bandwidth requirements that they cannot be supported with most contemporary general-purpose architectures and thus illustrate the advantage of a low-overhead static communication scheme. Even with real-time video constraints dictating a sustained communication bandwidth of over 8 MWord/sec, a prototype 10-node/8-DSP configuration manages this data rate while executing such filters as vertical stripe permutation, posterization and edge-detection.

Real-time speech recognition is the goal of another large-scale NuMesh application, developed in concert with the MIT Laboratory for Computer Science's Spoken Language Systems (SLS) group [28]. We have implemented a pipelined NuMesh version of the Viterbi search used to perform lexical access [9]. The parallel decomposition of the Viterbi algorithm for the NuMesh partitions and distributes the lexical network to the processors, each of which is then able to perform Viterbi searching for its own part of the lexical network. Communication between the processors occurs in the logical pattern of a binary tree, with data being efficiently merged and broadcast by the NuMesh.

Our implementation achieves a near-linear speedup on the number of prototype nodes that have currently been built. For large dictionaries, nodes with convenient access to large memories are preferred, such as our SPARC nodes with 8 MB of cached dynamic memory. However, the other phases of the speech recognition system (spectral analysis, acoustic segmentation and phonetic classification) are best performed on DSPs. The ability to support such heterogeneity is a natural consequence of the definition of the NuMesh as purely a network substrate with a well-defined interface to the processing elements.

For vocabularies in the hundreds of words, a NuMesh network of less than a dozen nodes has currently achieved real-time lexical access for the speech recognition system. The SLS group typically runs their system on a SPARCstation 4/490 with an attached DSP card; a configuration which does not provide full real-time speech

recognition. We are in the process of evaluating the performance of an implementation of the *entire* SLS system from microphone to recognized phrases for the NuMesh, using appropriate processor types to implement the different phases of the speech recognition model.

Branch-and-bound techniques to solve search problems such as traveling salesman (TSP) are also prime candidates for flow-controlled static routing. In our implementation, we split a given TSP into an enumerable set of subproblems that can be independently solved. The strategy is to distribute these subproblems as tasks to the individual processors in a NuMesh, collecting and merging results until a definitive answer is computed. Primary communication requirements are subtask allocation among processors and distribution of the current best bound.

One approach forms a Hamiltonian path of active processors through which a monotonic current subtask number and best bound are continually forwarded via flow-controlled static routing. By embedding the dynamic communication paths in static streams of relatively high capacity, we are able to achieve good scaling performance over the modest-sized NuMesh configurations that have been constructed. A single-processor SPARCstation 2 implementation takes just over 145 seconds to run a given 14-city tour; the same tour on a single NuMesh TMS320C30 node runs in 157 seconds, and scales virtually linearly to about 14.5 seconds on the 12-node mesh. Although this approach does efficiently provide a correct solution, its reliance on a precomputed message route implies that if any processor or CFSM fails, the computation is unable to complete. Furthermore, discovering the best Hamiltonian path among the active processors requires solving a TSP to begin with—a computationally intensive precompilation step.

5.3 Diffusion Scheduling

An alternative model uses a task allocation mechanism that avoids these flaws by reallocation of subtasks between adjacent nodes based on relative sizes of task queues at each node. In an initial configuration, the entire collection of subtasks is assigned to one densely loaded node. As the computation progresses subtasks diffuse towards less dense nodes and the system tends to a steady state in which subtasks are evenly distributed among all processors. Each processor periodically sends a fixed number of messages, one to each of its physical neighbors. By controlling the contents of these messages, the processor can send individual messages to particular neighbors. This *diffusion scheduling* [11] approach eliminates the need for any precomputed communication paths.

Tolerance to processor, router, and link failures is achieved via redundancy of subtask allocation. Each node keeps a record of all subtasks it has diffused away, although at a lower priority than non-diffused subtasks. In the case of perfect hardware operation, no redundant computation is performed. When nodes do fail, the lower priority subtasks are eventually executed and the correct answer is reached, albeit with an increase in total running time. In fact, we have demonstrated on our prototype that removing a node from the mesh during execution does not affect the computation beyond an incremental slowdown.

5.4 Dynamic Routing

The most flexible routing method is the dynamic routing technique discussed in Section 2.3. The static multigrid simulation mentioned in Section 5.1 was re-implemented using dynamic routing with CFSM caching to provide a simple comparison. The simulated FSM code is reduced to one state (ROUTE and loop), and the processor is given the responsibility of forming explicit packets with a header word and (in this case) one data word. For a 9-by-9 multigrid run the dynamic routing simulation required two to four times as many cycles as the static version (depending on the assumed cache miss time), but required a maximum cache size of only 17 header words to hold all cached routes in the mesh. As the network size grows and cache requirements increase, the overhead will increase due to cache replacement.

Primitive exploration of such dynamic routing approaches using current prototype hardware involves some combination of processor interaction and dynamic reprogramming of the CFSMs. This approach was followed in the implementation of the diffusion scheduling described above.

6 Perspective

The systems approach described here deliberately reflects, and borrows heavily from, several trends increasingly evidenced in architectural research as well as commercially available computers.

Many contemporary multiprocessors rely primarily upon run time decision-making for routing general communication traffic [2, 7, 8, 10, 25], whereas systolic architectures [14, 15] and systems such as the Supercomputer Toolkit [1], use compile-time approaches to communication scheduling that offer potential cost/performance advantages in applications where communication patterns are largely predictable and macroscopically static. The NuMesh shares the bias of these latter systems toward compile-time resolution of communication choreography, but uses a homogeneous substrate of communication support to provide a simpler physical model for spatially-aware compilers as well as the potential for post-manufacture scalability.

Generalized communications support has been integrated with processors in a variety of research and commercial projects, including iWarp [6], the Transputer [13], and the GDP [3]; such provisions are beginning to appear in contemporary DSPs, such as the Texas Instruments TMS320C40. In the early 1980's, the Warp project [4] introduced the idea of a reconfigurable systolic communication framework that, like NuMesh, uses routing information obtained at compile-time to develop a communication pattern maximizing computational efficiency. Subsequently, iWarp [22], an enhanced version of Warp, provided improved compile-time capabilities and support for message-based communication using dynamic routing. It was found that although systolic communication was appropriate for many forms of scientific or business computing, some dynamic routing capabilities were needed to support general-purpose computing.

Current Transputer systems from Inmos [12] share NuMesh features such as run-time reconfigurability and high level programmability. The Transputer uses relatively slow asynchronous com-

munication links, and is targeted primarily at embedded control applications.

An extreme of reconfigurability in a processor mesh is reached by the novel Geometry-Defining Processor (GDP) [3], whose processors communicate with neighbors using bit-serial optical links, also at modest speeds. The resulting flexibility supports an approach to the analysis of physical structures using processor configurations isomorphic to the emulated object. A typical GDP structure is used as an attached processor whose host provides a compilation, pre- and post-processing environment.

7 Conclusion

A major attraction of the NuMesh approach is its early promise of high performance in the restricted but important class of applications whose communication structures are amenable to compile-time analysis. Initial work in such domains as speech, image processing, finite element analysis, and similar static, compute-intensive applications uses NuMesh for the prototyping of high-performance application-specific multiprocessor systems.

The NuMesh represents an abstraction for the specification of digital systems that relies on reprogrammable, minimal hardware rather than on hard-coding current architectural theories in hardware. Thus, we can implement these theories in software, taking advantage of technological and architectural enhancements by providing a highly reconfigurable base.

More ambitious goals of the project address generalized models of computation, exploring the boundary between run-time and compile-time decision mechanism. Reconciliation of dynamic communication patterns with this approach involves exploration of new ground, and consequently higher technical risk. While we are optimistic that incremental compilation technologies combined with minimal hardware support for critical run-time routing decisions will yield attractive engineering alternatives to current practice, the most challenging of these goals must be viewed as addressing open research questions.

8 Acknowledgment

This paper presents an overview of and motivation for early work done by members of the Computer Architecture Group at MIT. Anant Agarwal, Bill Dally, and other members of the group have been important and continuing influences on the architecture and philosophy of the NuMesh; the name itself was suggested by Dally in deference to the NuBus, a relic of the group's ancestry.

This work was supported in part by DARPA, Texas Instruments, and AT&T.

References

[1] H. Abelson, A. Berlin, J. Katzenelson, W. McAllister, G. Rozas, and G. Sussman. The Supercomputer Toolkit and its applications. In *Proc. of the Fifth Jerusalem Conference on Information Technology*, Oct. 1990.

[2] A. Agarwal, D. Chaiken, G. D'Souza, K. Johnson, D. Kranz, J. Kubiatowicz, K. Kurihara, B.-H. Lim, G. Maa, D. Nussbaum, M. Parkin, and D. Yeung. The MIT Alewife machine: A large-scale distributed-memory multiprocessor. In *Proceedings of Workshop on Scalable Shared Memory Multiprocessors*. Kluwer Academic Publishers, 1991. An extended version of this paper has been submitted for publication, and appears as MIT/LCS Memo TM-454, 1991.

[3] G. Anagnostou, D. Dewey, and A. T. Patera. Geometry-defining processors for engineering design and analysis. *The Visual Computer*, 5:304–315, 1989.

[4] M. Annaratone et al. Warp architecture and implementation. In *Proc. 13th Annual International Symposium on Computer Architecture*, pages 346–356, Los Alamitos, Calif., 1986. IEEE Computer Society Press.

[5] J. C. Bier et al. Gabriel: A design environment for DSP. *IEEE Micro*, 10(5):28–45, Oct. 1990.

[6] S. Borkar, R. Cohn, G. Cox, T. Gross, H. T. Kung, M. Lam, M. Levine, B. Moore, W. Moore, C. Peterson, J. Susman, J. Sutton, J. Urbanski, and J. Webb. Supporting systolic and memory communication in iWarp. In *Proc. 17th Annual Symposium on Computer Architecture*, Seattle, May 1990.

[7] W. J. Dally et al. The J-Machine: A fine-grain concurrent computer. In *Information Processing 89*, Elsevier, 1989.

[8] W. J. Dally and C. L. Seitz. Deadlock free message routing in multiprocessor interconnection networks. *IEEE Transactions on Computers*, C-36(5), May 1987.

[9] R. Dujari. Parallel Viterbi search algorithm for speech recognition. Master's thesis, EECS Department, MIT, Feb. 1992.

[10] D. Gustavson et al. Scalable Coherent Interface: Logical, physical, and cache coherence specifications, Jan. 1991. Preliminary draft, P1596 Working Group of the IEEE Microprocessor Standards Committee.

[11] R. Halstead and S. Ward. The MuNet: A scalable decentralized architecture for parallel computation. In *Proc. Seventh International Symposium on Computer Architecture*, La Baule, France, May 1980.

[12] Inmos Corporation. *The Transputer Databook*, 1989. Consolidated Printers, Berkeley, Calif.

[13] Inmos Ltd. *IMS T800 Architecture*, 1987. Inmos Technical Note 6.

[14] H. T. Kung. Deadlock avoidance for systolic communication. In *Proc. 15th Symposium on Computer Architecture*, Honolulu, May 1988.

[15] H. T. Kung and C. E. Leiserson. Systolic arrays for VLSI. In C. A. Mead and L. A. Conway, editors, *Introduction to VLSI Systems*. Addison-Wesley, 1980.

[16] M. Lam. *A Systolic Array Optimizing Compiler*. Kluwer Academic Publishers, Boston, 1989.

[17] E. A. Lee, W. Ho, E. Goei, J. Bier, and S. Bhattacharyya. Gabriel: A design environment for DSP. *IEEE Trans. on Acoustics, Speech, and Signal Processing*, Nov. 1989.

[18] H. Li and Q. Stout. Reconfigurable massively parallel computers: An introduction. In Li and Stout, editors, *Reconfigurable Massively Parallel Computers*. Prentice Hall, 1991.

[19] V. M. Lo, S. Rajopadhye, S. Gupta, et al. OREGAMI: tools for mapping parallel computations to parallel architectures. *International Journal of Parallel Programming*, 20(3):237–270, June 1991.

[20] K. MacKenzie. Numesh prototype hardware description. NuMesh Systems Memo 1, MIT Computer Architecture Group, June 1990.

[21] S. F. McCormick, editor. *Multigrid methods*. Marcel Dekker, New York, 1989.

[22] C. Peterson, J. Sutton, and P. Wiley. iWarp: A 100-MOPS, LIW multiprocessor for multicomputers. *IEEE Micro*, pages 26–29, 81–87, June 1991.

[23] G. Pratt and J. Nguyen. Distributed synchronous clocking. *IEEE Transactions on Parallel and Distributed Systems*. In press.

[24] G. Pratt, S. Ward, C. Metcalf, J. Nguyen, and J. Pezaris. The diamond interconnect. In process, 1993.

[25] C. L. Seitz. The Cosmic Cube. *Communications of the ACM*, 28(1), Jan. 1985.

[26] S. B. Shukla and D. P. Agrawal. Scheduling pipelined communication in distributed memory multiprocessors for real-time applications. In *18th Annual International Symposium on Computer Architecture* (ACM SIGARCH Computer Architecture News vol. 19 no. 3), 1991.

[27] S. Ward. An approach to real-time computation. In *Proc. Seventh Texas Conference on Computing Systems*, Houston, TX., Oct. 1978.

[28] V. Zue, J. Glass, D. Goodine, M. Phillips, and S. Seneff. The Summit speech recognition system: phonological modelling and lexical access. In *IEEE Intl. Conference on Acoustics, Speech and Signal Processing*, Albuquerque, NM, Apr. 1990.