PARALLEL SYSTEMS WITH FLEXIBLE TOPOLOGY

by

Akos Ledeczi

Dissertation

Submitted to the Faculty of the

Graduate School of Vanderbilt University

in partial fulfillment of the requirements

for the degree

DOCTOR OF PHILOSOPHY

in

Electrical Engineering

December, 1995

Nashville, Tennessee

Approved: Date:

_____ _____

_____ _____

_____ _____

_____ _____

_____ _____

ACKNOWLEDGEMENTS

First and foremost, I would like to express my gratitude to my advisor, Janos Sztipanovits. He has always been there whenever I needed help, guidance, or motivation. Without him, this work wouldn't have been successful.

Ben Abbott, Ted Bapty, Csaba Biegl and Gabor Karsai form the core of the Measurement and Computing Systems Group. Their contribution to my education has been invaluable.

Thanks are in order to the additional committee members, Benoit Dawant, Jerry Spinrad, and Mitch Wilkes for their helpful advices.

Thanks to Hubertus Franke, Amit Misra, Michael Moore, James "Bubba" Davis and the rest of the group for putting up with me and my silly jokes.

Many thanks to the organizations providing financial support to this research: Vanderbilt University for awarding me a University Graduate Fellowship, the US Air Force for providing funding, and the IBM T. J. Watson Research Center for its summer internship program.

Thanks to the Department of Measurement and Instrument Engineering of the Technical University of Budapest for providing me with an excellent engineering education during my six years there.

Most of all, I would like to thank my wife, Júlia, for her constant love and support. Without her, this wouldn't have been possible. I owe a great deal to my brother, Tamás, who has always pushed me into the right direction. Thanks to my parents for their support and encouragement.

TABLE OF CONTENTS

Page

LIST OF FIGURES

LIST OF TABLES

CHAPTER I

INTRODUCTION

Parallel processing offers a way to increase computing power beyond the capabilities of sequential computers. Conventional supercomputers are being replaced by massively parallel machines, such as the Thinking Machines CM-5, the Cray T3D, or the IBM SP2. At the other end of the spectrum, low cost PC plug-in boards with multiple processors have appeared on the market providing desktop parallel processing. However, parallel computing is still not widely used, primarily because, as in many other areas of computer engineering, the software technology lags behind. One of the greatest challenges is complexity management. Even small parallel programs solving relatively simple problems are inherently complex. Large-scale parallel applications with thousands of processes and hundreds of processors are hardly manageable with software engineering techniques developed for sequential processing.

In recent years several new approaches to parallel processing have been developed. Parallel programming languages hide some of the complexity from the user and let the compiler manage it. Automatic parallelization tools try to convert existing sequential programs to run on parallel machines efficiently. Message passing standards have emerged providing portability across platforms. The most recent approaches support automatic parallel program generation. One of the most promising methodologies is model-integrated computing, where the application is automatically synthesized from high-level models of the system and its environment.

The Multigraph Architecture is one such model-integrated programming environment. It has been applied successfully in diverse fields, including process monitoring and control, fault detection, isolation and recovery, and discrete manufacturing. It suits parallel processing well. The graphical, multiple-aspect, hierarchical system models manage the software and hardware complexity of the application, while the automatic system synthesizer and the run-time environment provide process synchronization and communication transparently. Embedded parallel instrumentation and signal processing is a relatively new area lacking a mature software engineering technology. Model-integrated automatic system synthesis is a promising technology that has great potential in this domain. Applying the Multigraph Architecture to embedded signal processing and instrumentation on distributed memory multiprocessors with flexible interconnection topology is in the focus of this dissertation.

Motivation

Parallel computer architectures based on such processors as the Inmos T9000 transputer, the Texas Instruments TMS320C40, or the Analog Devices ADSP21060, have the following characteristics [35]:

High performance. The individual nodes are fast, state-of-the-art microprocessors.

Scalability. The size of the network can be easily adjusted as the system requirements change. Nodes can be added one-by-one at any location in the network with an available communication link.

Flexibility. The topology of the network can be arbitrary, limited only by the maximum degree of the nodes. The interconnection architecture can be designed

specifically for the given application.

High I/O bandwidth. The multiple, high-speed communication links can be used to interface to external devices. Furthermore, input data can enter the system already distributed in the network.

Modularity. A wide variety of boards with these processors are readily available. They can be connected together as modular building blocks to form large systems. Moreover, processor-memory modules have been introduced with standard interfaces (TRAM for transputer module, and TIM for TI module). Each module contains a single or dual processor with varying amount and type of memory. Motherboards with multiple module sockets are available. This approach provides an additional degree of flexibility.

Low cost. The price of these processors is similar to that of a general purpose microprocessor with comparable performance. Since the communication capabilities are implemented on-chip, no extra hardware is required to put together a network of these processors.

These favorable characteristics make these processors ideal for embedded parallel signal processing and instrumentation applications. Systems can contain from a couple of processors up to hundreds of nodes connected by an interconnection network with flexible topology. They are able to process data on multiple channels at high data rates in real-time. The software of such systems is difficult to manage by conventional software engineering methods because of the complexity of the large-scale parallel application and the flexibility of the hardware topology.

Multiple-aspect modeling and automatic application generation is a promising new

software technology that is able to address the issues associated with these systems. Previous research efforts left some of the important problems unsolved [5]. In particular, automatic process assignment and deadlock-free message routing are two key issues that need to be addressed in the context of model-based application generation for embedded parallel signal processing and instrumentation systems.

Definitions

There are two major approaches to parallel processing. With data parallel computing (also called Single Instruction Multiple Data or SIMD), every processor executes the same program on a different input set. This approach has several favorable characteristics. Program execution is synchronous, therefore, timing is easier to predict. The approach is very efficient and scalable. The only significant drawback is that data parallel computing is only applicable to highly regular problem classes. The best examples are simple image processing algorithms. When the required transformation involves only a handful of neighbors of every pixel, a large image can be divided up to small windows. Each processor in the system executes the same transformation on one window and sends the result to the master. Unfortunately, many real-world problems are irregular and asynchronous. Hence, they cannot be solved by the SIMD approach efficiently.

Task parallel computing (also called Multiple Instruction Multiple Data or MIMD) is able to exploit parallelism in irregular problems. Every processor executes a different program using a different input set. Processors can have several processes running concurrently. Consequently, MIMD is harder to implement than SIMD. Synchronization and communication are major issues. Program execution is asynchronous, the timing and

interaction of processes are hard to predict which makes debugging very difficult. Scalability is highly application dependent. Any software technology applied in the context of MIMD needs to address these issues.

This dissertation focuses on task parallel processing. Phrases, such as parallel programming, are used in the context of MIMD throughout the dissertation.

Dissertation Outline

The presentation of the dissertation proceeds in the following manner. Chapter II surveys previous work in the three areas of parallel processing addressed by this dissertation. In particular, the representation of parallel programs, message routing, and process assignment are examined. Chapter III defines the problem domain and presents the selected solution path. In Chapter IV, the modeling paradigm for embedded parallel instrumentation systems with flexible interconnection topology is described in the context of the Multigraph Architecture, the model-integrated approach selected as the framework for this research effort. Chapter V discusses the model interpretation phase, the process of automatically synthesizing the application from the system models. In Chapter VI, a deadlock-free message routing strategy is developed for arbitrary architectures. Chapter VII discusses automatic process assignment and hardware topology synthesis. Chapter VIII describes an example application. Finally, a summary and evaluation of the contributions of this dissertation are presented and future research directions are outlined.

CHAPTER II

BACKGROUND

In this chapter, previous work in the research areas in the focus of this dissertation is surveyed. The first section examines the representation of parallel programs with special emphasis on complexity management. Next, different issues in message routing are examined including flow control and deadlock avoidance. Finally, the much studied process assignment problem is analyzed.

Representation of Parallel Programs

The conventional way of representing a sequential program is writing it in a high-level language, such as C, Ada, or Fortran. The most basic approaches to parallel programming use similar techniques: several new languages, such as Occam, language extensions, such as High Performance Fortran, libraries, such as PVM, have been proposed and used for parallel processing. These techniques make parallel processing possible. They manage some of the low-level complexity of parallel programs. For instance, a typical command in a message passing environment is "send_message". The user does not have to know how the message is packed, sent out to the communication network, transported through it, and unpacked at the destination. In a way, it is similar to a subroutine call in a high-level programming language. The user does not care about how the parameters and the return address are pushed into the stack. All he knows is that he can use the parameters in the subroutine and then return to the place of the call. The concept of subroutines is

an important step; it makes structured programming possible. But today's complex applications need much more support.

These basic approaches (languages, language extensions, libraries) are suitable for writing small to medium size programs and to develop parallel algorithms, but they are inadequate to address the issues of large-scale, complex parallel processing.

The solution proposed by many research efforts is to provide an additional abstraction layer on top of one of these approaches. These higher-level representations of the parallel application can be partially or fully automatically converted into parallel code using the constructs of the underlying low-level environment.

Textual Representation

The Linda Program Builder (LPB) provides a text-based abstraction layer on top of Linda, a language extension of either C or Fortran. Linda is based on tuples, a shared memory concept. However, Linda is available on distributed platforms as well [56]. The main concept of the LPB is the template, a skeleton Linda program. The LPB provides templates for the most frequently used parallel constructs, such as the master-worker model. New templates can be added to the system. A template included in the user program must be filled with application specific code [8]. Other constructs are also supported by the LPB.

Several high-level operations smaller than a full template can be included in the program. These can be expanded, which involves actual code generation, and then abstracted back again. This feature providing hierarchical decomposition of programs manages complexity well. The LPB maintains a program database containing the

templates, the high-level operations, the expansion level, etc. [8].

The LPB provides a higher abstraction level than a Linda program itself. The hierarchical decomposition provides visibility control. The templates and other built-in parallel constructs support reuse. These techniques manage the source code. The LPB does not deal with the application, it deals with the code. This is appropriate for small to moderate complexity systems, but large-scale parallel applications require more support.

Graphical Representation

Graphical specification of problems, solutions, designs is commonplace in all engineering disciplines. Flowcharts, signal flow graphs, state charts are examples just from software engineering. Providing a graphical abstraction layer on top of lower level parallel processing approaches is a natural idea. This technique involves not only the graphical specification of the parallel application, but also automatic transformation of the graphical constructs into actual parallel code.

Hence for PVM

The Heterogeneous Network Computing Environment (Hence) is a graphical development environment supporting Parallel Virtual Machine (PVM) application generation [12]. PVM enables a heterogeneous network of UNIX workstations to work as a single parallel machine. With PVM, the user can start tasks on the different machines which are able to communicate and synchronize with each other. Applications can be written in Fortran or C, and can include PVM library calls to send or receive messages, etc. PVM is a true heterogeneous environment. For example, it handles data conversion

between different machines transparently. It also supports fault tolerance; applications are able to survive the failure of multiple machines in the network.

Since PVM is just a library, developing large complex applications with it is difficult. Hence enables the user to specify the application as a computational graph. The nodes of the graph are the elementary computations at the subroutine level. The edges represent data dependencies and control flow. Hence also supports several control constructs, such as looping, conditional dependency, fan-out, and pipelining. They specify how different parts of the graph are to be executed.

The user sets up the configuration of available machines. A cost matrix can also be provided specifying the estimated cost of executing different tasks on different machines. Hence automatically generates the executable programs using PVM calls, installs them on the different machines, and executes them. The program automatically assigns the tasks to the machines based on the cost matrix at runtime [12].

By providing a graphical user interface to PVM, Hence helps application development. However, it does not support hierarchical decomposition or any other technique to manage complex applications.

Express

The Express toolset provides an environment to develop efficient and portable programs for distributed computer platforms. Some of the key features of Express are [56]: hardware independent communication system, high-level communication library, static and dynamic load balancing, and heterogeneous system capability.

One target hardware platform of Express is transputer networks. The Cnftool program

aids in the configuration of the system. After running a worm program, the Cnftool displays the network in a graphical format. It can be modified (or created from scratch without running the worm) by adding and connecting nodes with the built-in graphical editor. Message routes are automatically generated and can be displayed as well. Note that deadlock-freedom is only guaranteed for hypercubes, meshes, and trees. The program also generates configuration information for the network loader [3].

Despite the lack of a high-level application specification layer, the Express toolset is quite popular. The Cnftool is a big step forward from the traditional manual configuration of reconfigurable networks by editing textual description files. Large processor networks, however, are hard to manage with the program. Hierarchical decomposition would help in this respect. Furthermore, Cnftool does not provide support to verify the topology of the network.

The Multigraph Architecture

The Multigraph Architecture (MGA) provides a unified software architecture and tools for: (1) building, testing, and storing multi-aspect, graphical domain models, and (2) transforming the models into executable programs and/or extracting information for system engineering tools [33]. The MGA has the following functional components (Figure 1):

> **Graphical Model Builder (GMB)**. The modeling paradigm supported by the GMB includes concepts, relationships, model composition principles, constraints, and representation techniques that are accepted and used in the application domain. The GMB tool provides a customizable model building environment for

domain experts. It enforces domain-specific constraints during model building, uses domain-specific graphical formalism, and supports checking the models against consistency and completeness criteria.

Model Database. The model database stores the complex, multiple-aspect models. Typically, off-the-shelf or public domain object oriented databases are used for this purpose.

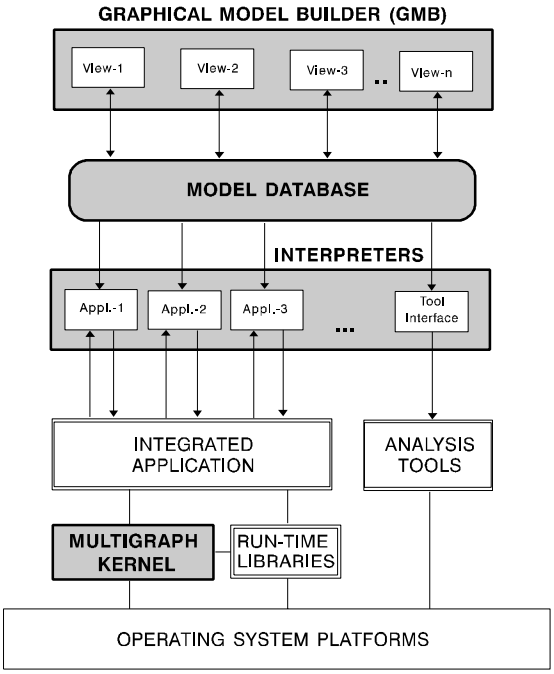

Figure 1 The Multigraph Architecture

Model Interpreters. Model interpreters synthesize executable programs from domain models and generate data structures for system engineering tools that perform various analyses of the systems to be built. Since the model interpreters

capture the relationship between the problem space and the solution space, they are specific to the domain.

Multigraph Kernel. The executable programs are composed in terms of the Multigraph Computational Model (MCM). The MCM is a macro-dataflow model providing a unified system integration layer above heterogeneous computing environments, including open system platforms, parallel/distributed computers, and signal processors. The run-time support of the MCM is the Multigraph Kernel (MGK). The MGK provides scheduling, synchronization, and communication. The elementary computations are carefully defined reusable code components that are part of application specific run-time libraries. The model interpreters synthesize the applications by building the dataflow graph and setting the parameters of the elementary computation blocks.

The Graphical Model Builder (GMB) is a configurable, visual model building environment. The general model organization principles provided by the GMB are [33]:

Connectivity. The GMB supports a graph-based modeling approach. All models are, in essence, nothing else but nodes with ports and connections between them.

Attributes. Graphical representation cannot express every aspect of a system. Therefore, model components can have numerical and textual attributes. These may include differential equations for process models, data types for signals, pieces of code for computational objects, etc.

Multiple aspects. Systems can be modeled from different aspects. For example, a plant can have a process model representing material, energy, and information transfer processes, and an equipment model describing the actual hardware of the

plant [33]. The GMB allows as many modeling aspects of the same system as required by the domain.

Hierarchy. Complex systems tend to have complex models. To help manage the complexity of the models themselves, the GMB supports hierarchical model building. Hierarchical models can represent the system with varying degrees of detail.

Multiple views. At any one level of the hierarchy, the complexity of the models can still be overwhelming. Many times the models can be partitioned into different views. For example, a model of a plant can have one view for electrical signals, and another for hydraulic signals. Interactions between different views is modeled by components appearing in multiple views. The GMB enforces the consistency of the views.

References. Interactions between different aspects can be modeled by reference objects linking components from two (or more) aspects together.

The configuration of the generic GMB for a new domain requires the specification of the domain concepts (e.g. predefined objects, how models are organized, what connections are allowed, etc.). A Model Definition File (MDF) must be created containing these specifications written in a declarative language designed for this purpose. The built-in parser of GMB reads this file and configures itself accordingly. Another program generates the model database interface codes.

The Multigraph Computational Model (MCM) is a macro-dataflow model [13,6]. The computations in the MCM are described as a directed, bipartite graph. The main components of the MCM are:

Actornodes. They are the computational operators of the dataflow graph. They have several components:

State. An actornode can be disabled, waiting, ready, or running.

Script. The script is a (preferably) reentrant routine written in a procedural or functional language that performs the necessary computations on the data received by the node and propagates the results.

Context. The context is a static local memory available to the actor. It is used to provide parameters for the actor by the builder and/or to save the state of the actor between successive firings.

Ports. Input and output ports provide interfaces for the actornodes to the datanodes. The number of ports available to an actor is decided at build time.

Control principle. The control principle determines when an actornode is ready for execution. An "ifall" actor can be fired only after data is available at all of its input ports. An "ifany" actor is ready whenever there is data at any one of its input ports. An adder actor, for example, must be "ifall", while a merger actor must be "ifany".

Datanodes. They provide queuing and connection functions between actornodes. Actors can only be connected to datanodes and vice versa. Any number of actor output ports can be connected to a datanode. A datanode can be connected to any number of actor input ports. Datanodes provide a straightforward interface to build, control, and monitor the dataflow graph.

Environments. They are used to protect system resources and to prioritize the graph. Every actor is assigned to an environment. Only one actor per environment

is executed at any one time. Actors are scheduled based on the priority of their environment.

Tasks. Tasks provide a generic interface to the basic computational resources of the underlying hardware. In a multitasking environment, they are simply the different processing threads available to the Multigraph Kernel. In a multiprocessor system, tasks are the processors themselves. Environments are attached to tasks.

Abbott utilizes the Multigraph Architecture for the automatic synthesis of parallel software systems [5]. He specifically targets complex signal processing and instrumentation applications running on parallel architectures with flexible interconnection topology. One such application is the Computer Assisted Dynamic Data Monitoring and Analysis System (CADDMAS) used for turbine engine testing at Arnold Engineering Development Center [10].

He uses the Multigraph Architecture comprised of an earlier set of tools, including a Lisp-based Graphical Model Builder and Model Database. They do not provide the flexibility and richness of the new environment. Model building and model interpretation are much slower.

Abbott models the parallel systems from two aspects. The signal flow aspect describes the building blocks and the structure of the signal flow of the application. Since every application can have a different hardware configuration, the available computing resources and their interconnection topology is modeled as well.

Bapty utilizes the Multigraph Architecture for the automatic synthesis of parallel real-time systems [11]. He proposes a multi-phase modeling and model interpretation approach

to adapt the environment to the different classes of users involved in the different steps of the system development process. Real-time constraints are explicitly modeled. The model interpreters automatically synthesize the application providing automatic process assignment and static scheduling that guarantees real-time performance.

The Starburst Environment

The Starburst parallel programming environment is implemented on top of the Processing Graph Method (PGM). The PGM is a dataflow computational model targeted at signal processing [31]. A PGM application consists of a processing graph describing the signal flow of the system and a command program managing the graph execution. The main components of processing graphs are [31]:

Nodes. Nodes can represent one computation element, such as an FFT routine, or whole subgraphs creating a hierarchical structure. Nodes have the following parts:

Ports. Each node must have at least one input port and may have any number of output ports. A set of Node Execution Parameters is associated with each input port. When the number of data elements available for each of the node inputs are greater than or equal to the corresponding *threshold amounts*, the node is ready for execution. At each firing, the node skips a number of input data elements specified by the *offset amount*, reads a number of data elements determined by the *read amount*, and removes a number of data elements specified by the *consume amount*.

Primitives. The primitive specifies the computation that the node executes.

Queues. The directed edges of processing graphs represent queues. Queues are

strongly typed FIFO data structures. At most one node output port can be connected to a queue input. The queue output can be connected to at most one node input port.

Graph variables. A graph variable is a global variable accessible by multiple nodes. Its value can be modified by any node or the command program.

Command programs run on the host system. They are used to start and stop the graph execution, enter parameters, flush the data from queues, etc. Processing graphs can be specified by the Signal Processing Graph Notation language.

Starburst is a graphical development environment for parallel signal processing applications. Its target hardware platform is the Intel i860-based Mercury RACE shared memory multiprocessor. The Application Designer program is used to specify the signal flow graph of the system. Nodes can represent a single computation unit, such as a filter or an adder, or a subgraph. Nodes have input and output ports connected by streams. Static parameters can also be assigned to nodes. The depth of a node is an integer specifying the number of instances of the node. This is a technique to simplify the specification of repetitive structures. The inside and outside connections of these node instances must be exactly the same [4].

With Starburst, process assignment must be done manually. The program automatically builds the application using the integrated Processing Graph Method and executes it on the multiprocessor. It also provides debugging support on the host computer [4].

Starburst has some common characteristics with the Multigraph Architecture. They both automatically generate executable code based on a dataflow computation model from

high-level graphical specifications. The scope of Starburst is quite narrow, it targets signal processing applications on one specific architecture.

Evaluation

New languages, language extensions, and libraries make parallel processing possible by providing basic constructs for communication and synchronization transparently to the user. To manage the complexity of large-scale applications, however, a higher level abstraction is needed. Most textual representation techniques provide support to write, handle, and look at the source code more effectively. The problem is described by the source code, not in terms of the application domain. A radically different approach is needed to manage the complexity of large-scale parallel programs.

A lot of information specifying a complex parallel application is inherently topological. Graphical representation, therefore, is a natural way of describing such systems. Simple approaches to specify the application by a computation graph, where nodes represent processing at the subroutine level and edges correspond to data and control flow and data dependency, provide a clean abstraction layer. However, the computation graph of an application quickly becomes unmanageable as the system size increases. Additional techniques are needed for complexity management. Hierarchical decomposition aids in this respect. A node of a hierarchical computation graph may hide a subgraph which may itself be hierarchical. The graph can be examined at the currently required level of detail.

While hierarchy helps, it alone is not able to handle the complexity of large-scale applications. As Harel points out, multi-aspect modeling, coupled with hierarchical

decomposition and automatic application generation, offers an excellent solution to complex system development [29]. The Starburst programming environment and other approaches utilize some of these elements. The Multigraph Architecture is customizable to radically different application domains and incorporates a wide variety of techniques to manage the complexity of the application and the system models themselves. As previous experience shows [5, 6, 11], the Multigraph Architecture is particularly suitable for large-scale parallel processing. On the other hand, some important issues have not yet been addressed satisfactorily. How the requirements of automatic process assignment and deadlock-free message routing affects the modeling paradigm needs to be studied.

Message Routing

In distributed memory systems, processors communicate with each other by sending messages. Interconnection networks consist of nodes with a limited number of communication channels. A fully interconnected topology, therefore, is not possible for larger systems. Several regular topologies have been proposed in the literature, including k-ary n-cubes (e.g. hypercube, toroid) and n-dimensional meshes (e.g. 2D and 3D meshes) [46]. A processor can send messages directly to its neighbors using their common communication channel. Messages to processors further away in the network have to pass through intermediate nodes. Message routing is the process of selecting the path to take through the network and transporting the message from the source to the destination accordingly.

If the message path is always one of the shortest paths for all source destination node pairs, the routing is *minimal*, otherwise it is non-minimal. With minimal routing, as a

message moves through the network, its distance to the destination is always decreasing, i.e. it is always one step closer to the destination after every hop.

If the routing supports sending messages between any two nodes, it is said to be *connected*. If only selected sets of processors need to communicate with each other in the given application, the message paths can be optimized by not considering the unneeded paths. Such message routing is called *partially connected*.

There are two basic interconnection strategies for multicomputers: *direct and indirect networks*. There are two types of nodes in indirect networks: processing and switching (or routing) nodes. Messages between processors are routed indirectly through the switching nodes. Direct networks, on the other hand, consist of one type of node. All switching is performed in the processing nodes [20].

Route Selection

There are two different ways of specifying the route a message must take in the network. With *source routing*, the whole route is decided at the source node and the routing information is included in the message header. Each intermediate node must read this header and decide whether to keep the message or forward it through the specified communication link. The routing information included in the message is an overhead which must minimized. A good technique is called street-sign routing because of its similarity to directions given to a driver in a city. There is a default forwarding channel at every node for every incoming link. The header only contains information for a node if a different channel must be taken, i.e. the message must "turn" [42].

In *distributed routing*, a routing decision is made at every node. The simplest method

uses routing tables. One such table is stored at every node containing forwarding information for every possible destination node. Routing here is a simple table lookup using the destination node address included in the message header as an index. When the routing is implemented in hardware by a routing chip, the available memory for the routing table limits the size of the network. Therefore, the table size must be reduced. One good method is assigning addresses to the nodes in such a way that the same outgoing channel corresponds to a whole range of addresses [42]. This technique is called *interval labeling*. Instead of table lookup, more complicated algorithms can be applied in distributed routing. The algorithm must be fast and easily implementable in hardware.

When the message paths taken depend only on the source and destination nodes, the routing is *deterministic*. Better performance and fault tolerance can be achieved by adapting the routes to network conditions. Such routing is referred to as *adaptive routing*. Usually, decisions are made based on local network conditions. Gathering global state information would add communication overhead to the system [42]. If the routing allows any of the shortest paths, it is fully adaptive, otherwise it is partially adaptive. An intermediate solution called *oblivious routing* selects a path from the available set of paths based not on network conditions but some other information. Most frequently the choice is random or pseudo-random.

Routing Techniques

Messages are usually divided into smaller units called packets before transmission. This way the utilization of communication bandwidth is more efficient and fair. The packet is the smallest unit of communication which contains routing and sequencing

information [42]. There are four different methods for transporting the packets to their destination.

The oldest approach is *store-and-forward routing*. Here the entire packet is stored in a buffer upon arriving at an intermediate node. When the required output link and an appropriate buffer at the next node is available, the packet is forwarded to the specified neighbor. Store-and-forward routing is the simplest, but the least efficient method. Since every packet has to be received in its entirety before forwarding it out, the network latency is

$$T = (L_p / B) \cdot D \qquad (1)$$

where L_p is the packet length, B is the communication bandwidth, and D is the distance between the source and destination node (i.e. the path length) [42]. With store-and-forward routing, the network topology is very important because the path length is a multiplicative factor in the message latency. The other disadvantage of this approach is the relatively high memory usage for packet buffers.

With *virtual cut-through*, a packet is forwarded directly from the input link to the output, unless the latter is busy, in which case the packet is stored in a buffer. The latency, assuming no other traffic in the network, is

$$T = (L_h / B) \cdot D + L_p / B \qquad (2)$$

where L_h is the length of the header [42]. The header is typically much smaller than a packet, therefore, the path length has very little effect on the latency. See Figure 2 for a comparison of latency with store-and-forward and virtual cut-through routing.

With *circuit switching*, the packet transmission is divided into three phases. During the circuit establishment phase, the whole path between the source and destination node

is reserved for the message. Then the packets are transmitted all the way to the destination one by one. Consequently, no buffering is necessary on intermediate nodes. In the last phase, the communication links are freed as the last packet of the message is being transmitted [42]. If the whole path cannot be established because of a busy communication link, the partial path is either torn down or it stays reserved until the link becomes available. In the former case, message delivery will be attempted later. The latency with circuit switching is

$$T = (L_c / B) \cdot D + L_p / B \quad (3)$$

where L_c is the length of the control packet [42]. Similarly to virtual cut-through, the dominant factor is L_p/B, since $L_c << L$. Therefore, the path length has negligible effect on the latency.

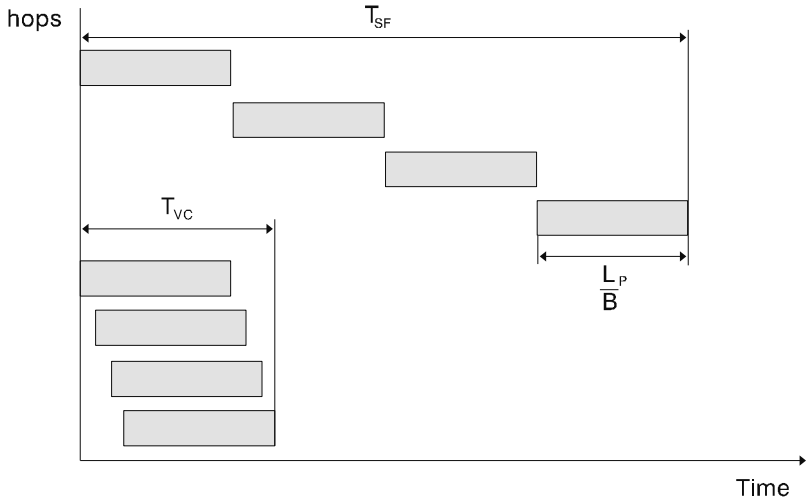

Figure 2 Latency of store-and-forward and virtual cut-through

Most modern distributed memory multicomputers use *wormhole routing*. This

approach is similar to virtual cut-through, but the packets are further divided into small flow-control digits, or flits. Only a limited number of flits are buffered on every intermediate node. The first couple of flits of a packet contain routing information. After examining these flits, every node starts to forward them down the next communication link in a pipeline fashion. This way, the whole packet spreads across the path. Except for the header, flits do not contain routing information, therefore, messages cannot be interleaved on communication links. When the first flit is blocked because of a busy channel, the flits stop advancing and remain distributed in the flit buffers across the path. The involved links stay blocked until the message can be delivered [18]. The latency of wormhole routing is

$$T = (L_f / B) \cdot D + L_p / B \qquad (4)$$

where L_f is the flit length [42]. Similarly to circuit switching and virtual cut-through, the path length has little effect on the latency, because the flit size is small.

The network latency of virtual cut-through, circuit switching, and wormhole routing are quite similar. Virtual cut-through requires more memory for storing buffers at intermediate nodes than wormhole routing, which stores only a couple of flits at any one node. Circuit switching reserves every communication link along a message path. Other messages cannot use these until the whole message is transported. We shall see later that wormhole routing, on the other hand, supports sharing of physical channels by many packets. In the next two sections, the concept of virtual channels and their use in deadlock avoidance and flow control will be described.

Deadlock Avoidance

A deadlock occurs when there is a cyclic dependency for resources in the network. With store-and-forward or virtual cut-through routing, the shared resources are packet buffers. Imagine a unidirectional ring of four nodes. When every node is trying to send a message to the node the furthest away, if every node allocates the single packet buffer on its neighbor, none of the messages can progress, a deadlock occurs. A message routing scheme in distributed memory multiprocessors must not allow deadlocks.

Structured Buffer Pool

A straightforward way to avoid deadlocks is to prevent cyclic dependencies between buffers by structured buffer pools [20]. A packet in a buffer on a given node can be stored only in a limited set of buffers on the next node. This relationship between packet buffers can be best described by a directed graph. The nodes of the graph are the buffers. There is an edge from a node N_i to another node N_j if a packet can move from buffer N_i to buffer N_j. This graph is called the buffer dependency graph. The message routing is deadlock-free if and only if the buffer dependency graph is acyclic [20]. An acyclic buffer dependency graph defines a partial order of the shared resources.

An acyclic buffer dependency graph can be achieved by allocating D_G number of buffers on each node, where D_G is the diameter of the network. Packets use buffers based on the current distance from the destination: if a packet is n hops away it is stored in buffer #n. Cycles cannot be formed if the routing is minimal [20].

The large amount of storage required for structured buffer pools makes this approach unattractive. A better solution can be achieved using wormhole routing with virtual channels.

Virtual Channels

Dally and Seitz introduces virtual channels for deadlock-free, connected, deterministic wormhole routing [18]. They use the following notation. The interconnection network, I= G(N,C) is a directed graph. The vertices N represent the processing nodes, the edges C the communication channels. Routing is a function R: $C \times N \rightarrow C$. It maps the current channel and the destination node to the next channel. Traditionally, deterministic routing has been a function R': $N \times N \rightarrow C$, i.e. at an intermediate node, the current and the destination node determined the forwarding channel. With their new approach, the input channel and the destination determines the output channel allowing them to introduce the idea of channel dependence. The *channel dependency graph* D for a given interconnection network I and routing function R is a directed graph. The vertices of D are the channels of I. The edges of D are the pairs of channels connected by R.

Deadlocks can occur when there are cyclic dependencies in the routing which result in cycles in the channel dependency graph. A routing function R for an interconnection network I is deadlock-free if and only if there are no cycles in the channel dependency graph D. Cycles in D can be broken by using multiple *virtual channels* per physical link [18].

A group of virtual channels shares the same physical link, but each virtual channel requires its own queue of flit buffers. A blocked message holds a virtual channel, but messages using other virtual channels sharing the same physical link can progress. Consider a unidirectional four-cycle. The channel dependency graph is cyclic. After

splitting each physical link to two virtual channels, a high c_{1i} and a low c_{0i} (Figure 3), and routing messages at a node numbered less than their destination node on the high channel and vice versa, the cycles are broken [18].

Multiplexing several virtual channels on the same physical link requires hardware support in the form of additional control logic and lines. Therefore, the number of virtual channels is bound by the available hardware.

In addition to helping deadlock avoidance, virtual channels increase the connectivity of the network facilitating mapping logical topologies required by certain applications to the given physical topology. They can be used in adaptive wormhole routing as well.

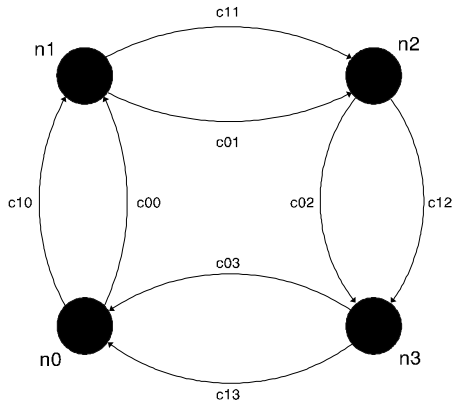

Figure 3 Unidirectional four-cycle with virtual channels

Adaptive Wormhole Routing

Duato develops a theoretical background for the design of deadlock-free adaptive routing algorithms for wormhole routing [24]. His work is based on the methodology of

Dally and Seitz described in the previous section. However, he allows adaptive routing, therefore, his routing function maps to P(C), the power set of C. More formally the routing function is R: $N \times N \rightarrow P(C)$. Note that the domain of R is $N \times N$ instead of $C \times N$ as in [18]. A selection function $S:P(C \times F) \rightarrow C$ selects a free output channel (if any) from the set provided by R. Here F is the channel status: F = {free, busy}. The channel dependency graph is defined as in [18].

He proves the straightforward extension of the theory in [18] for adaptive routing: an adaptive connected routing function R for an interconnection network I is deadlock-free if there are no cycles in the channel dependency graph D. Notice that an acyclic channel dependency graph here is only a sufficient condition for deadlock-free routing.

Duato goes on to introduce a routing subfunction and indirect channel dependency. A routing subfunction R1 for a given routing function R and channel subset $C1 \subseteq C$ is a routing function

$$R1: N \times N \rightarrow P(C1) \quad \text{where } R1(x,y) = R(x,y) \cap C1 \text{ for all } x, y \in N. \qquad (5)$$

There is an indirect dependency from nonadjacent channels c_i to c_j if and only if it is possible to establish a path from the source node of c_i to the destination node of c_j. c_i and c_j are the first and last channels in the path and the only ones belonging to C1. In other words, an indirect dependency is a dependency between two channels in the channel subset that exists only because of the intermediate use of one or more channels not in the subset. An extended channel dependency graph D_E for a given interconnection network and routing subfunction R1 is a directed graph. The vertices of D_E are the channels that define R1. The edges of D_E are the pairs of channels (c_i,c_j), such that there is a direct or an indirect dependency from c_i to c_j.

Duato proves in [24] that a connected adaptive routing function R for an interconnection network I is deadlock-free if there exists a subset of channels $C1 \subseteq C$ that defines a routing subfunction R1 which is connected and has no cycles in its extended channel dependency graph D_E. In other words, one can have an adaptive routing function with cyclic dependencies between channels as long as there are alternative paths without cyclic dependencies to send a given flit towards its destination. Physical channels can be split to multiple virtual channels to provide greater flexibility.

Duato provides two general methodologies to construct deadlock-free adaptive routing functions based on his new theorem. He supplies results of extensive simulation and concludes that his approach results in much lower latency than static routing and it improves throughput as well [24]. Schwiebert and Jayasihma develops an optimal routing algorithm for meshes based on Duato's work in [48].

Lin et al. proposes a methodology to prove deadlock freedom. If every packet that uses a given channel is guaranteed to reach its destination, then no deadlock can occur from the use of this channel. The routing is deadlock-free if no channel can be held forever by any packet regardless of the destination and path taken [39].

Duato proposes a necessary and sufficient condition for deadlock-free adaptive wormhole routing [23]. His proof technique applies only to routing relations of the form $R: N \times N \rightarrow C(P)$. Schwiebert et al. propose a necessary and sufficient condition for the more general routing relation $R: C \times N \rightarrow P(C)$. Routing relation of this latter form can always be converted to the first form, but not the other way around.

Schwiebert et al. define a variation of the channel dependency graph called the Channel Waiting Graph (CWG). The nodes of the CWG are the channels of the

interconnection network. There is a directed edge from c_i to c_j if the routing relation R allows a packet to wait for channel c_j after using c_i (not necessarily immediately after) [47]. The idea behind the CWG is that the routing allows a node to select an output channel from a set of available channels. When all possible channels are busy, however, the routing allows the packet to wait for a channel selected from a smaller set of channels. Note that the idea of the CWG is similar to that of Duato's routing subfunction R1.

Schwiebert et al. prove that a routing is deadlock-free if and only if it is wait-connected and its CWG has no True Cycles. A routing is wait-connected if for every input channel on a path, there exists a waiting channel through which the message can reach its destination. What are True Cycles? When creating the CWG, the specific intermediate channels are not considered between two dependent channels defining and edge in the CWG. Therefore, there can be cycles in the CWG which require a channel to be used more than once as an intermediate channel. Obviously, a deadlock cannot be caused by a cycle like this because a channel cannot be used by more than one packet at the same time. A True Cycle is a cycle in the CWG that allows every packet in the cycle to use channels not used by any other packet in the cycle. Schwiebert et al. use their necessary and sufficient condition for deadlock-free adaptive wormhole routing to develop a fully adaptive routing algorithm for hypercubes [47].

Topology-Based Deadlock Avoidance

Certain architectures have deadlock-free minimal routing strategies. A tree is a trivial example. Since there are no cycles in the network, there cannot be any cycles in the

channel dependency graph. Hence, any routing is deadlock-free. Meshes and hypercubes, on the other hand, have chordless 4-cycles in the network. Therefore, there are minimal routing strategies that are not deadlock-free. However, the so-called dimension ordered routing is guaranteed to be free of deadlocks for both meshes and hypercubes.

In a 2-D mesh, for example, if every message path is such that first the message travels in the X dimension all the way to correct column and then in the Y direction all the way to the destination, then it is easy to show that the channel dependency graph is acyclic.

The e-cube routing in hypercubes is a similar strategy. The nodes in a hypercube can be labeled with a binary number in a way that the Hamming distance of any two neighbors is 1. The bit positions of the labels are the dimensions. The routing algorithm takes the labels of the source and destination nodes and compares the bits from MSB to LSB (or in any other fixed order). If a difference is encountered, the message is sent to the neighbor with the label that differs from the current node in the given dimension. The channel dependency graph can be shown to be acyclic.

Flow Control

Flow control deals with the allocation of resources to packets as they travel across the network. Flow control methods are distinguished by how they resolve packet collisions. A collision occurs when a packet requests a resource held by another packet. A good flow control policy must avoid channel congestion and reduce network latency [42]. There are four basic flow control techniques [20]:

Buffering. The whole packet is received at the intermediate node and stored in a

buffer. Virtual cut-through employs this technique. The drawback of this approach is the large amount of storage required. Moreover, the buffers must be allocated in an acyclic manner to avoid deadlocks (structured buffer pool). The biggest advantage of buffering is that no network resources are wasted when a packet is blocked.

Blocking. The packet stops advancing holding the network resources already allocated and waiting for the busy resource. Wormhole routing follows this technique. Blocking flow control with a small number of flit buffers gives good performance for a given set of storage and communication resources.

Dropping. The whole packet is received at the intermediate node and thrown away. The packet must be retransmitted later. Dropping is inefficient because it wastes network resources. Beyond a certain level of network traffic, the throughput decreases, only a few messages are delivered. Dropping requires acknowledgement of delivered messages, further wasting bandwidth.

Misrouting. The packet is routed to an available but incorrect channel. This method results in non-minimal routing. Livelocks must be avoided with misrouting. A livelock occurs when a packet continues to be routed through the network but never reaches its destination.

Most networks combine some of these basic flow control techniques [20]. Dally proposes to use virtual channels not only for deadlock avoidance, but also for flow control [21]. There are minimal deadlock-free routing algorithms for certain topologies without using virtual channels (e.g. 2D meshes). For some others, two virtual channels per physical channel are enough. Dally's idea is to split every physical channel into several

virtual channels regardless. This way blocked packets can be passed, dramatically improving network performance.

Flow control is performed at two levels with virtual channel flow control. Virtual channels are assigned on a packet-by-packet basis. Physical channel bandwidth is allocated at the flit level. The routing algorithm assigns an arriving packet to an output virtual channel. The virtual channels associated with a physical channel arbitrate for bandwidth on a flit-by-flit basis. Decoupling resource allocation allows for more flexible physical channel utilization. Bandwidth may be allocated on the basis of type, age, or deadline. This advantage is especially important for real-time systems [21].

Example Systems

The Torus Routing Chip

The first implementation of wormhole routing with virtual channels was the torus routing chip (TRC) [19]. The TRC employs dimension order routing. A packet header contains two address bytes, one for the X direction and one for the Y direction. The addresses are relative, they contain a count of the number of channels the packet must traverse to reach the address of the destination in that dimension. First packets are routed in the X direction. The relative address is decremented at each step. When the address becomes 0, the correct X coordinate has been reached and the packet is routed in the Y direction until it arrives at the destination [19].

Both the X and Y physical channels are split into two virtual channels. In each dimension, a packet is routed on virtual channel 1 until it reaches its destination or

address 0 in the direction of routing. After a packet crosses address 0, it is routed on virtual channel 0. This routing strategy results in an acyclic channel dependency graph, therefore, it is deadlock-free [19].

The Torus Routing Chip is the result of Dally's and Seitz's groundbreaking work in message routing. Wormhole routing and virtual channels are two key elements in most of today's message passing parallel machines.

The Connection Machine

The original Connection Machine supports fine-grain parallelism. The prototype, the CM-1, consists of 64K extremely simple processing elements. Each unit has 8 bits of internal state information. Data paths are one bit wide. An operation consists of combining one bit of state information with two bits from external memory according to some specified logical operation generating two result bits [30].

The interconnection network of the Connection Machine is indirect. A router is connected to 16 processing elements. The 4096 routers form 12 dimensional hypercube. Every message contains the relative address of the destination. The router cycles through the dimensions. At step K, it searches all messages it has. It selects the oldest available message that has its bit K set in the address and forwards it along dimension K. Messages that are not sent in the given cycle are buffered. When the router runs out of buffers space, newly arriving messages can be misrouted [30].

The CM-5

The Connection Machine Model CM-5 contains between 32 and 16K processing

nodes. Each node consists of a SPARC processor, memory, and a high-performance vector processing unit. The machine contains three communication networks: a data network, a control network, and a diagnostic network. The control network provides cooperative operations, e.g. synchronization. The diagnostic network allows independent access to all hardware resources to detect and isolate errors [38].

The data network provides fast point-to-point communication between processing elements. The topology of this indirect network is a 4-ary fat tree. Each processor is connected to two routers. Routers at the first two levels are connected to two routers at the next level. Routers higher up are connected to four routers at the next level. The network bandwidth scales linearly up to 16K nodes.

Routing in the data network is partially adaptive, minimal wormhole routing. Messages progressing upward have several possible paths to take to reach the level of the least common ancestor of the source and destination nodes. The message header contains routing instructions allowing the routers to select an available channel pseudo-randomly. Then messages move along the single available path down the tree to the destination [38].

The IBM SP2

The IBM Scalable POWERparallel Systems SP2 is an indirect network connecting RS6000 workstations. The routing chip, called the High-Performance Switch, provides high communication bandwidth [54]. In the SP2, multiple 4-way to 4-way bidirectional switching elements are used. Except for nodes directly attached to the same switch, there are multiple shortest paths for all node pairs. For SP2 topologies, any minimal routing is inherently deadlock-free [54].

SP2 employs deterministic (and supports oblivious) source wormhole routing. Before sending out a message, each node performs a routing table-lookup and inserts routing information into the packet headers. The routing tables are generated by a breadth-first search performed on the network detected by a worm program providing fault-tolerance.

The SP2 flow control strategy differs from traditional wormhole routing. Each switching element contains a relatively large buffer to store blocked messages. This central queue is shared among input channels. Storage is allocated dynamically according to demand. This strategy reduces the impact of network contention [54].

The MIT J-Machine

The J-Machine is an experimental multicomputer developed at the MIT Artificial Intelligence Laboratory. The system provides low level support for synchronization of threads, data communication, and global naming of objects [44]. The processing node of the J-Machine is the Message Driven Processor (MDP). The MDP consists of a fixed-point CPU, a memory management unit, a router, a network interface, some SRAM, and a DRAM controller, all integrated on a single VLSI chip. The topology of the interconnection network is a 3D mesh. The J-Machine employs deterministic, dimension order wormhole routing [44].

The CRAY T3D

The CRAY T3D is one of the most powerful supercomputers. A 3 dimensional torus network connects from 32 up to 2048 processing elements. A node contains two 64-bit DEC RISC processors. Both has up to 8 Mbytes of local memory. Each node has a

network router chip. The CRAY T3D employs dimension order deterministic wormhole routing. Packets are transported first in the X dimension in either positive or negative direction. Then the Y and finally the Z dimension follows. Deadlocks are avoided by splitting each physical channel into two virtual channels. In each dimension, there is one selected physical channel called the dateline communication link. Packets that will cross that channel use exclusively the lower virtual channels in that dimension. Packets that will not cross the "dateline" use the higher channels in that dimension. The channel dependency graph is, therefore, acyclic.

The IMS T9000

The IMS T9000 represents the new generation of transputers from Inmos. The processor has superscalar architecture, hardware scheduler, on-chip cache, and a communication processor. Multiple processes are scheduled by the T9000; they can communicate via channels. The four bidirectional communication links of the T9000 allow multiple transputers to be connected together. Processes running on different processors can communicate with each other transparently. Each communicating process pair has its own virtual channel. Multiple virtual channels are multiplexed onto the same physical link. The T9000 does not support message forwarding in hardware. Inmos provides this capability in a separate routing chip, the IMS C104, for T9000 systems [41].

The IMS C104 includes a full 32 x 32 non-blocking crossbar switch. It implements wormhole routing, however, it does not support virtual channels. Therefore, deadlock avoidance must be solved by other means. The router employs interval labeling. Labeling algorithms for meshes, hypercubes, and trees can be found in [41].

The T9000 is a good parallel processor. It employs state-of-the-art concepts, such as wormhole routing with virtual channels. The concept of virtual channel flow control advocated by Dally [21] is already implemented in the T9000. Moreover, Inmos realized the need for a separate chip to support message routing. Unfortunately, there is a mismatch between the processor and the routing chip, since the latter does not support virtual channels. This decreases the usefulness of the T9000 as a building block for large parallel systems.

The TI TMS320C40

The Texas Instruments TMS320C40 is a 32-bit floating-point digital signal processor. It has six bidirectional communication ports driven by independent DMA engines. These ports allow glueless point-to-point connection between processors [52]. Multicomputer networks of arbitrary size and topology (limited by the maximum degree of one node) can be constructed from C40s. Unfortunately, the processor does not provide any additional support for message routing. Physical channels cannot be split to multiple virtual channels.

A message routing library for C40 networks is available from TI. The routing is software controlled, the CPU makes routing decisions and keeps track of the status of the communication links. The routing is distributed and wormhole-like except for a specific case, when buffering the message is necessary to avoid a deadlock condition due the C40 communication channel protocol [1].

The TI TMS320C40 is a fast floating-point processor. Its six bidirectional communication channels driven by DMA engines make the processor a good candidate

for parallel processing. Message routing, however, is not supported in hardware. There is no routing chip available for the C40. Bus synchronization issues make it very hard to implement routing without interaction with the CPU. Without virtual channels, deadlock avoidance must be achieved by other means.

Evaluation

Wormhole routing is the clear choice for message routing in today's parallel machines. The two key characteristics of it are the small message latency and the low buffer memory requirement. Wormhole routing with virtual channel flow control offers even better capabilities, including deadlock avoidance, traffic isolation and protection, and improved network efficiency and performance. Suresh, however, points out that they are expensive to support [43]. Currently, only a handful of commercial machines implement virtual channels. The Cray T3D, for example, use them only for deadlock avoidance in its torus network. The Inmos T9000 is the only industrial product that utilizes virtual channel flow control. On the other, the accompanying router chip does not support virtual channels at all.

Most experts agree that adaptive routing has good potentials [43]. Scott observes that this approach offers greater bandwidth and better worst-case behavior than deterministic routing [43]. The only drawback is the need for more complicated routers to handle, among other things, out of order arrival. The CM-5 is one of the few machines to support adaptive routing.

Most high-end parallel machines employ regular topologies with guaranteed deadlock-free routing, such as hypercubes or meshes. Some apply other topologies and

virtual channels to avoid deadlocks. Current trends indicate that the next generation parallel supercomputer will employ a 3D mesh (or torus) or a fat-tree like multistage topology, fully adaptive wormhole routing, and virtual channel flow control.

Nowadays building inexpensive, dedicated parallel machines is possible with such building blocks as the transputer or the TMS320C40. They do not support virtual channels or adaptive routing in hardware. Existing solutions use either deadlock-free topologies or avoid wormhole routing altogether. Since one of the advantages of these systems is their flexible topology, restricting the possible topologies to a narrow class of graphs is not acceptable. Since wormhole routing provides the best performance of all routing techniques, it would be desirable to use it with these systems. How to provide deadlock-free wormhole routing in these networks with flexible topology is an open problem.

Assignment

The assignment problem (also known as the mapping problem) is the placement of the processes of a parallel program on a network of processors in such a way that the program execution time is minimized. The goal can be achieved by balancing two factors: (1) distributing the computational load on the nodes evenly, and (2) minimizing interprocessor communication. These factors, however, are not orthogonal. For example, assigning all the processes to a single processor results in zero communication overhead, but the load distribution is the worst possible.

Since generating candidate solutions for the problem and measuring their execution time is not feasible in most cases, a cost function must be defined. The cost function describes the quality of the solution. It should be correlated to the actual execution time

as closely as possible. The more complicated the cost function is, the better it can describe the assignment, but the more time it takes to compute. The next sections describes several cost functions proposed in the literature.

A program consisting of multiple communicating processes can be best described by a directed weighted graph. The nodes of the graph are the processes. A directed edge between node A and B specifies that A is sending messages to B. The weights of the edges represent the communication rate on the given edge, while the weights on nodes specify the estimated execution time of the process. Similarly, the processor interconnection network can be described as a directed weighted graph. The nodes are the processors, the edges are the communication channels between the processors. The weights of the edges represent the capacity of the channels, the node weights specify the computational capacity of the nodes. In addition to specifying communication between processes, the edges of the process graph can also specify precedence constraints.

The assignment problem is the mapping of one graph to another minimizing a cost function. This is a well-known NP-complete problem [27, 25]. Several different approaches have been proposed in the literature. Talbi and Muntean give a taxonomy of mapping strategies in [55] (Figure 4). Optimal solutions find the global minimum, but they can be used only for small problems because of the required time. General optimization methods, such as genetic algorithms or simulated annealing, have been successfully applied to the assignment problem. Most approaches, however, apply heuristics specifically designed for the problem.

One interesting approach generates an initial assignment and improves it during runtime based on the actual timing of the executing system. This technique is called

dynamic load balancing. Moving a process from a processor to a different node is referred to as process migration. This dissertation concentrates on static assignment.

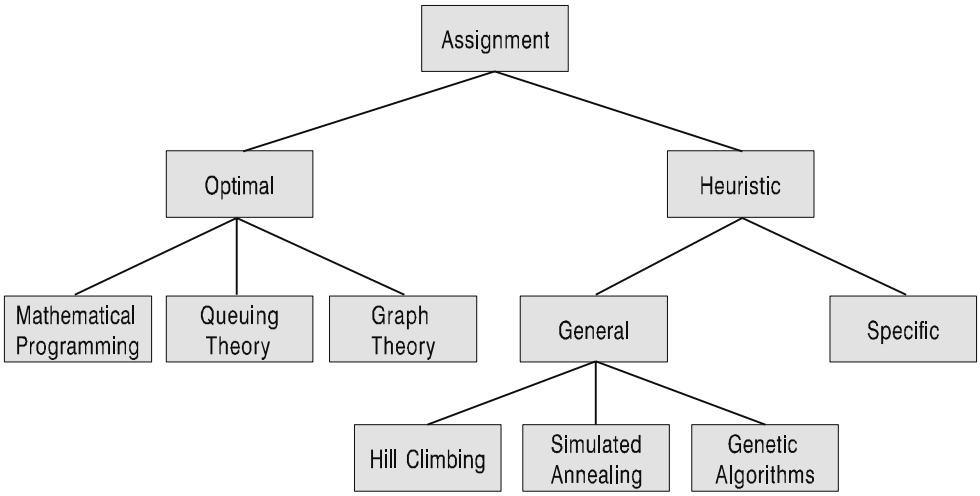

Figure 4 Taxonomy of process assignment strategies

Cost Functions

There are two somewhat contradictory requirements for a good cost function. First, it must describe the assignment configurations as accurately as possible, i.e. there must be a strong correlation between the value of the cost function and the execution time of the program for a given assignment. Second, the cost function must be inexpensive to compute. The faster the cost can be computed the more possibilities the search algorithm can evaluate in a limited time.

The simplest cost function is proposed by Bokhari in [14]. He assumes that the number of processes is equal to the number of processors. Hence, the computational load

does not need to be balanced. The cost function is defined as the cardinality of the assignment which means the number of process graph edges falling on single processor graph edges. This cost function does not consider any weights on any of the graphs. Moreover, it does not take the distance between non-neighbor processors having communicating processes assigned to them into account. The main advantage of this approach is its simplicity.

A better measure of the communication cost is the sum of the products of the weights of process graph edges and the corresponding distances in the processor graph [37]. However, this cost function does not take shared physical links into account. Two or more process graph edges may share the same physical channel causing delays. The actual communication overhead depends on the particular assignment, the timing of the communication between the processes, and the communication control rules of the system [37]. Even if the necessary information is available, such an elaborate cost function would be computationally expensive.

Lee proposes to sort the process graph edges into sets corresponding to phases in the communication. Edges in the same set are needed at the same time, while edges in different sets do not overlap [37]. His cost function is the sum of the maximum communication overhead in the same set over all sets. This measure is quite reasonable, but the timing information may not be available or accurate enough to partition the process graph edges into sets. Non-overlapping phases may not even exist in the system.

The cost function for computation overhead is typically the variance of the processor loads [55]. The measure for the communication and computation overheads are usually combined according to the simple formula:

$$C = C_{comm} + W \cdot C_{comp} \quad (6)$$

where W is an empirical constant, C_{comm} is the communication cost, and C_{comp} is the computational cost.

Optimal Solutions

Several methods finding the global minimum of the cost function have been proposed. Some interesting approaches are based on graph theoretic foundation. Others employ mathematical programming or queuing theory (Figure 4).

Graph Theoretical Approaches

Stone applies network flow algorithms to solve the assignment problem [53]. The maximum flow problem involves directed acyclic graphs. Nodes having outgoing edges only are called source nodes. Nodes with incoming edges only are called sink nodes. Source nodes are capable of producing an infinite amount of commodity. Similarly, sink nodes can absorb an infinite amount of commodity. The edges of the graph are labeled by integer pairs. The first number specifies the capacity of edges, i.e. the maximum amount of commodity flow it can transport. The second number specifies the amount of the current flow. A feasible flow in the network originating from the sources and ending at the sinks has the following properties [53]:

The sum of flows into an intermediate (i.e. neither source, nor sink) node equals to the flow out of the node.

The flow on any edge in the network is non-negative and does not exceed the capacity of the edge.

The value of a flow is the sum of the flows out of the source nodes which must be equal to the flow incoming to sink nodes. The maximum flow is a feasible flow with the highest value. The max-flow, min-cut theorem states that value of the maximum flow in a network equals to the weight of the minimum weight cutset. There are several efficient algorithms for finding the maximum flow (and the minimum cut) in a network [53].

Stone describes a program consisting of communicating processes by a directed acyclic graph. There are weights on the edges representing the communication requirements between the processes. Stone observes that the minimum cut of this graph is the optimal assignment of the processes on two processors if only the communication overhead is considered. He wants to incorporate the process execution time on the different processors into his model. He adds two nodes to the network, one representing each processor. The first node P1 is a source, the second P2 is a sink. He adds an edge between every original node and the two new ones. The weight on the edges of P1 represent the execution time if the process is assigned to the second (!) node and vice versa. The minimum cut of this graph is the optimal assignment of the program on two processors [53].

Stone tries to generalize his results for N processors. While he succeeds in formalizing the problem in terms of minimum cutsets, no efficient algorithm is found. This is not surprising, since the general assignment problem is NP-complete. Bokhari discusses several efficient algorithms for special cases of the assignment problem, including tree-structured process graphs and single-host multiple-satellite computer systems [15].

Shen and Tsai propose to use a type of graph matching called weak homomorphism for the assignment problem [49]: Let $G_1 = (V_1 , E_1)$ and $G_2 = (V_2 , E_2)$ be two graphs.

G_1 is weakly homomorphic to G_2 if there exist a mapping M: $V_1 \rightarrow V_2$ such that if edge (a , b) $\in$ E_1, then edge (M(a) , M(b)) $\in$ E_2. They propose to find the weak homomorphism with the smallest cost between process graph and a processor network. In order for this approach to work, they need to add a self loop to each node in the processor graph when there are more processes than processors. Note that neighboring processes will be assigned to neighboring processors which may not result in the optimal solution. Since the best algorithm for finding the minimal cost weak homomorphism has exponential time complexity, Shen and Tsai propose a heuristic search method that is not guaranteed to find the best solution.

Mathematical Programming

Ma et al. apply the branch-and-bound technique to solve the assignment problem [40]. The search space is represented by a tree. The levels of the tree correspond to processes. An allocation decision represents a branching at a node. For example, there are N branches (where N is the number of processors) at any node. Branch k of any node on level m means that process m is assigned to processor k. A path from the root to a leaf represents a complete assignment. Although Ma et al. propose several constraints pruning the search tree, the time complexity of the algorithm is still exponential [40].

General Optimization Methods

Hill Climbing

Hill climbing is a simple general combinatorial optimization method. The algorithm

starts from a complete solution and tries to improve it by local transformations. A new configuration is accepted only if it provides a better solution than the previous one. This process is repeated until no improvement is possible. The algorithm finds a local minimum of the cost function. The usual way to find a better solution is to repeat the whole procedure several times starting from different randomly generated initial configurations. A hill climbing algorithm applied to the assignment problem is described in [55].

Simulated Annealing

Annealing is the process of heating a substance and then cooling it down to reach a low-energy state of the matter. Lowering the temperature must be done slowly, and increasing amount of time must be spent at lower temperatures. Otherwise, the substance gets out of equilibrium, the resulting crystal will have defects, or the substance may form glass containing only metastable, locally optimal structures [34].

Kirkpatrick et al. propose to apply the Metropolis algorithm used for approximate numerical simulation of annealing for combinatorial optimization problems [34]. The cost function is defined to represent the "energy" of the system, while the "temperature" is a control parameter having the same unit as the cost function. The process starts in a high-temperature state. The system is "cooled down" slowly. At each step, small random changes are applied to the system. Changes resulting in a lower-energy state (i.e. smaller cost) are always accepted. Higher energy states are accepted with a finite probability. This feature prevents simulated annealing to get stuck in a local minimum. The probability of the acceptance of increased energy state is decreasing with the temperature. The process

ends in a low energy state, which may not be the global minimum [16].

The simulated annealing method applied to a combinatorial optimization problem has four elements [16]:

The System Configuration must be a good description of the problem allowing for easy specification of system perturbations and efficient calculation of the cost function.

The Move Set is the collection of allowed rearrangement operations. Having a large number of moves is desirable, therefore, they must be simple to generate and evaluate.

The Cost Function must be incrementally computable for efficiency.

The Annealing Schedule defines how the temperature is changed during annealing including initial and final temperatures, cooling rate, and the time spent at each temperature value.

Bollinger and Midkiff apply simulated annealing to the assignment problem [16]. They assume that the number of processors and processes are the same, and every process is assigned to a unique processor. Therefore, their cost functions represent only communication overhead. They apply a two-stage approach using simulated annealing in both cases. First they optimize the processor placement using a simple cost function. In the second stage, they generate the message routes to avoid congestion.

The annealing schedule in both phases is based on a simple logarithmic formula:

$$T_n = \alpha^n \cdot T_{n-1} \tag{7}$$

where the annealing factor α is at least 0.9 ensuring a slow cooling rate. For the processor placement, the cost function combines the average and the maximum communication

time. The moves are simple pairwise exchanges of processes. In the second phase, the communication links between processes are mapped onto the physical channels. Minimal routing is not required. The move set is more complicated in this phase because of the nature of the problem. The cost function is computed from the average and maximum distances between communicating processes.

Bollinger and Midkiff provides test cases with encouraging results. Their method finds near optimal solutions in most cases. However, the assumption that the number of processes equals to the number of processors is a severe limitation. In general, simulated annealing is a good optimization approach. It works well for the assignment problem. The biggest drawback is the required time because of the large number of steps required for real size problems.

Genetic Algorithms

In his famous paper, Holland proposes the idea of genetic algorithms – search strategies based on the mechanics of natural selection and genetics [32]. They differ from traditional optimization procedures in several ways [28]:

They work with a *coding* of the parameter set, not the parameters themselves.

They search from a *set of points* (called population), not a single point.

They use *probabilistic* transition rules, not deterministic ones.

Genetic algorithms require the parameter set of the optimization problem to be coded as a finite-length string over some finite alphabet [28]. The search starts from a set of configurations. The basic genetic operators (reproduction, crossover, mutation) are applied to them to generate a new population. With reproduction, individual strings are copied

based on their cost function (fitness value). The higher the fitness, the more likely the string will contribute one or more offsprings to the next generation. With crossover, pairs of strings are selected randomly and their characters are swapped at a randomly selected position. For example, when the two strings ABCDEF and UVWXYZ are crossed at position number 3, the two new strings generated are ABCXYZ and UVWDEF. Reproduction and crossover give genetic algorithms much of their power. However, occasionally they may lose some important genetic material (i.e. a letter at a particular location) [28]. The third operator, mutation is designed to overcome this problem. Mutation is the random alteration of the value at a selected string location. Mutation, however, should be used sparingly [28].

Talbi and Muntean apply genetic algorithms for the assignment problem [55]. They label each processor by a unique symbol; the set of these symbols constitute the alphabet. A string describing a configuration is of length N, where N is the number of processes. A symbol S at position k means that process k is assigned to processor S. They implement a standard genetic algorithm in parallel. They compare the approach to hill-climbing and simulated annealing.

In general, genetic algorithms and simulated annealing provide good solutions to the assignment problem. Genetic algorithms are faster, mainly because they are easier to parallelize.

Assignment-Specific Heuristics

A popular solution to the assignment problem consists of two phases. First, an initial assignment is generated using either a simple cost function, or a simple search method,

or both. In the second phase, the initial assignment is improved by local transformations. Usually, a more complicated cost function is used than in the first phase. The initial assignment cuts the search space allowing a more expensive search in the second phase. Such an approach is taken by Lee and Aggarwal [37].

They define the communication intensity of a process node by adding the weights of all of its edges. The mapping algorithm starts by assigning the node with the highest intensity to a processor. Then they repeatedly assign the highest communication intensity process that is adjacent to at least one of the already assigned processes. At each step, the assignment is made in such a way that the cost function is minimized [37].

In the second phase, they try to improve the assignment by pairwise exchanges of processes. They select a promising process according to a criterion derived from the cost function (which may be different than the one applied in the first phase). Then they check every pair involving the selected process and perform the exchange with the process giving the smallest cost if it is smaller than the original cost. The algorithm stops when the cost is acceptable or no further improvement is possible.

Fully Connected Processor Graph

A popular and widely available distributed processing platform is local area networks connecting workstations. They employ broadcast-based communication, hence the processor graph is uniform and fully connected. In this case, there is no need for a processor graph at all. The assignment problem is to divide the process graph into n subgraphs where n is the number of processors. Unfortunately, this problem is still NP-complete [9].

Paralex is a programming environment for distributed workstations [9]. Programs in Paralex can be specified as weighted directed acyclic graphs. The graph represents dataflow, therefore, the edges not only specify communication between processes, but precedence constraints as well. Nodes lying along a chain of edges must be executed sequentially. A straightforward approach to the assignment problem is to try the group chains of nodes together [9].

Paralex employs a greedy heuristic algorithm to identify chains. First, every node is put into a chain by itself. Then Paralex selects the two non-parallel nodes communicating most intensely and puts them in a single chain. Edges originating in the first node of the new chain are deleted. This step is repeated until no more chains are found [9]. Note that the number of chains found is not necessarily equal to the number of available processors. Paralex is able to incorporate assignment constraints set by the user. Certain processes can be specified to be (or not to be) assigned to certain nodes.

The Impact of Message Routing on the Assignment

Message routing has a considerable impact on the communication load in interconnection networks. Therefore, some approaches incorporate message routing into their assignment strategy. Shen assigns arbitrary process graphs to a 2D torus network of transputers [50]. He applies a three-phase strategy. First, he groups the processes into clusters to get the same number of processes as processors. His message routing strategy requires edge-disjoint paths between communicating processors, therefore, no process in the clustered process graph can have more edges than the number of physical channels the processors have. The grouping algorithm must satisfy these constraints, while

balancing computation and communication load.

In the second phase, a one-to-one assignment of the processes to processors is generated. The algorithm tries to minimize the average distance between communicating processors. Shen applies a neighbor-first heuristic. The search starts by placing the most I/O intensive process on the host node and placing neighbor processes on neighboring processors if possible. In the last phase, a heuristic search for edge-disjoint message routes is performed.

Shen's approach has two major drawbacks. First, edge-disjoint routing is not possible for all process graphs. Second, even when there exists one, his algorithm is not guaranteed to find it. He realizes this problem and suggest the placement and routing phases to be run iteratively until a solution is found. The quality of the assignment is questionable, since all three phases of his approach relies heavily on heuristics.

Dixit-Radiya and Panda propose a task assignment strategy for systems with adaptive wormhole routing [22]. They employ a Temporal Communication Graph (TCG) to model task graphs and to identify communication steps that conflict both temporally and spatially. A TCG is a directed acyclic connected graph. Nodes represent computation stages of processes. Edges between nodes corresponding to the same process are called sequence edges. Edges between nodes corresponding to different processes are called communication edges. Weights on edges specify communication load. The weight of sequence edges are zero. Weights on nodes represent computation load.

TCG representation incorporates sufficient temporal information to estimate the timing of computational and communication steps of the application [22]. Dixit-Radiya and Panda define parameters, such as Earliest Start Time, Earliest Finish Time, Latest Start

Time, and Latest Finish Time. Channel contention delays are assumed to be zero when computing these estimates. The actual value of these parameters are obtained by running the TCG on a simulator. The Actual Earliest Finish Time is the cost function itself. The difference between the actual and estimated parameters gives information about link contention in the system.

The assignment is carried out in two phases. The initial assignment phase uses a simple heuristic trying to minimize the distance between heavily communicating processes. This heuristic does not consider link contention. The second phase starts from the solution provided by the initial assignment. The objective is to minimize the maximum link contention. The algorithm locates the edge in the TCG with maximum link contention and tries to decrease the distance between the involved processes by pairwise exchanges.

The merit of this approach is that it utilizes information on the actual communication overhead by considering link contention. However, it relies heavily on the accuracy of the temporal information captured in the TCG. Other limitations include the process graph being acyclic and the need to simulate the application to gather temporal information.

Topology Synthesis

Lee and Smitley take a significantly different approach. They try to solve the assignment problem for reconfigurable interconnection networks. Instead of assigning the process graph to a fixed architecture, they configure the topology of the processor network to match the process graph [36].

They assume that they have n processors, each with d communication channels, and

a process graph with n processes. Moreover, they assume that the amount of communication between connected processes are the same. Their objective is to synthesize a processor graph maximizing the number of pairs of intercommunicating processes that fall on neighboring processors, while keeping the maximum degree smaller than or equal to d. Furthermore, the resulting processors network must be connected. Unfortunately, they show that the problem is NP-complete [36].

However, a polynomial time complexity algorithm exists if the restriction that the resultant graph be connected is removed. They develop a two-phase approach. First, they employ the algorithm to find the optimal solution consisting of possibly unconnected components. Then they apply a heuristic technique to connect the components. The first straightforward step is to connect components which have communicating processors and also have nodes with degrees smaller than d. If the graph is still not connected, some edges have to be removed to connect the remaining components [36].

The approach attacks the problem from an interesting new angle. It can be generalized for weighted process graphs [36]. The most important limitation of the technique is that the number of processors and processes must be equal. The heuristic, that the more neighboring processes are assigned to neighboring processors the better the solution is, is too simplistic.

Evaluation

The assignment is a well-known, much studied problem of computer engineering. The time complexity of finding the global minimum of the cost function is exponential. Therefore, optimal solutions are feasible for small systems only. Different heuristic

approaches have different advantages and drawbacks. In general, a particular solution must be tailored to the constraints and requirements of the given application.

A good balance must be found between the accuracy and computational price of the applied cost function. A more extensive search can be done with a simple cost function. On the other hand, is finding even the global minimum of a cheap cost function worthwhile if it does not describe the quality of the assignment accurately? A more complicated cost function is more desirable even at the expense of the search strategy.

A good cost function must describe the overhead of interprocessor communication as accurately as possible. Since physical communication channels are shared resources, it is not enough to consider the distance a message must take or even the specific route itself, message contention must be taken into account. Two approaches discussed earlier address this issue. Shen allows only edge disjoint message paths eliminating message contention entirely. This is, however, a severe restriction. Dixit-Radiya and Panda require extensive timing information of the application and compute the level of message contention. Without running the application, however, the timing information is not accurate enough to provide a good description of the communication patterns of the program. Finding an accurate estimate of the communication overhead is still an open problem.

The time complexity of general optimization techniques, such as simulated annealing or genetic algorithms, is high. Large number of configurations must be evaluated before an acceptable solutions can be found. Even with simple cost functions, the price of these approaches can be prohibitive.

Assignment-specific heuristics have the advantage of utilizing the constraints and other attributes of the given problem. Most such approaches divide the problem into several

phases. First, they cluster the processes, so that the number of processors and processes become equal. Then the assignment is generated. Finally, the message routes are produced. Unfortunately, these parts of the problem have strong correlation with each other. A unified approach, if feasible, would provide a better solution.

For parallel architectures with flexible topology, synthesizing the processor network to match the topology of the process graph has its merits. The same considerations regarding the cost function and the search strategy apply here as well.

CHAPTER III

AUTOMATIC PARALLEL APPLICATION SYNTHESIS

Problem Statement

The objective of this dissertation research is to develop a framework for automatic synthesis of large-scale, parallel instrumentation and signal processing applications characterized by high I/O bandwidth, computationally intensive processing requirements, and frequently changing software specifications and hardware configurations. The target hardware platform is distributed memory multiprocessors with flexible interconnection topology. To achieve this goal, the following issues are addressed:

Representation. The application specifications must be represented in a computer readable format to facilitate automatic application synthesis. Furthermore, the representation format must be easily comprehensible by humans. In order to manage the complexity introduced by low level parallel processing and systems engineering issues, high level system descriptions are needed. The specifications must include the application requirements and the available software and hardware resources. The representation technique must provide means to manage the complexity of the specifications themselves.

Automatic application synthesis. The parallel instrumentation application must be automatically synthesized from the high level system specifications. The software system needs to be partitioned and assigned to the hardware platform. Executables, message routing information, and network loader configuration are

to be automatically generated. The specific requirements of the process assignment and the message routing strategy are as follows:

Process assignment. Process assignment must be carried out automatically in order to optimize the performance of the synthesized system. A cost function is needed that accurately describes the quality of the assignment. Locating the optimal solution cannot be guaranteed because the problem is NP-complete. The search space must be restricted to keep system synthesis time polynomially bound.

Message routing. Deadlock-free wormhole routing in networks with arbitrary topologies is an open problem. Deadlock-freedom must be guaranteed. Minimizing the communication overhead is critical to the performance of the system.

The following restrictions are placed on the problem domain to keep the research well focused and the problems in the preceding list manageable:

Signal flow dominance. The class of targeted applications are limited to signal flow dominant systems. The structure of such systems can be described by a signal flow graph.

Static structure. The signal flow graph of the system is static. Dynamic reconfiguration is not permitted.

Continuous execution. The execution of the signal flow graph is continuous. Processing of consecutive input sets overlap in a pipeline fashion.

Throughput. The objective of the system synthesis process is to maximize system throughput. Real-time constraints are not considered.

The solution domain is restricted by the following factors:

Task parallelism. The data parallel computational model is not considered.

Hardware platform. The target hardware platform is distributed memory multiprocessors with flexible interconnection topology.

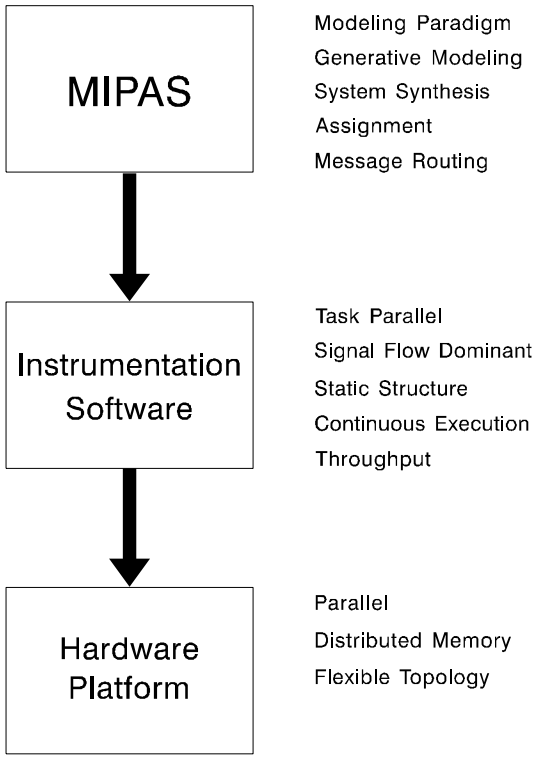

Figure 5 Model-Integrated Approach

Model-Integrated Parallel Application Synthesis

The problem of automatically synthesizing large-scale, parallel instrumentation and signal processing applications for distributed memory multiprocessors with flexible interconnection topology is solved in the framework of the Multigraph Architecture (MGA). Figure 5 illustrates the approach. The Model-Integrated Parallel Application Synthesizer (MIPAS) includes the configurable graphical modeling environment and the

model database of the MGA along with the domain-specific model interpreters.

In order to manage the high complexity of the system models, the declarative modeling capabilities of the MGA are augmented by an additional model organization principle: generative model building. The parallel instrumentation domain mandates three modeling aspects: signal flow, hardware, and assignment constraints aspects. It is the task of the model interpreters to partition the signal flow graph and assign the partitions to the nodes of the processor network while satisfying the assignment constraints. A deadlock-free wormhole routing strategy for networks with arbitrary topologies is developed and integrated into the model interpreters. The output of the MIPAS includes an executable, a dataflow graph partition, and a message routing map for each processor in the system.

CHAPTER IV

MODELING PARADIGM

The Multigraph Architecture (MGA) supports declarative modeling of complex systems. It provides several model organizational principles to manage the complexity of the system models. Multiple aspects, model types and instances support modular modeling. Model references aid in the description of interactions between modeling aspects. Hierarchy and multiple views provide visibility control. While these are powerful techniques that help the management of the complexity of the system models, experience shows that there is a clear need for an additional model organization principle primarily for modeling repetitive structures. The following section describes how generative modeling can satisfy this need, and how it can be incorporated into the declarative modeling environment of the Multigraph Architecture.

Generative Modeling

When several parts of a model have the same components and structure, simple replicators could reduce the complexity of the models. Instead of repetitively building the same model for every occurrence, one copy and the desired number of replications could be specified. However, the interface of such replicators poses problems. Since there is only one actual copy of the model, only one connection can be made to each of its ports. Such a connection could be interpreted as one connection to each replicated instance or as a single connection to the first instance. Some complicated constructs could be defined

for different cases, but the solution would not be intuitive and easy to use. A situation similar to that of the replicators exists with conditional model components whose existence depend on some condition.

Conditional models are very useful for modeling complex systems. For example, changing requirements and varying resources may force the user to change the system models frequently. The required "size" of the system changes most often. For instance, the number of channels required in a multi-channel system can vary from day to day. Similarly, the number of available hardware resources, such as processing nodes, disk drives, printers, etc., can also change frequently. Editing the system models often is cumbersome and error-prone. A simple solution is to model the biggest expected configuration and conditionalize parts of it. Conditionals are similar to replicators because they specify the number of occurrences of a model component, which can be zero or one. These two modeling constructs, replicators and conditionals, can be combined and implemented with generative modeling.

With generative modeling, the user can specify model structure, i.e. components and connections, by writing a program in some language. Generative modeling is similar to the generate statements of VHDL [45]. To interface this style of model building to the declarative (graphical) modeling paradigm of the MGA, the textual attribute feature of the modeling environment is utilized.

Each component of a Multigraph model can have multiple textual attributes to capture information that cannot be represented graphically. A varying number of textual attributes are dedicated to generative modeling depending on the type of the model component. Model components with inner structure have a *structure attribute* that is used to create

new, or destroy existing connections between parts of the given model. Every model component has a *repetition attribute*, which expresses the number of repetitions of the current model. A *reference attribute* is assigned to models that contain references to components in other aspects. Since model references can be made only to graphically specified model components, this attribute is used to refer to components specified by generative modeling.

These textual attributes are called *generative attributes*. The language selected for generative modeling is the C++ programming language because it is widely used and compilers are readily available.

The primary modeling methodology in the Model-Integrated Parallel Application Synthesizer is the only method supported directly by the MGA: declarative modeling. Generative modeling plays a secondary role. It is reflected in the fact that generative modeling is implemented through textual attributes which are assigned to graphical model components. A totally new model component cannot be generated, only replications of an existing graphical model can be. This is intentional; declarative modeling is a powerful methodology and its usage is favored in the MIPAS. Generative modeling is supported only to augment the capabilities of graphical model building. Its main purpose is the compact description of repetitive structures and the flexible specification of conditional components.

This double paradigm, declarative and generative, has a minor drawback. Neither the graphical, nor the textual model representation contains all the information about the system. Consequently, the models are hard to comprehend by humans. The MIPAS contains a special model interpreter dedicated to overcome this problem. The Model

Transformation Tool (MTT) converts these mixed models to purely graphical representation by evaluating the generative attributes and creating a new model database. The MTT is described in greater detail in the following chapter.

Generative modeling is very useful for reducing complexity and speeding up the modeling process. Along with the MTT, it transforms modeling into a two-stage process. The user first creates the models with the mixed declarative and generative specifications. Then the MTT is used to transform the models. Then the user can apply the Graphical Model Builder again to check the models visually. While it is possible to modify the automatically generated models, it is not considered to be a good modeling practice, since the automatic transformation works in one direction only. There is no support provided to modify the original models automatically based on the manual changes of the models generated by the MTT.

Modeling Aspects

The objective of this research is to automatically synthesize large-scale instrumentation systems running on distributed memory multiprocessors with flexible interconnection topology. What do the system models need to contain to achieve this goal? There are two main aspects of the problem: the software and the hardware, i.e. the signal processing and other computations that need to be performed and the target hardware architecture. Consequently, one modeling aspect is assigned to each.

The signal flow graph is a widely accepted way of describing instrumentation/signal processing systems. The first modeling aspect, the *Signal Flow Aspect,* closely resembles a signal flow graph. The *Hardware Aspect* describes the available hardware resources and

their interconnection topology. These two aspects of the system are not independent. Different elements of the signal flow graph may have certain hardware resource requirements. They constitute assignment constraints that must be satisfied during system synthesis. The third and final aspect of the system models describe these resource requirements. It is called the *Assignment Constraints Aspect*.

Signal Flow Aspect

The signal flow models consist of predefined atomic and user-defined aggregate components. The model structuring concepts applied in this aspect are summarized in Table 1. The atomic components are listed in Table 2. The primitive and the compound are the aggregate model components of the Signal Flow Aspect (Table 3).

Table 1 Signal flow model structuring concepts

Structure	Description
Part-whole hierarchy	predefined atomic and user-defined aggregate components; aggregates contain atomic and/or aggregate components
Module interconnection	connections between selected type of atomic components and selected parts of aggregate components
Generative modeling	textual description of model components and structure

The primitive is the lowest level computational block. It contains atomic objects only. It does not have any connections. It has different textual and numerical attributes associated with it, such as a script, a priority, etc. In addition to atomic objects, the compound contains user-defined components, i.e. previously defined primitives and compounds. Compounds containing compounds create the model hierarchy.

Table 2 Signal flow model atomic components

Components	Description
Input signal	input data interface of aggregate components
Output signal	output data interface of aggregate components
Local signal	data queuing, merging, splitting; corresponds to datanodes in the MCM
Local parameter	static parameter for aggregate components; corresponds to contexts in the MCM
Input parameter	parameter interface of aggregate components
Condition	condition specification

Table 3 Signal flow model aggregate components

Components	Description
Primitive	elementary computational unit; only atomic parts; no local signal or condition part; no connections; corresponds to actors in the MCM
Compound	aggregate (primitive and/or compound) and atomic parts; signal and parameter connections

The signal flow model of a system must have exactly one top level compound model containing every lower level model. This hierarchy is best described by a tree whose nodes are the compound and primitive models. The children of a node are the aggregate models it contains. The root of the tree is the top level compound model. The leaves are the primitive models (Figure 6).

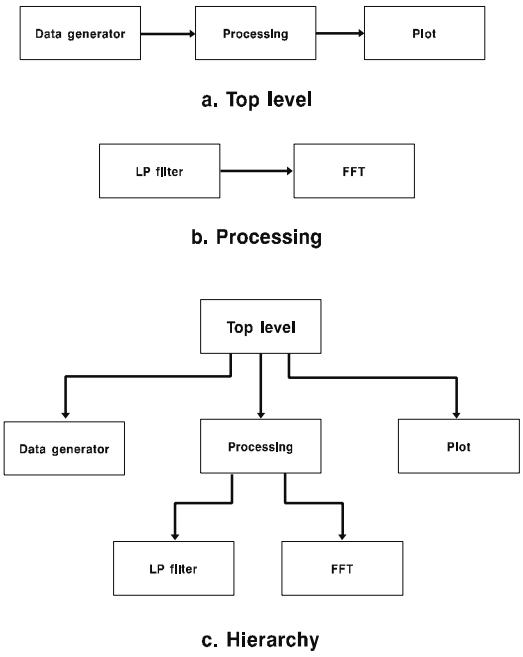

Figure 6 Model hierarchy example

Primitive Model

Primitive models correspond to actornodes in the MCM. The most important attribute a primitive model has is the script. The script is a subroutine written in a procedural or functional language that is executed every time the actornode is fired. The script attribute contains the name of the subroutine and the object or library file name where it is located. A related attribute is the estimated execution time. This is needed by the system synthesizer to ensure good processor allocation.

The atomic objects that primitives can contain are the input and output signals, and the input and local parameters. The input and output signals constitute the data interface

of the primitive. They correspond to actornode ports in the Multigraph Computational Model (MCM). The Multigraph Kernel (MGK) provides a set of functions to access these "data ports" to receive or propagate data at runtime. At higher levels of the model hierarchy, the icons corresponding to the input and output signals are connected to create the signal flow graph. The attributes of input and output signals include data rates. These are needed by the system synthesizer to ensure good allocation of communication resources.

The local and input parameters of the primitive model are used to assemble the actornode context. Local parameters have data types and values specified by the user. Input parameters are used to propagate the value of a local parameter specified at a higher level of the model hierarchy down to the primitive model. The local and input parameters can have simple data types, e.g. integers or doubles, or pointers to more complicated, user-defined types.

The primitive model has an optional attribute, the init-script. This is a subroutine that is called before the execution of the dataflow graph starts. It can be used to initialize the context of the actornode, or to reset a hardware device, etc. Another optional attribute is the so-called secondary script. It supports the real-time actornode construct of the MCM. Further attributes include static priority and firing condition. These are used by the MGK for scheduling. The attribute editor for primitive models is shown in Figure 7.

Note that at this level, i.e. inside the primitive model, there are no connections. The primitives are the elementary building blocks of the signal flow graph. All the connections are at higher levels of the hierarchy. The input signals and the input and local parameters are simply consumed; the output signals are generated by the script of the primitive.

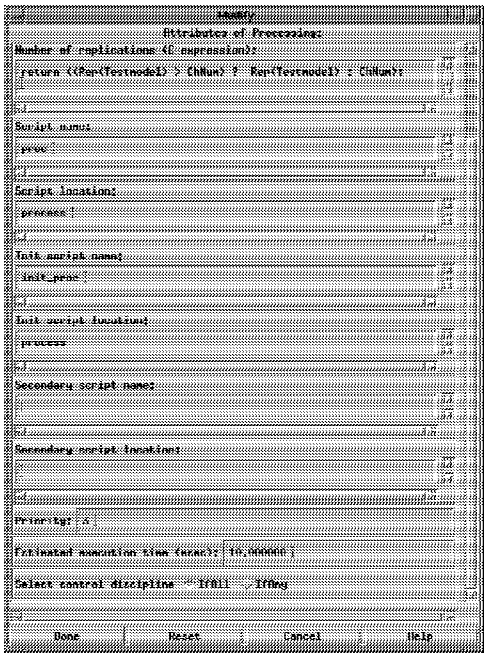

Figure 7 Primitive model attributes

Compound Model

Compounds may contain primitives, compounds, and atomic objects. These atomic objects can be input, output and local signals, input and local parameters, and conditions (Table 2). Local signals correspond to datanodes in the MCM. Their attributes include data type and buffer length. Input and output signals describe the data interface of the compound model. Icons representing primitive and compound components have ports for their input and output signals (and for their input parameters). The signal flow is modeled by connections between selected types of atomic components and ports of aggregate components (primitives and compounds). The valid signal connections are listed in Table 4. These connection constraints ensure that the resulting dataflow graph is bipartite as required by the MCM: actornodes are connected to datanodes and vice versa.

Table 4 Signal flow model signal connections

From	To
input signal	input port of a component
output port of a component	output signal
local signal	input port of a component
output port of a component	local signal
output port of a component	input port of a component

Table 5 Signal flow model parameter connections

From	To
local parameter	input parameter port of a component
input parameter	input parameter port of a component

Ultimately, connections are between primitives. Because of the model hierarchy, however, intermediate connections are needed. Input signals and input ports of compounds are used to propagate data down the hierarchy to the input port of a primitive model. Similarly, output signals and output ports of compounds are used to propagate data up the hierarchy. An output port can be directly connected to an input port, or a local signal can be inserted. The latter method is necessary when the default queuing behavior is not acceptable for the given data connection, or merging or splitting of data streams are required.

Figure 8 shows the top level compound model in the model editor (and the model browser) of the example depicted in Figure 6.

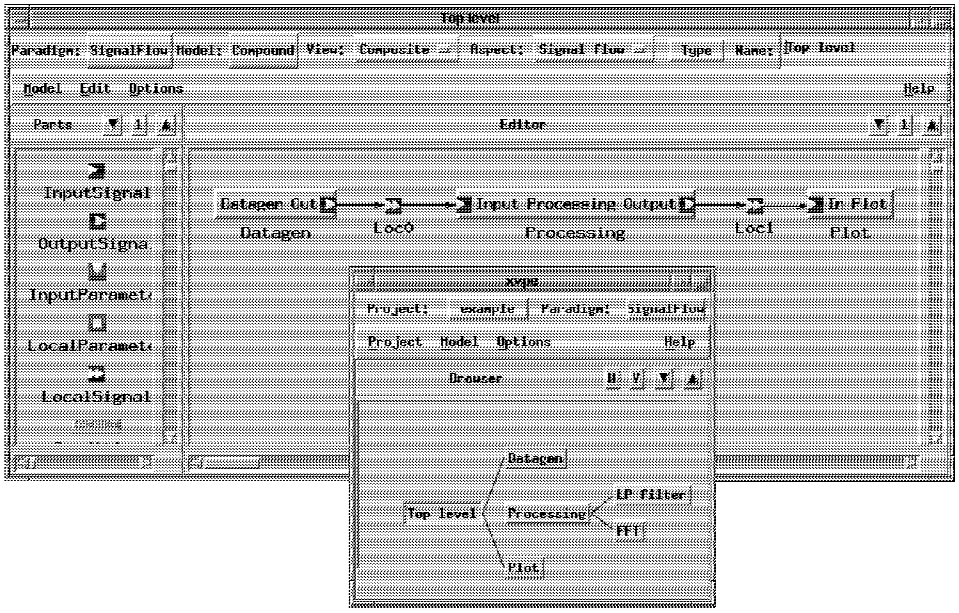

Figure 8 Signal flow model example

The value of each local parameter needs to be propagated down all the way to the primitive, where it becomes a parameter for the script as part of its context. This is modeled by connections between local and input parameters (Table 5). As an example, consider a primitive model corresponding to a simple amplifier actornode. It has one input signal for the input data, one input parameter for the gain, and one output signal for the output data. Several instances of this model can be used in different compound models. For each instance, a local parameter needs to be defined and connected to the input parameter of the amplifier primitive. The value of this local parameter is the gain, which can be different for each instance of the amplifier (Figure 9).

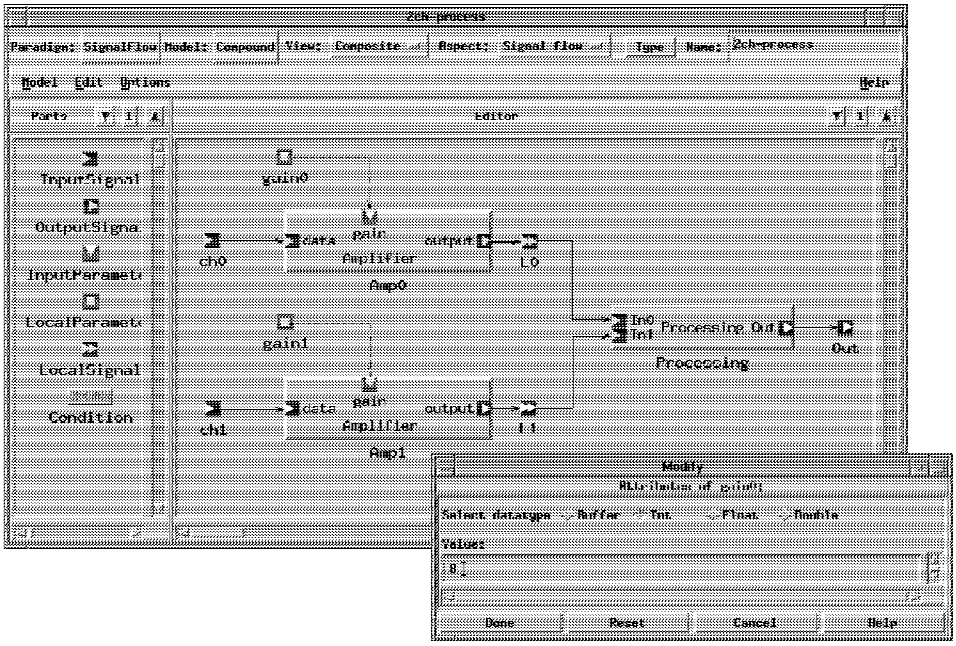

Figure 9 Local parameter example

Generative Attributes

Generative modeling is supported through generative attributes in the MIPAS. In the signal flow aspect, every primitive, compound, input, output and local signal, and input and local parameter has a repetition textual attribute. This is specified as a C++ function body that returns an integer, the number of repetitions of the model component. The repetition generative attribute defaults to "return 1;".

Model connections can be specified in the structure attribute. Only compound models have inner structure, therefore, only they have this attribute. The structure attribute is specified as a C++ function body. In the code, components of the current model can be accessed by name. Connections can be created or destroyed by calling the predefined functions Connect(a,b) and Disconnect(a). These are overloaded C++ functions. They

allow parameter type combinations for all legal connection types. For example, input signal to input signal port, output signal port to local signal etc. Other predefined functions include Rep(modelname) that returns the actual number of repetitions of the specified model, and Connected(a) that returns the object connected to the specified part allowing indirect references to models. The available predefined functions are listed in Table 6.

Note that the replication of a model type definition is meaningless. Therefore, the repetition attributes of model types are ignored by the model interpreters.

Compound models can contain one or more conditions. Conditions are atomic objects. They contain a user specified numerical value. This value can be accessed by name in the repetition and structure attributes. Generative modeling and conditions provide a very flexible and powerful modeling technique.

Table 6 Predefined functions for signal flow generative modeling

Function	Description
Connect(a,b)	connects atomic parts and/or ports a and b
Disconnect(a)	deletes existing connections of atomic part or port a
Disconnect_all(m)	deletes every connection of model m
Connected(a)	returns the atomic part or port connected to a
Input_signal(m,i)	returns the #i input port of model m
Output_signal(m,i)	returns the #i output port of model m
Input_parameter(m,i)	returns the #i input parameter of model m
Rep(m)	returns the number of repetitions of model m

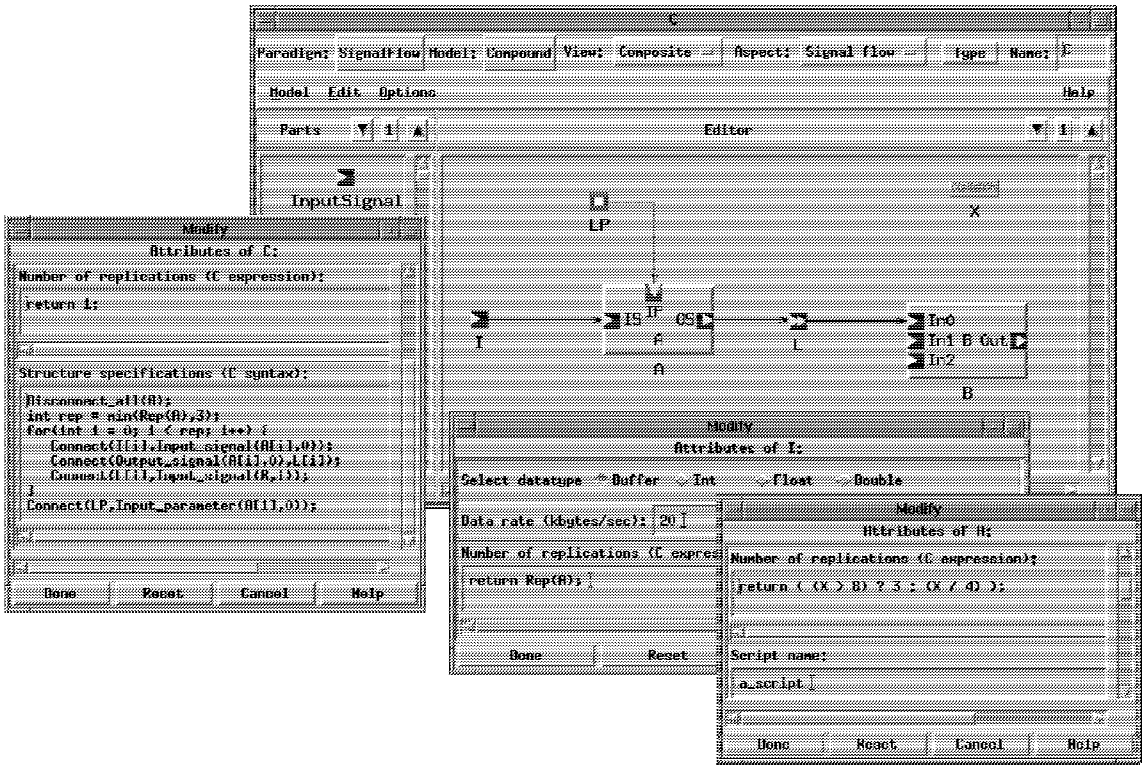

Figure 10 Generative modeling example

Figure 10 shows a simple example of the technique. The repetition attribute of model **A** depends on condition **X** and evaluates to 3. The repetition attributes of input signal **I** and local signal **L** depend on the number of repetitions of **A**. They also evaluate to 3. By convention, the graphically specified connections always correspond to the first instance of the model. The structure specifications of compound **C** start with deleting all graphical connections (Figure 10). The only purpose for this is to control every connection from the code. The next line is for error recovery. If somehow the number of repetitions specified for model **A** is greater than 3, still no connection is made to more than 3 of model **B**'s

input ports, since it has only 3. Next the appropriate input signals are connected to the corresponding input ports of model **A**, the output ports of model **A** to the local signals, and finally, the local signals to the input ports of model **B**. The last line specifies the connection between local parameter **LP** and the input parameters of the second instance of model **A**. The generated model is shown in Figure 11.

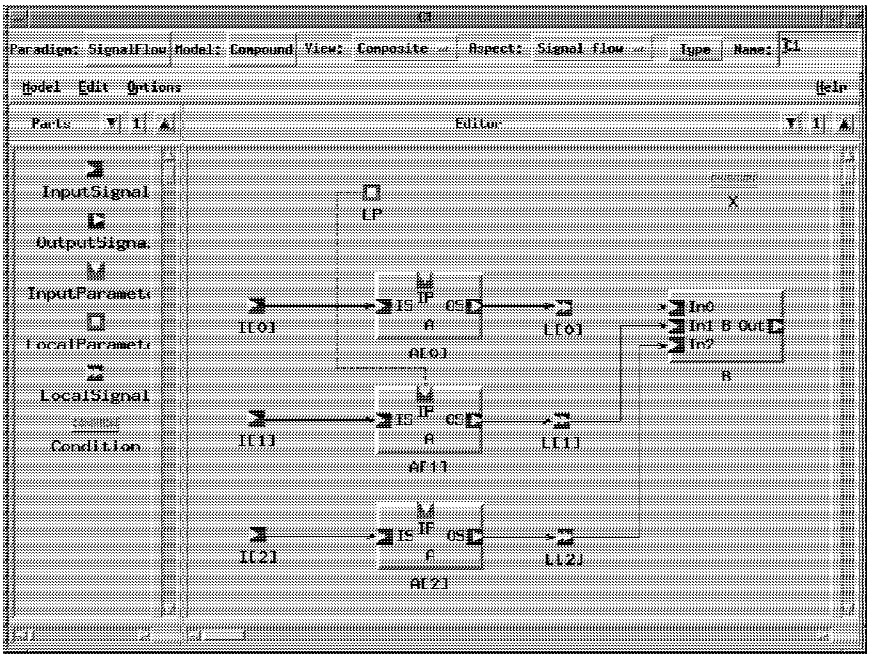

Figure 11 Automatically generated models

Hardware Aspect

The MIPAS target hardware platforms are distributed memory multiprocessors with flexible interconnection topology, such as TMS320C40 or T9000 networks. The key information the models need to capture are the topology of the network and the available resources. Tables 7 through 9 list the components and structuring concepts applied in the hardware aspect.

Table 7 Hardware model structuring concepts

Structure	Description
Part-whole hierarchy	predefined atomic and user-defined aggregate components; aggregates contain atomic and/or aggregate components
Module interconnection	connections between selected type of atomic components and selected parts of aggregate components
Generative modeling	textual description of model components and structure

Table 8 Hardware model atomic components

Components	Description
Communication link	interface between aggregate components
Resource	capabilities of aggregate components

Table 9 Hardware model aggregate components

Components	Description
Node	elementary aggregate component; only atomic parts; corresponds to processors
Network	aggregate (node and/or network) and atomic parts

The hierarchy of the hardware aspect is organized in a manner similar to that of the signal flow. The concept of the two user-defined components, the node and the network, is similar to that of the primitive and the compound. A node can have only atomic parts, e.g. communication links, while a network can have node and network components as well. Node models correspond to processors, while network models describe uni- or multiprocessor boards, subsystems, systems, etc.

The communication links of nodes have maximum data rate attributes specifying their speed. The nodes have attributes specifying their performance. These are needed by the system synthesizer for resource allocation.

Networks can have node, network, as well as communication link components. As all connections in the physical network are between processors, the communication link parts of networks serve simply as tools to propagate connections up and then down the hierarchy from one node to another. The valid connections are listed in Table 10. Figure 12 shows a top level network model with a host computer (**Host**, node model) and several boards (network models).

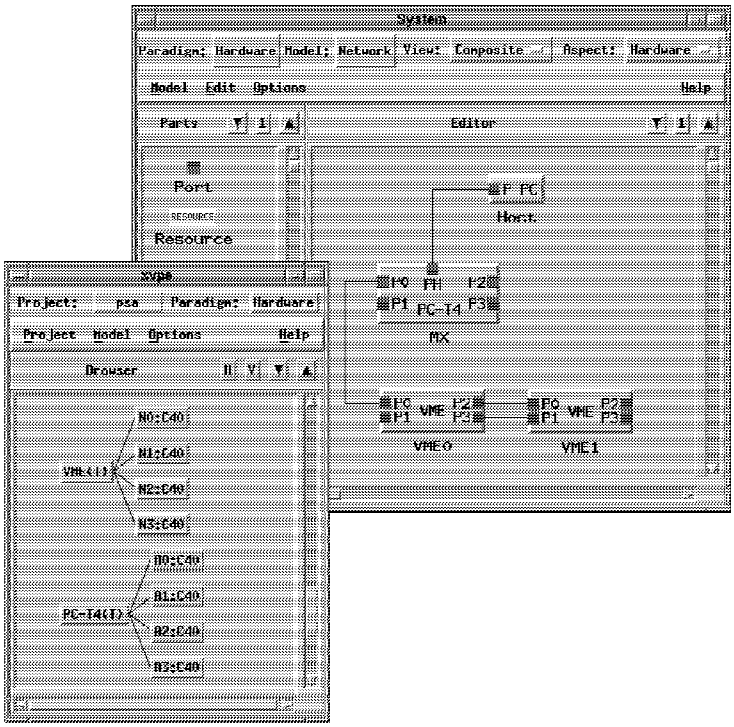

Figure 12 Hardware model example

Table 10 Hardware model connections

From	To
communication link	communication link port of component
communication link port of component	communication link
communication link port of component	communication link port of component

Nodes and networks can have resources attached to them. Resources are atomic objects. Their only attributes are their names. They represent special capabilities of nodes or networks, for example, such devices as A/D converters, disks, or printers attached to them. They can be used in the assignment constraints aspect to express resource requirements of different blocks of the signal flow model.

Generative modeling in the hardware aspect is similar to that in the signal flow aspect. Nodes, networks, and ports have the repetition attribute. Networks have the structure attribute as well. The predefined functions accessible in the generative attributes are listed in Table 11.

Table 11 Predefined functions for hardware generative modeling

Function	Description
Connect(a,b)	connects atomic parts and/or ports a and b
Disconnect(a)	deletes existing connections of atomic part or port a
Disconnect_all(m)	deletes every connection of model m
Connected(a)	returns the atomic part or port connected to a
Link(m,i)	returns the #i communication link of model m
Rep(m)	returns the number of repetitions of model m

Assignment Constraints Aspect

The Assignment Constraints Aspect constitutes a two-level hierarchy. The top level model, called configuration, contains a set of lower level models, called rules and the bans (Tables 12-13). Rules specify positive assignment constraints, e.g. that a given signal flow module must be assigned to a given hardware module. Bans specify negative assignment constraints, e.g. that a given signal flow module must not be assigned to a given hardware module.

Table 12 Assignment constraints model structuring concepts

Structure	Description
Two-level part-whole hierarchy	1st level aggregates contain atomic parts; 2nd level aggregate contains 1st level aggregates
Generative modeling	Textual specifications of references to generative model components in different aspects

Table 13 Assignment constraints model atomic components

Components	Description
Reference to signal flow model aggregates	group of signal flow components
Reference to hardware model aggregates	group of hardware components
Resource requirements	capability requirements

Rules and bans can have references to user-defined signal flow model components and to user-defined hardware model components. Rules can have resource requirement parts.

Resource requirements are atomic objects, they represent special needs of the signal flow components. A model component can be referenced as a type or as an instance. Assigning a specific computation to a specific processor requires instance references. But assigning a type of computation, e.g. a generic FFT primitive, to a class of nodes, e.g. digital signal processors, requires type references.

Table 14 Assignment constraints model aggregate components

Components	Description
Rule	one positive assignment constraint; only atomic parts
Ban	one negative assignment constraint; only atomic parts; no resource requirement part
Configuration	set of all assignment constraints; only rule and ban parts

A rule must contain one or more signal flow references, and can contain one or more hardware references (Table 14, Figure 13). The signal flow references of a rule mean that the run-time objects corresponding to the (type of) signal flow components must be assigned to the same node. If the rule contains a hardware reference that means that the run-time objects must be assigned to one of the processors specified by the reference. If there are more then one hardware references, then the assignment can be made to any one of them. If the rule contains resources requirements, that means that the assignment must be made to a node containing all the specified resources or to a node specified by a hardware reference.

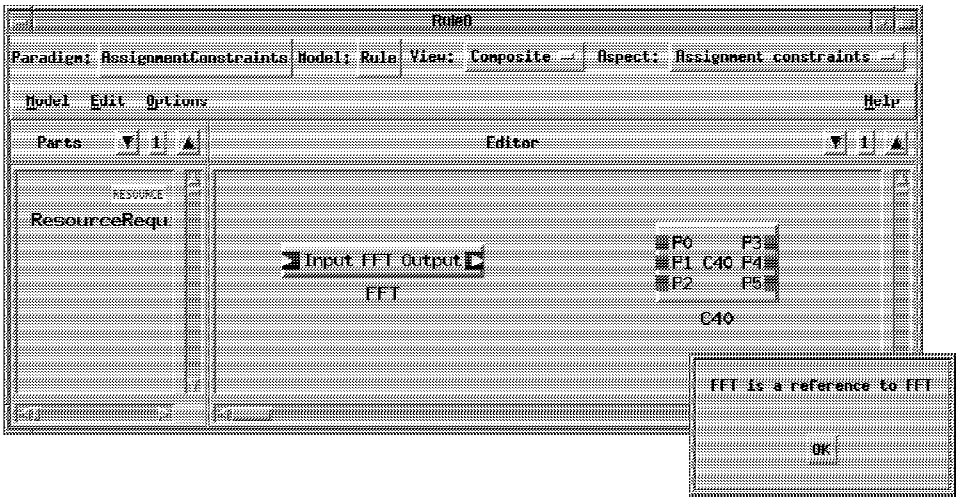

Figure 13 Assignment constraints example

These relations can be expressed by the logical expression:

$$[SF_1 \wedge ... \wedge SF_n] \rightarrow [HW_1 \vee ... \vee HW_m \vee HWnode\{RS_1 \wedge ... \wedge RS_l\}] \quad (8)$$

meaning that run-time objects corresponding to SF_i signal flow component references (type or instance) must be assigned together to one of the processors corresponding to HW_j hardware model component references (type or instance) or any of the processors having every RS_k resource. The hardware and/or the resource component part can be omitted, but at least one signal flow component must be present.

A ban must contain one or more signal flow references, and one or more hardware references (Table 14). The signal flow references of a rule mean that none of the run-time objects corresponding to the (type of) signal flow components may be assigned to any of the processors specified by the hardware references.

These relations can be expressed by the logical expression:

$$\sim\{[SF_1 \vee ... \vee SF_n] \rightarrow [HW_1 \vee ... \vee HW_m]\} \quad (9)$$

meaning that none of the run-time objects corresponding to SF_i signal flow component references (type or instance) may be assigned to any of the processors corresponding to HW_j hardware model component references (type or instance).

Reference attribute

Every rule and ban has a reference generative attribute. It is used to reference generative model components of other aspects. The reference attribute specializes the meaning of the graphically generated references. It can either modify the current rule (ban) or create a new one.

The reference attribute contains a C++ function body. The predefined function **Refer(ref,i)** is used to specify the instance of a model the graphical reference points to. Here **ref** is the name of the reference and **i** is an integer. For instance, if a rule contains a reference **FFT** to a signal flow compound **FFT** and the repetition attribute of this model specifies 5 instances, then the reference attribute **Refer(FFT,3)** specifies that the **FFT** reference points to the fourth instance of the **FFT** compound (C convention). A single reference can have several instantiations by multiple **Refer(ref,i)** calls.

However, if a separate rule is necessary to specify, for example, that instance #i of the **FFT** compound must (not) run on node **C40** #i, then the **Rule** and **Rule_end** (**Ban**, **Ban_end**) macros can be used. Figure 14 illustrates the concept through an example.

The for loop is executed as many times as there are **FFT** compounds. The **Model(ref)** function is called to return the model referenced by **ref**. **Rep(m)** returns the number of repetitions of the model. Each call to the **Rule** macro generates a new rule. For i=0, for example, the new rule states that the first instance of the signal flow compound **FFT** must

be assigned to the first instance of the hardware node **C40**.

This example assumes that the value the repetition attribute of the **C40** model evaluates to is greater than or equal to the value of the repetition attribute of the **FFT** compound. Otherwise, the assignment constraint specification are erroneous. Such errors are detected by the model interpreters. Table 15 lists the predefined macros and functions available in the reference attribute.

```
int i;
for( i = 0; i < Rep(Model(FFT)); i++) {
        Rule {
                Refer(FFT,i);
                Refer(C40,i);
        } Rule_end;
}
```

Figure 14 Reference attribute example

Table 15 Predefined functions for assignment constraints generative modeling

Function	Description
Refer(ref,i)	specifies that model instance #i is referenced by ref
Rule	macro; starts a rule definition
Rule_end	macro; ends a rule definition
Ban	macro; starts a ban definition
Ban_end	macro; ends a ban definition
Model(ref)	returns the model referenced by ref
Rep(m)	returns the number of repetitions of model m

Note that only instance references are allowed in the reference attribute, since repetition of a model type definition is not meaningful.

Configuration

The top level assignment constraints model, the configuration, is a collection of rules and bans.

CHAPTER V

MODEL INTERPRETATION AND ANALYSIS

Model Interpretation

The task of the model interpretation in the Multigraph Architecture is to synthesize applications from the domain models and run-time libraries, and to produce input to various system engineering tools. In the case of the parallel signal processing domain, there are two different model interpreters and a model analysis tool responsible for distinct tasks. Figure 15 illustrates their location in the overall structure of the Model-Integrated Parallel Application Synthesizer (MIPAS).

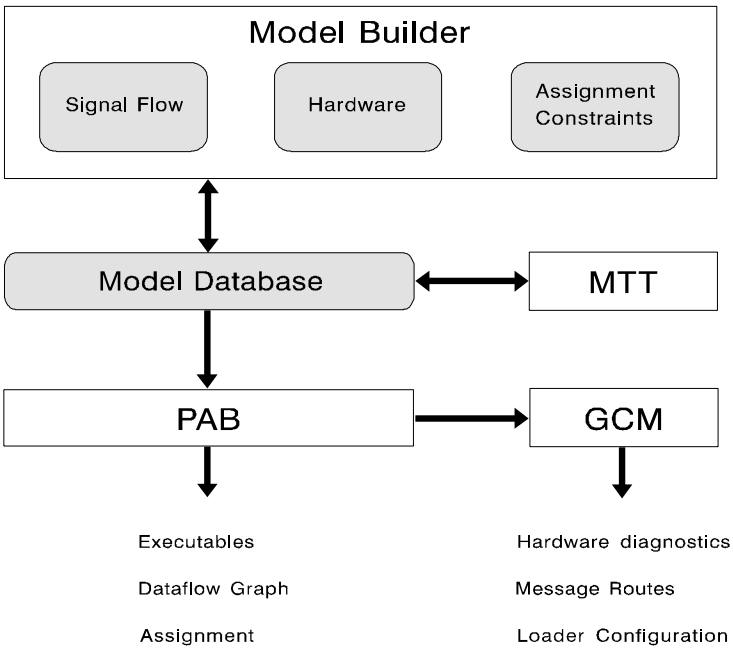

Figure 15 Model-Integrated Parallel Application Synthesis

The Model Transformation Tool

The task of the Model Transformation Tool (MTT) is to visualize generative models to help debug system models. The models in the MIPAS are mixed declarative (graphical) and generative (textual). It is relatively easy to make a mistake, which can go undetected in the modeling phase, because of this double paradigm. The MTT takes the system models, evaluates the generative attributes of the model components, and generates a new set of models that are purely declarative (graphical) and, therefore, easier to debug.

The MTT has two main parts. The first one evaluates the models and creates the second part, a C++ program, automatically. It generates data structures based on the models, and writes wrappers around the function bodies specified by the user as generative attributes. The generated program contains the predefined functions for each aspect and a main function. The main function creates the appropriate number of replications for each model component by calling the functions created from the repetition attributes, makes the additional connections using the functions generated from the structure attributes, and creates the references specified by the reference attributes. This second part lays out the model components and routes the connections automatically for the GUI of the model builder. The generated C++ program is compiled, linked, and executed. Syntax errors in the user-specified attributes cause compiler or linker errors. Since these errors are not easy to trace back to the models, a tool is needed to locate the original errors in the model specifications automatically.

The Parallel Application Builder

The primary model interpreter in the MIPAS is the Parallel Application Builder (PAB). It is responsible for: (1) creating the macro dataflow graph corresponding to the signal flow models, (2) partitioning the graph, (3) assigning the partitions to the nodes of the processor network specified in the hardware models while satisfying the requirements specified in the assignment constraint models, (4) creating the executables for the nodes, and (5) providing the hardware description and communication information to the Graphical Configuration Manager (GCM), the model analysis tool responsible for hardware diagnostics and model verification, message routing, and network loader configuration (Figure 15).

The PAB first creates a signal flow builder object network corresponding to the signal flow models residing in the model database. There is a builder object corresponding to every model object, including compounds, primitives, signals, parameters, and conditions. The generative attributes are evaluated to create the currently required number of builder objects. Either the original models, or the models generated by the Model Transformation Tool (MTT) can be provided to the PAB. The resulting builder object network is a tree, the root of which is the top level model builder corresponding to the top level signal flow model. Figure 16 shows the builder tree for the example introduced in Figure 6 and Figure 8.

The PAB creates the connections specified in the generative attributes of the models. The program checks every model connection (signal and parameter) for datatype consistency and creates the appropriate connections in the builder network as well. All direct connections (between primitive input and output signals and local signals, and

between local parameters and primitive input parameters) corresponding to the connections in the dataflow graph are generated bypassing the hierarchy. Next the corresponding MCM objects are created: actornodes for primitives, datanodes for direct signal connections (or local signals), and contexts for parameters. Then the actornodes and datanodes are connected to form the dataflow graph.

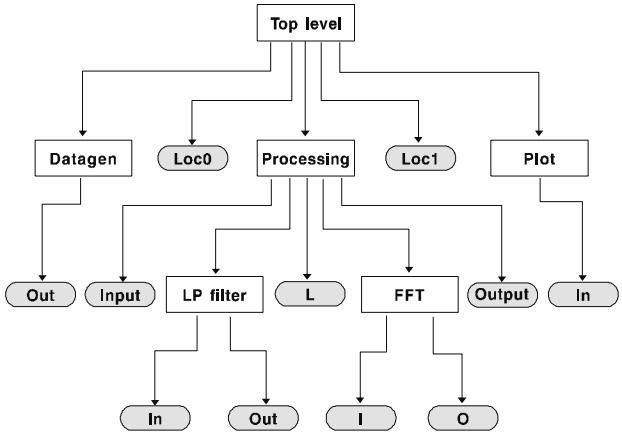

Figure 16 Signal flow builder object network

The next step is to make a builder network for the hardware models. This is performed similarly to the signal flow model interpretation: a builder tree is created, where the nodes are the network, node, communication link, and resource component builders. All the connections between model components and the direct node to node connections are generated as well.

The PAB evaluates the rules and bans of the assignment constraints configuration and provides each signal flow primitive builder with a list of hardware node builders the corresponding actor can be assigned to. Infeasible requirements are detected at this point.

Since the assignment problem is NP-complete, the PAB employs a heuristic procedure to partition the dataflow graph and assign the subgraphs to the hardware nodes. It utilizes the hierarchy of the signal flow models to cut the search space and guide the search. The assignment problem in the MIPAS is illustrated in Figure 17. Chapter VII is dedicated to the assignment problem. The whole approach, including the selected cost function and the algorithm assigning signal flow primitives to hardware nodes, is described in detail.

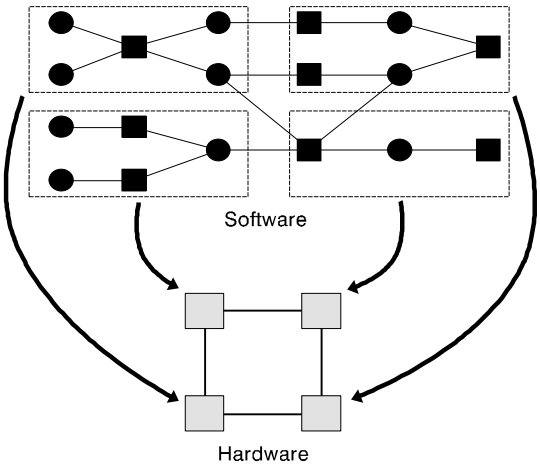

Figure 17 The assignment of the dataflow graph

An alternative to the traditional assignment procedure, where one assigns processes to a preconfigured hardware platform, is hardware topology synthesis. The input to this procedure is the signal flow graph and a set of nodes. Hardware topology synthesis creates the processor interconnection network. It tries to match the topology of it to that of the signal flow graph. This problem is NP-complete as well, therefore, a heuristic approach must be used. The PAB implements hardware topology synthesis utilizing the

hierarchy of the signal flow models as a user-defined heuristic. The approach is described in Chapter VII.

Once the assignment is completed, the PAB generates a makefile to link the appropriate MGK and the required object and library files to create the executable for each processor. Then the make utility is executed.

At this point, the user has the option to save the builder networks into the database of the PAB. This is useful when the application needs to be executed more than once without changing the system models, since considerable time can be saved by not performing the building and assignment related tasks. The possibility of performing the model interpretation itself in parallel in order to speed it up is examined in [26].

The last step is to create the environments and tasks for the MGK according to the assignment. Before activating the dataflow graph and starting the execution of the application, message route generation and network loader configuration needs to be performed. These tasks are carried out by a model analysis tool, the Graphical Configuration Manager.

Model Analysis

The PAB generates input information required by the Graphical Configuration Manager (GCM), a model analysis tool. The GCM is capable of comparing the actual hardware configuration to the hardware models, generating network loader configuration files and deadlock-free message routing for wormhole and store-and-forward routers [35]. Figure 18 shows the same processor network loaded into the GCM as was shown in Figure 12 in the GMB. Notice that the hierarchy of the hardware models has been

unrolled. GCM utilizes a flat processor network model because it suits its tasks the best.

The purpose of comparing the hardware models to the actual processor network is twofold. First, it validates the models and second, it diagnoses the hardware itself. The GCM uses a worm program (the standard INMOS check program for transputers, and TICK, the worm/loader/debugger for TMS320C40 networks developed at Vanderbilt [7]) to explore the network. The GCM is capable of detecting missing (or broken) processors, missing (or broken) communication links, and swapped links as well.

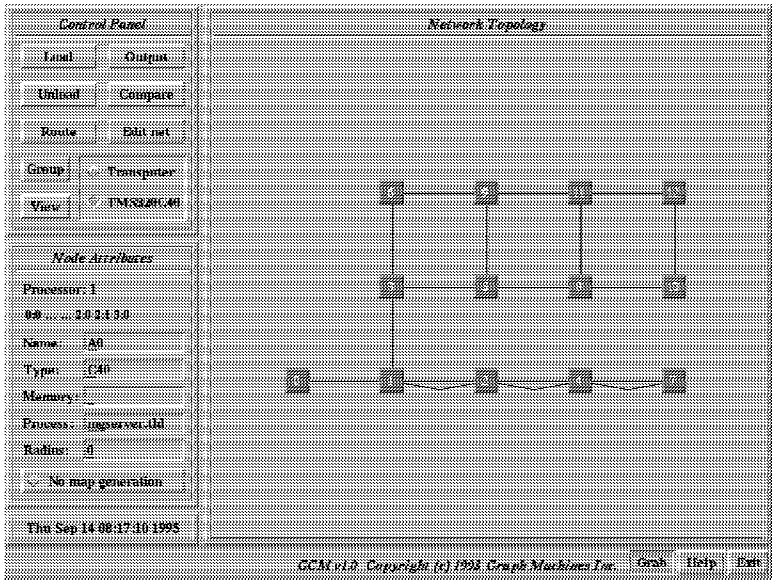

Figure 18 The Graphical Configuration Manager

The output of the worm is transformed into a graph similar to the processor network model. The two graphs are then compared. This task is not equivalent to the NP-complete general graph isomorphism problem [27] for two reasons. First, the mapping between two nodes is known beforehand, since the host computer is fixed. Second, all graph edges have associated port numbers. To test isomorphism, a simple breadth-first search can

provide a yes/no answer. However, if the two graphs are not isomorphic, the type and exact location of their differences are not straightforward to find.

The applied algorithm is able to locate the three most common types of errors: missing nodes, missing connections, and swapped connections. First, it decides whether the physical network is a subgraph of the modeled one. If so, it locates any missing nodes and connections. If not, it repeatedly transforms the graph corresponding to the physical network by swapping selected links until it becomes a valid subgraph of the model graph. If a transformation is not possible, the procedure is aborted. Otherwise, a detailed report is generated (Figure 19).

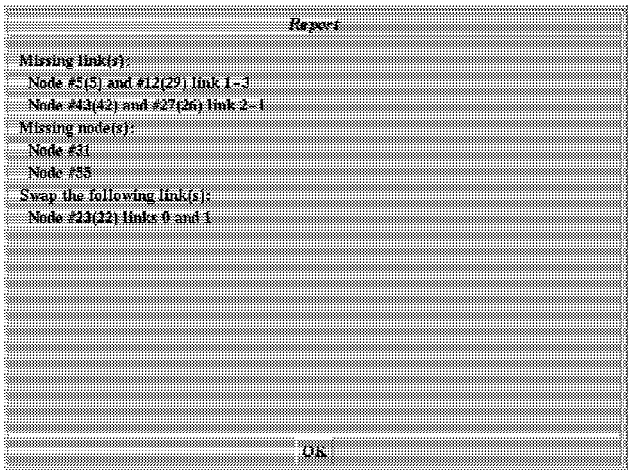

Figure 19 The results of network comparison

Notice that an error message, such as "Missing node #31", is much more meaningful with the GCM than when manually analyzing the output of a worm program. In general, node numbers are meaningless; they are assigned incrementally as the worm finds the processors. A small change in the topology can alter most of them. GCM supports node

attributes along with numerical identifiers. Since they include unique node names extracted from the models by the PAB, no additional time is needed to locate the errors. Another important task of the GCM is network loader configuration.

Network loaders require information on the processor interconnection network and the processes of the application in the form of a configuration file. Each loader has its own requirements and configuration language. A configuration file is typically big and complicated. The GCM is capable of generating configuration files for four different loaders: the Logical Systems and the Inmos loader for transputers, the 3L environment for transputers and C40s, and the C40 loader TICK [7]. The most important task of the GCM is deadlock-free message route generation.

Deadlock avoidance with store-and-forward and virtual cut-through routing is simple with a careful message buffer allocation strategy. Wormhole routing, the most efficient message routing method, is, however, deadlock-prone. The two known deadlock avoidance methods cannot be utilized. Virtual channels require hardware support not available on the MIPAS target platforms. Topology-based deadlock avoidance is too restrictive for the purposes of the MIPAS.

Partially connected message routing can be used in the MIPAS because the PAB provides not only the description of the hardware, but also the list of processors that need to communicate with each other. Chapter VI is dedicated to message routing in the MIPAS. It is shown that partially connected minimal deadlock-free routing is NP-complete. Consequently, a non-minimal approach must be used. A partially connected, non-minimal message routing strategy, that guarantees deadlock-freedom and provides comparable, in many cases smaller, communication overhead than minimal routing

strategies, is introduced and evaluated in Chapter VI. The GCM incorporates this algorithm to generate deadlock-free message routing for wormhole routed networks.

Another system engineering tool can be used to predict the performance of the generated application. The program loads the system description generated by the PAB and produces a Generalized Stochastic Petri Net, which is solved for various performance metrics, such as processor utilization and application response time [17].

CHAPTER VI

MESSAGE ROUTING

Wormhole routing is the most efficient message routing technique [42]. There are two different methods for deadlock avoidance in wormhole routed networks: virtual channels and topology-based deadlock avoidance. Virtual channels not only help in deadlock avoidance, but also in flow control. However, they require hardware support not available on the target platform of this work. Topology-based deadlock avoidance restricts the possible topologies to the known deadlock-free configurations, but requires no special hardware support.

The known deadlock-free configurations are hypercube-, mesh-, or tree-based networks. They are the only known direct networks with deadlock-free minimal routing strategies. The class of deadlock-free, indirect network topologies is wider. This is only possible because indirect networks have two types of nodes: processing and switching. Processing nodes do not route messages, and switching nodes do not send or receive them. This fact simplifies routing. This dissertation focuses on direct networks because of the target hardware platform.

There are very few known deadlock-free, direct network topologies. Since one of the biggest advantages of the target hardware platform is its flexibility, restricting the interconnection network to these topologies is not acceptable. In the next section, a wider class of graphs is identified as inherently deadlock-free. Any minimal routing is shown to be deadlock-free if and only if the network is chordal. While chordal graphs constitute

a much wider class of deadlock-free topologies than was known before, they still do not provide enough flexibility. Therefore, other means of deadlock avoidance need to be provided.

Since all communication requirements of the application are captured in the system models, connected routing is not required. However, as is shown in this chapter, partially connected minimal routing is an NP-complete problem. The only requirement that can be removed in order to achieve a solution is the minimality of the routing. This does not have any adverse effects on message routing performance for two reasons. First, message path length is a negligible factor in message latency with wormhole routing (see Chapter II). Second, a non-minimal routing strategy is not forced to use a shortest path, and is therefore able to decrease message contention in certain situations when minimal routing is not able to.

This dissertation introduces a non-minimal, partially connected routing algorithm that employs a novel approach for deadlock avoidance. It generates a deadlock-free routing in a chordal subgraph of the network. A message path from this routing is used only if there is no path with less cost in the whole network that does not result in a deadlocked configuration. The algorithm was compared to minimal routing algorithms which do not guarantee deadlock-freedom and have exponential time complexity. In the majority of the test cases, the new algorithm produced equal or better results.

The presentation of the work takes the following path. First, topology-based deadlock avoidance is examined with special emphasis on chordal graphs. Next, the partially connected minimal routing problem is classified as NP-complete. Finally, a non-minimal routing algorithm is developed and analyzed.

Note that throughout the chapter message routing refers to deterministic message routing. Adaptive routing may provide a better solution, but one needs to develop results for deterministic routing before applying them to adaptive routing.

Topology-Based Deadlock Avoidance

Dally and Seitz provide the necessary and sufficient condition for deadlock-free routing in [18]. They use the following notation. The interconnection network I = G(N,C) is a directed graph. The vertices **N** represent the processing nodes, and the edges **C** the communication channels. A routing is a function R: C×N $\rightarrow$ C. It maps the current channel and the destination node to the next channel. The channel dependency graph **CDG** for a given interconnection network **I** and routing function R is a directed graph. The vertices of CDG are the channels of I. The edges of CDG are the pairs of channels connected by R. A routing function R for an interconnection network I is deadlock-free if and only if there are no cycles in the channel dependency graph CDG [18].

The authors assume unidirectional communication channels. However, it is easy to generalize their results for bidirectional links. For every communication channel, two nodes need to be added to the channel dependency graph, one for each direction. Furthermore, the routing function needs to be restricted, so that a message cannot be routed back to the node from which it has just arrived. Formally,

$$R(c_{ij},n_d) \neq c_{ji} \tag{10}$$

where c_{kl} denotes the edge from n_k to n_l. This causes the interconnection network I to be undirected. The channel dependency graph CDG is still directed. There are more generalizations possible, as described below.

Multiple links between two nodes must be allowed. Furthermore, the routing must not be restricted unnecessarily. The routing function R assigns an outgoing channel to an incoming channel destination node pair. Consider the situation depicted in Figure 20. Nodes **A** and **B** send messages to node **G**. They reach **D** on the same incoming channel. Their destination nodes are also the same. The routing function R forces the messages of both **A** and **B** to be routed to the same outgoing channel. However, it would be clearly better to route messages to different channels. For example, messages from **A** could be routed through node **E** and messages from **B** through node **F**. Therefore, a modified routing function $R:N \times C \times N \rightarrow C$ is used that assigns an outgoing channel to a source node, incoming channel, destination node trio. The channel dependency graph is constructed as before: if the routing function connects channel c_i to channel c_j, a directed edge is added between the corresponding nodes in the channel dependency graph. Dally's and Seitz's theorem and proof are still valid with these generalizations.

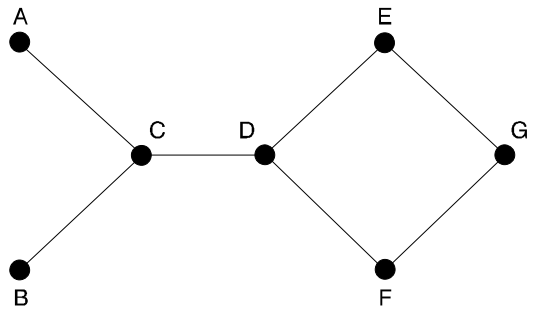

Figure 20 Routing example

Deadlock-freedom depends on the topology of the interconnection network and the selected routing function. For topology-based deadlock avoidance, the class of networks

with deadlock-free routing strategies must be identified. The next theorem states a trivial result.

Theorem I In a processor interconnection network I, any routing R is deadlock-free if and only if I is a tree.

Proof: The nodes of the channel dependency graph CDG are the edges of the network I. Neighboring nodes of a cycle in CDG are adjoining edges of I. Therefore, they form a cycle in I. There are no cycles possible in a tree. Hence, there are no cycles possible in CDG.

If I is not a tree, then it has at least one cycle. Consider that cycle. Since minimal routing is not required, the following routing strategy can be selected. From each node in the cycle to its left hand neighbor, messages are routed all around the cycle and not through their shared link. Each such route adds at least one edge in the channel dependency graph CDG. These edges form a cycle in CDG, which has the same size as the cycle in I. The routing is not deadlock-free. QED

If the routing is restricted to be minimal, the class of graphs with deadlock-free routing becomes wider:

Theorem II In a processor interconnection network I, any minimal routing R is deadlock-free if and only if I is chordal.

In other words, chordal graphs are the only inherently deadlock-free topologies. By

definition, chordal graphs are graphs with no chordless cycles. Chordal graphs are also called triangulated, because any cycle in a chordal graph must consist of triangles. Since trees are chordal, Theorem II does not contradict Theorem I.

Proof: Let I be a chordal interconnection network, R be a routing function, and CDG be the resulting channel dependency graph with at least one cycle. Consider a cycle in CDG. There are two cases. If the size of the cycle is greater than 3, then the corresponding cycle in I must have at least one chord forming a triangle with two neighboring edges in the cycle in I. Otherwise, I would have a chordless 4-cycle. In Figure 21, the nodes on the cycle are **A**, **B**, and **C** and the chord is **AC**. In CDG, the nodes corresponding to **AB** and **BC** have an edge connecting them. This edge is part of the cycle in CDG. This means that the routing function assigns the output channel **BC** to channel **AB**. In other words, it routes some messages from **A** to **C** through **B**, not through channel **AC**. Therefore, the routing is not minimal.

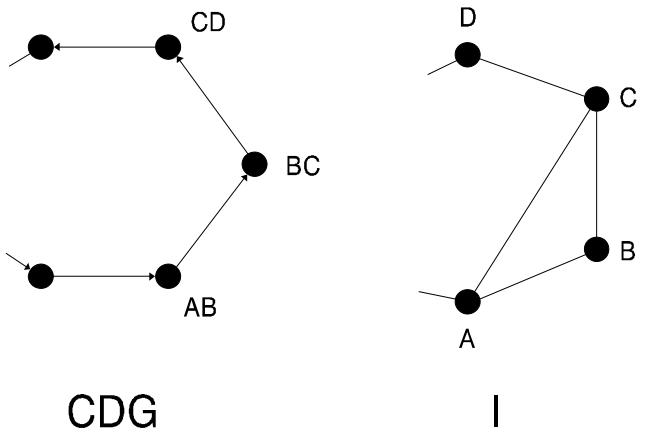

Figure 21 Chordal graph and corresponding CDG

In the second case, the size of the cycle in CDG is 3. The corresponding cycle in I is a triangle too. Consider the triangle **ABC** in Figure 21. The exact same argument applies here for this triangle: since **AB** and **BC** are connected in CDG, some messages from **A** to **C** are routed through **B**. Therefore, the routing is not minimal.

If I is not chordal, then it has at least one chordless cycle with size at least 4. Consider that cycle. The following routing strategy can be selected. From each node in the cycle to the second node on the right hand side, route to the right. Since the cycle is chordless and its size is at least 4, this is a shortest path. Each such route adds one edge in the channel dependency graph. These edges form a cycle in CDG. The routing is not deadlock-free. QED

Chordal graphs are the only inherently deadlock-free class of topologies. They are the only graphs in which every minimal routing is deadlock-free. Chordal graphs, however, form only a small subset of all possible topologies. Limiting the choice of topologies to these would be too restrictive. There are other topologies, i.e. meshes and hypercubes, that have minimal routing strategies. However, they tend to have a limited number of known deadlock-free message routing algorithms. That does not leave room for optimization. Consequently, other means of deadlock avoidance must be identified.

Partially Connected Routing

Partially connected, deadlock-free, minimal routing is possible even in networks where connected routing is not, depending on which nodes must communicate with each other. In the Model-Integrated Parallel Application Synthesizer (MIPAS), the system models and the automatically generated assignment determine the required message source-destination

node pairs. Therefore, partially connected routing can be used. However, the problem is NP-complete.

Theorem III Given a processor interconnection network I(N,C) and a set of message source-destination pairs $L = \{(n_i\ n_j) \mid n_i, n_j \in N, n_i \neq n_j\}$, identifying a deadlock-free, minimal, partially connected routing for I and L is NP-complete.

Proof: For each pair in L, there are a finite number of possible shortest paths. A nondeterministic algorithm needs to pick one for each and check in polynomial time whether the generated channel dependency graph is acyclic. Therefore, the problem is in NP.

The satisfiability problem can be transformed to the partially connected routing problem. The satisfiability problem is defined in [27] as follows:

Given a set of Boolean variables $V = \{ v_1, v_2 \ldots v_n \}$, a truth assignment t is a function $t:V \rightarrow \{True, False\}$. v_i is True if and only if $t(v_i) = True$. $\sim v_i$ is True if and only if $t(v_i) = False$. A clause over V is a set of literals representing their disjunction. A clause is satisfied by a truth assignment if and only if at least one of its literals is True. A collection C of clauses over V is satisfiable if and only if there exists some truth assignment for V that simultaneously satisfies all the clauses in C. Given a set V of Boolean variables and a set C of clauses finding a satisfying truth assignment is the satisfiability problem. Cook's Theorem states that the satisfiability problem is NP-complete [27].

Given a set V of Boolean variables and a set C of clauses, a corresponding processor

interconnection network I and a set L of message source-destination pairs can be constructed. For each variable v_i in V, construct a 6-cycle. Let the nodes be v_i1, v_i2, v_i3, v_i4, v_i5, v_i6 as shown in Figure 22. For each clause c_j in C, add two nodes c_j1 and c_j2. For each variable v_k in c_j, connect c_j1 and v_k6, and v_k4 and c_j2. For each variable negate $\sim v_p$ in c_j, connect c_j1 and v_p3, and v_p1 and cj2. See Figure 22. For each clause c_j, add (c_j1 c_j2) to the set of message source-destination node pairs. For each variable v_i, add (v_i2,v_i6), (v_i1,v_i5), (v_i5,v_i3) and (v_i4,v_i2), as well.

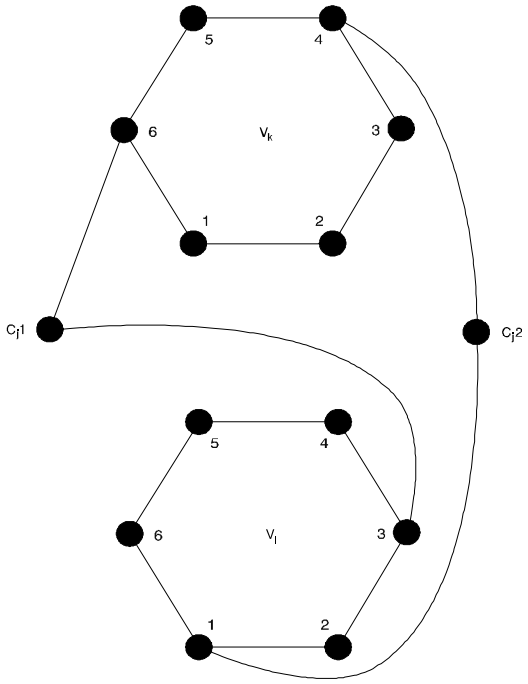

Figure 22 Transformation of the satisfiability problem

The partially connected routing problem for the resulting I and L is equivalent to the satisfiability problem of C on V. A deadlock-free minimal routing for I and L (if it exists) provides a truth assignment that satisfies C. For each clause, there exists a route from c_j1 to c_j2. It goes through v_i5 (v_i2), where v_i ($\sim v_i$) is one of the variables in the clause c_j. Assign True (False) to that variable. Since the routing is deadlock-free, there cannot be a route through v_i2 (v_i5). That would result in a cycle in the channel dependency graph, because of the four routes added for each variable in its corresponding 6-cycle. This means that if v_i is assigned True (False) because of clause c_j, then no other clause forces v_i to be False (True). Since there is a route for all c_j1, c_j2 pairs, a truth assignment that satisfies every clause has been found.

If there is no possible deadlock-free minimal routing for I and L, then C is not satisfiable. If a truth assignment that satisfied C exists, then a route from c_j1 to c_j2 can be selected based on that assignment. If v_i ($\sim v_i$) in c_j is True, this route is c_j1, v_i6, v_i5, v_i4, c_j2 (c_j1, v_i3, v_i2, v_i1, c_j2). No other clause forces v_i ($\sim v_i$) to be False, therefore, there is no route through v_i2 (v_i5). Consequently, there are no cycles in the channel dependency graph (because of the construction of I and L, the only cycles possible are the 6-cycles corresponding to variables). The routing is deadlock-free. Hence, C is not satisfiable. QED For an example of this transformation, see Figure 23.

The only requirement that can be removed in order to achieve a deadlock-free solution in polynomial time is the minimality of the routing. In the following section, a partially connected, non-minimal routing algorithm is developed that guarantees deadlock-freedom and minimizes message contention.

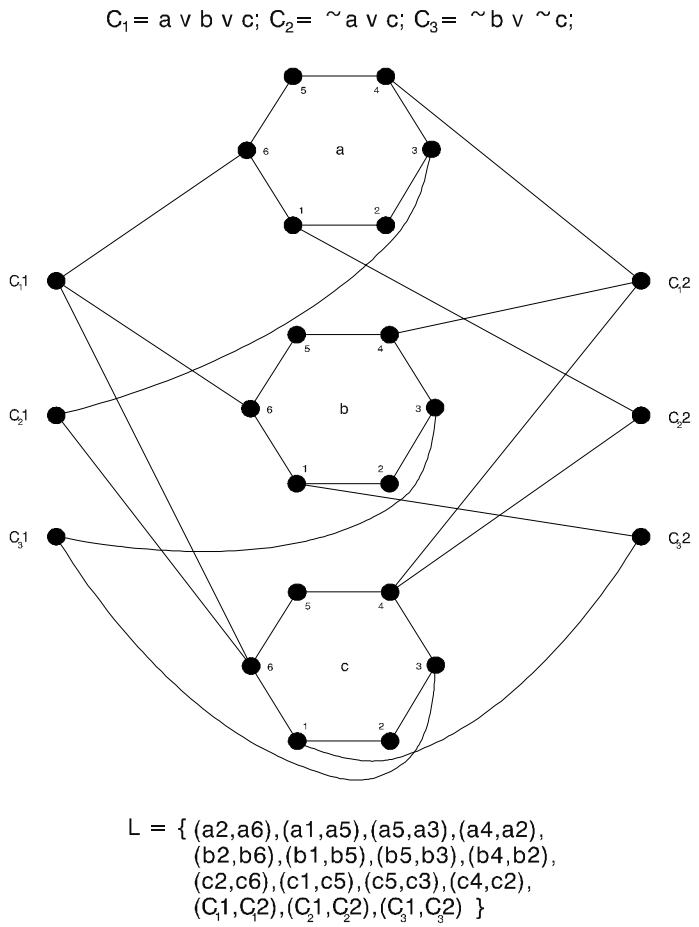

Figure 23 Example transformation

Non-Minimal Routing

Non-minimal, connected or partially connected, deadlock-free routing can be achieved in polynomial time. The simplest method is to route along a spanning tree of the network. This is highly inefficient, since only (n-1) links are used. (These networks usually have Kn/2 links, where n is the number of nodes and K is typically 4 to 6.) As was shown earlier, chordal graphs are inherently deadlock-free with respect to minimal routing.

Instead of a spanning tree, a spanning chordal subgraph could be used. For certain topologies, it would be much better. For some others, it would not be better at all. For instance, there are no triangles in a 2D mesh, therefore, a spanning chordal subgraph is a spanning tree.

But the chordal subgraph is needed only for deadlock-avoidance. *It is possible to route directly in the full network and use the chordal subgraph as a backup if and when a cycle in the channel dependency graph would be created.*

First, a spanning chordal subgraph is created. Then a minimal routing in this graph and its corresponding channel dependency graph CDG is generated. Note that CDG is guaranteed to be acyclic at this point. Then the routing is started again using the whole network. When a path is added to the routing, the channel dependency graph is checked. If a cycle is found, another path is selected. This step is performed iteratively until a path is found that does not add a cycle to CDG. The existence of at least one such path is guaranteed, since the route in the chordal subgraph is already in the channel dependency graph.

Cost Function

Before the details of the algorithms are described, consider the input parameters of the message routing. These are the interconnection network I and the list of source-destination node pairs L specifying the communication requirements of the application. An estimated load of each communication pair is also available, which enables the definition of a meaningful cost function for message route optimization.

As was shown in Chapter II, the distance a message covers has only a minor effect

on the latency with wormhole routing. However, that formula was developed assuming no other traffic in the network. The usage of paths other than the shortest ones increases the traffic. The more distance a message has to cover, the more likely it will block or be blocked by other messages. Therefore, message path length must be integrated into the cost function.

The load and the distance can be combined together in the following manner. For each link, the loads of the messages that use the given link are added together. Then the average load of the links in the whole network is computed. Longer messages increase the average load more than shorter ones. Messages with higher load have a larger effect on the cost. However, this cost function in itself is one sided.

Any minimal routing yields the same minimal cost for the given communication requirements. However, a system with a small average load can still perform poorly if one or more links are heavily loaded. The link with the maximum load represents a bottleneck in the system, and thus needs to be integrated into the cost function. The following formula is used:

$$C = W_a \cdot L_a + W_h \cdot L_h \tag{11}$$

where L_a is the average load, L_h is the maximum load, and W_a and W_h are empirical constants.

This cost function utilizes all the available information in the models. Other advantages include its low computational cost and its possibility of being incrementally computed as new message paths are added to the routing.

Non-Minimal, Partially Connected Routing Algorithm

Figure 24 shows the routing algorithm in detail. The first step is to order the list of communication pairs with decreasing load. Since there is no backtracking, the more important message routes must be considered first. This way, they have a better chance to use shorter paths. Next, the channel dependency graph (CDG) is initialized. All the nodes are added, but initially there are no links in it. Then a spanning chordal subgraph C is found. The algorithm locating C is described in the following section.

```
order communication pairs with decreasing load
create the nodes of channel dependency graph CDG for I
find chordal subgraph C for I
for every communication pair {
        select the shortest path with minimal cost in C
        route along that path
        add path to CDG
}
reset cost to 0
for every communication pair {
        find a set of A paths with increasing length in I
                while(set is not empty) {
                        select path with minimal cost
                        add path to CDG
                        if CDG is acyclic then
                                break loop
                        delete path from CDG and set
                }
if no path selected then
        route along the one selected in C
```

Figure 24 Deadlock-free, non-minimal, partially connected routing algorithm

Then a minimal routing in C is generated. The channel dependency graph is updated accordingly. The cost function is used to select a path in case of multiple shortest paths. At this point, CDG is guaranteed to be deadlock-free. CDG is used to store the load

values for each link in I, which are reset at this point, since the routes in C may not be used at all.

Next, a route for each communication pair in I is created. In general, every possible path cannot be evaluated, since the number of paths is not polynomial in the worst case. The number of paths considered is limited by a constant that is set high enough to have a rich set from which to select. The path resulting in the lowest overall cost is chosen. The channel dependency graph is searched for cycles. If no cycle is found, the path is accepted. Otherwise, the next best path is chosen repeatedly until the resulting channel dependency graph is acyclic. If none of the paths are acceptable, then the original path in the chordless subgraph C is selected. Since it is already included in CDG, the routing remains deadlock-free.

CDG remains acyclic during the entire process. Therefore, the routing is deadlock-free. A detailed evaluation of the algorithm is presented at the end of the chapter.

Spanning Chordal Subgraph

Figure 25 shows the algorithm for finding a spanning chordal subgraph in detail. First, every triangle of I is added to the chordal subgraph C if it does not introduce a chordless cycle in C. After this phase, C may be disconnected. To connect the components, the communication pairs are used to add those links to C that are likely to be needed in the routing. If a shortest path for a communication pair includes a link connecting two components, it is added to C, merging the two components. Since the communication pairs are ordered, the ones with higher loads are used first.

After evaluating all the communication pairs, C may still be disconnected. This means that all necessary routing could be done in the components separately. However, the spanning chordal subgraph needs to be connected. Any links can be used to connect the components, since they are not likely to be used in the routing.

```
for each triangle in I {
        if triangle does not add a chordless cycle to C then
                add triangle to C
}
while C is disconnected {
        select next communication pair
        find a shortest path in I
        if any link in the path connects two components of C then
                connect the components using that link
}
while C is disconnected {
        for every link in I {
                if link connects two components of C then {
                        connect the components using that link
                        if C is connected then
                                break the loop
                }
        }
}
```

Figure 25 Spanning chordal subgraph algorithm

Complexity Analysis

Let the number of nodes in I be n. Since every node has a constant number of links, the number of links in I is O(n). Let the number of communication pairs be p. What is the time complexity of finding a spanning chordal subgraph?

The number of triangles in I is O(n), since a node has a constant number of neighbors. Therefore, it can only be part of a constant number of triangles. Searching for possible

chordless cycles takes $O(n^2)$ time. The complexity of this phase is, therefore, $O(n^3)$.

There are p communication pairs. Finding a shortest path and checking whether it connects components takes O(n) time. The complexity of this phase is O(pn). In the final phase, every link is checked which takes O(n) time. The overall time complexity of the algorithm is, therefore, $O(n^3) + O(np)$. However, p is at most $O(n^2)$. Hence, the time complexity of finding a spanning chordal subgraph is $O(n^3)$.

The time complexity of the routing algorithm consists of the following components. Ordering the communication pairs takes $O(p \cdot \log p)$ time. Creating the channel dependency graph is O(n). Finding the chordal subgraph is $O(n^3)$. In the worst case, there may be an exponential number of shortest paths between two nodes. The algorithm can be modified to search only for a constant number of shortest paths. In this case, finding the paths takes O(n) time. Selecting the best path, routing, and updating CDG are each O(n). Therefore, routing in the subgraph takes $O(np) + O(n^3)$ time, which is $O(n^3)$ in the worst case.

In the second phase, a constant number of paths are found for each communication pair. This takes O(n) time. Selecting the best path and updating CDG are each O(n). Searching for cycles in the CDG takes $O(n^2)$ time. Therefore, the complexity of this phase is $O(pn^2)$.

The overall time complexity of the non-minimal, partially connected routing algorithm is

$$O(n^3) + O(pn^2) \tag{12}$$

Evaluation

The algorithm simultaneously avoids deadlocks and optimizes a cost function. The

optimization is a hill climbing procedure. As such, it finds only a local minimum. Nonetheless, the algorithm produces good results because the communication pairs are ordered.

The time complexity of the algorithm is $O(n^4)$ in the worst case. However, p is typically O(n), not $O(n^2)$, making the complexity $O(n^3)$. In practical cases, n is at most a few hundred. The exponential time complexity of optimal algorithms is prohibitive with this size. $O(n^3)$ is comparatively fast.

The algorithm has been compared to two others. Both are minimal routing algorithms, hence, neither guarantees deadlock-freedom. The first is a similar hill-climbing procedure, but it only allows shortest paths. If at a certain point all shortest paths result in cyclic CDG, the algorithm backtracks which makes the time complexity exponential. This algorithm is called *minimal*.

The *optimal minimal* algorithm generates all shortest paths for every communication pair and finds the deadlock-free configuration (if one exists) which produces the global minimum of the cost function. Its time complexity is consequently exponential. On average, its performance is even worse than that of the minimal algorithm, since it evaluates every combination all the time, while the other quickly finds a configuration if no deadlocks are encountered.

All three algorithms employs the same cost function as described above. The constants W_a and W_h have each been set to 0.5. The list of communication pairs has been generated randomly. The same pair could appear in the list more than once. This corresponds to the fact that more than one processes can be assigned to a node.

The first series of tests has been performed on the graph shown in Figure 26. There

is no deadlock-free, minimal, connected routing possible in this topology. The small size makes it possible to execute the exponential time complexity algorithms. However, the graph is slightly more complex than a simple 5-cycle. Table 16 summarizes the values of the cost function for the three different algorithms.

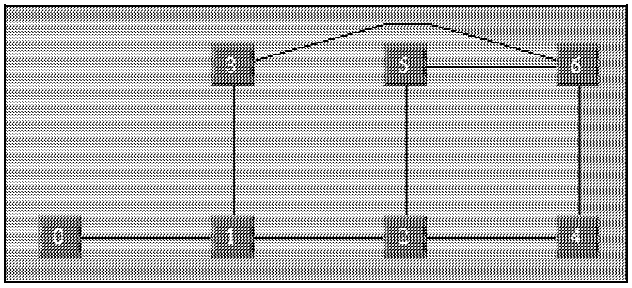

Figure 26 Test topology #1

Table 16 Results for test #1

p	Optimal Minimal	Minimal	Non-minimal
128	x	x	174.50
64	87.53	90.53	86.59
32	41.12	42.12	42.25
16	54.50	54.50	46.53
8	20.19	20.19	17.09
4	8.62	8.62	8.62
2	7.34	7.34	7.34
1	7.97	7.97	7.97

For small p (i.e. few communication pairs), there is no difference between the three

algorithms, as expected. For p=16, the two minimal algorithms produced the same result, but the non-minimal found a better solution in terms of the cost function. Since the non-minimal algorithm always has a higher (or equal) average load than the minimal ones, it is the highest load that makes the difference. The non-minimal algorithm is not restricted to shortest paths. It can select a longer path and avoid hot spots in the system.

On the other hand, because it is a hill-climbing procedure, it can get stuck in a local minimum. For p=32, the optimal minimal algorithm found the best solution. The minimal one provided a slightly worse result, while the non-minimal had the highest cost. However, the difference between the costs was negligible.

At p=64, the non-minimal provided the best solution. At p=128, the minimal algorithms failed to find a deadlock-free configuration, because no such solution existed for the input set. Note that the routing found by the non-minimal algorithm has a cost that is about twice as much as for p=64. Since the amount of communication approximately doubled, this means that finding a deadlock-free solution did not cause any additional cost increase.

For the second series of tests, a 5 by 5 torus has been selected as shown in Figure 27. There is no deadlock-free, minimal, connected routing for this topology. The size of the graph does not allow to run the optimal minimal algorithm for large p. Table 17 shows the number of iterations of the optimal minimum algorithm in the test cases. Table 18 summarizes the results.

For p=8 and below, the three algorithms produced the same results. For bigger p's, the optimal minimal could not be tested. For p=16, the minimal and non-minimal algorithms found the same solutions.

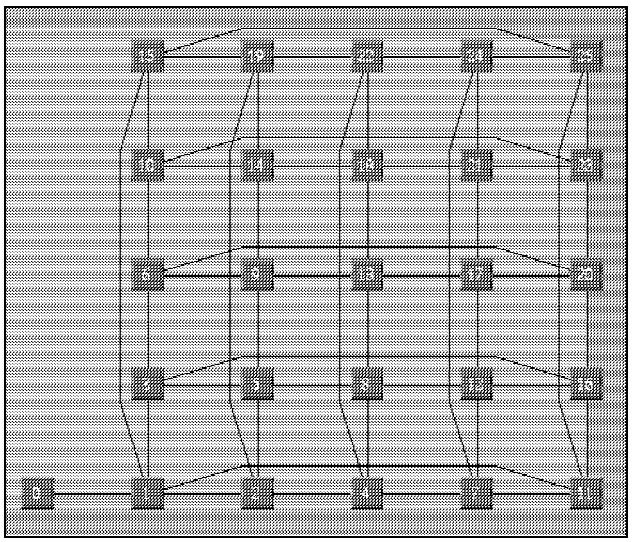

Figure 27 Test topology #2

Table 17 Number of iterations for the optimal minimal algorithm

p	Iterations
256	2E80
128	3E42
64	2E20
32	2E8
16	3E6
8	1296
4	4
2	6
1	3

Table 18 Results for test #2

p	Optimal minimal	Minimal	Non-minimal
256	x	x	105.80
128	x	45.37	54.07
64	x	29.33	25.89
32	x	20.47	14.66
16	x	21.24	21.24
8	9.42	9.42	9.42
4	8.25	8.25	8.25
2	5.11	5.11	5.11
1	3.60	3.60	3.60

For p=32 and p=64, the non-minimal proved to be better. Note that for p=32, the cost of the non-minimal solution is 50% better. For p=128, the minimal was able to find a 20% better solution. For p=256, the minimal did not finish in a reasonable time, probably because no deadlock-free configuration was found and it had to evaluate every possible combination (2E80 of them). The non-minimal algorithm produced a solution with a cost about twice as much as for p=128.

The analysis and the test results show that the presented non-minimal, partially connected routing algorithm is indeed a good solution to the routing problem in networks with arbitrary topologies. The solution guarantees deadlock-freedom and finds a good routing strategy in terms of the defined cost function. The time complexity of the algorithm is $O(n^4)$ which provides for fast execution in practical cases. Note that the algorithm is not limited to the signal processing domain. It is applicable to any domain where the communication requirements of the applications are known before runtime.

Implementation

The non-minimal, partially connected routing algorithm has been implemented as part of the Graphical Configuration Manager (GCM). The model interpreter, the Parallel Application Builder (PAB), generates the hardware description file for the GCM, the list of required message source-destination node pairs, and the load information. The GCM performs the routing and generates message routing maps for each node in the system.

CHAPTER VII

ASSIGNMENT

One of the most important phases of the model interpretation is the assignment of signal flow primitives to hardware nodes. The goal is to maximize the throughput of the system. The system models contain information relevant to the assignment problem. The signal flow models describe the computational blocks with their estimated execution time, and the communication between them with the required bandwidth. The hardware models describe the available processors and their interconnection topology. In case of heterogeneous networks, the speed of the individual nodes and the bandwidth of the communication channels are also available. The assignment constraints models represent different restrictions on the assignment. In Chapter II, three general methods for solving the assignment problem were described: optimal solutions, general optimization techniques, and assignment specific heuristic methods.

Optimal methods are only feasible for very small systems. General optimization techniques, such as simulated annealing or genetic algorithms, are restricted to relatively simple cost functions because of the size of the search they perform. A simplistic cost function cannot describe the solution of such a complicated problem as the process assignment accurately. It is not worthwhile to find even the global minimum of an inaccurate measure. The information available in the system models makes the definition of a more accurate, though computationally more expensive, cost function possible. Furthermore, the assignment constraints can guide a deterministic procedure, but they are

no help in a random search that simulated annealing and genetic algorithms employ. An assignment specific heuristic search offers the best solution for the Model-Integrated Parallel Application Synthesizer (MIPAS).

The role of the heuristic is to reduce the size of the exponential search space. Decomposing the problem into smaller subproblems, solving them independently, and combining the results together to get the overall solution is a widely used technique in optimization. The quality of the solution depends greatly on the decomposition of the problem. The coupling among the subproblems must be weak, while they must contain strongly coupled components. This decomposition technique is itself a heuristic approach. Identifying the subproblems, however, can be done quantitatively.

The technique of nested epsilon decompositions is proposed by Siljak [51]. The underlying idea is to disconnect the edges of the problem graph with a weight smaller than a given threshold ε, and identify the resulting subgraphs. This step is performed iteratively on the subgraphs with increasing thresholds. The result is a nested, i.e. hierarchical, decomposition of the problem. ε, the strength of coupling, plays an important role determining the size and number of subproblems. The epsilon decomposition algorithm is a simple, but powerful technique. Its only significant drawback is that it is not guaranteed to find a decomposition of the problem. There are simple example graphs that are not epsilon decomposable. Consider, for example, a graph with a spanning tree having all edges with the maximum weight w in the system. If ε is smaller than w, then no decomposition is possible. If ε is greater than w, then all the edges of the graph are removed. A typical parallel instrumentation system with a single host collecting processed data from the network has a signal flow graph with a structure similar to this example.

Hence, nested epsilon decompositions cannot be applied in the domain of the MIPAS, nonetheless, they illustrate the strength of hierarchical decomposition.

The models of the system in the Multigraph Architecture are organized into hierarchies. In the MIPAS, the low-level signal flow models, the primitives, are grouped together to form compounds. Compounds are clustered to construct higher level compounds. The hierarchical tree structure of the signal flow models provides a decomposition of the graph similar to nested epsilon decompositions. The hierarchy is not based on the selection of thresholds, it is user-defined. Utilizing the hierarchy of the models for solving the assignment problem assumes that the decomposition of the signal flow graph groups strongly coupled components together. The relation between these components may not always be based on the weight of their connection, i.e. the communication between them, but some other criterium. The hierarchy typically reflects how the structure of the signal flow graph is logical to the user. This is not necessarily a problem, since a logical decomposition can provide a good assignment. Furthermore, the topology of the multiprocessor interconnection network is usually designed to match that of the signal flow graph. Since there are more signal flow primitives than hardware nodes, the hardware topology usually reflects the clustering of the signal flow models.

Nevertheless, utilizing the hierarchical decomposition of the signal flow models for solving the assignment problem is a heuristic approach. As such, it is not guaranteed to find the optimal solution.

The chapter is organized as follows. The first section describes and analyzes the hierarchical assignment procedure. Next, an alternative to the traditional assignment technique, a processor interconnection network synthesis approach is presented.

Hierarchical Assignment

Cost function

In general, the cost function describes the quality of an assignment. A delicate balance must be found between the accuracy and the computational cost of the cost function. The more accurately it describes the assignment, the better the expected results will be. However, a cheaper function allows more possibilities to be evaluated making the search space less restricted. The cost function consists of two parts: one corresponds to the computational load balance, the other describes the interprocessor communication overhead.

The variance of the loads of the individual processors captures the computational load balance accurately and it is computationally inexpensive. Note that the loads of the individual hardware nodes are normalized according to their performance attribute. The function employed in the previous chapter describes the cost of the interprocessor communication as accurately as possible in the absence of measured timing information. The computational expense of the function is low when used for message routing. However, using the function for the assignment would require generating the message routes as part of the cost function. The time complexity of the routing algorithm is prohibitive for this purpose; this cost function cannot be applied.

A reasonable compromise is computing the sum of the products of the communication loads and the corresponding distances in the processor graph, as proposed by Aggarwal [37]. Even though this cost function does not consider shared physical links, the message routing strategy will minimize their effect by employing the more accurate cost function

and optimizing message paths.

The communication and computation costs are combined and the overall cost is computed according to the following formula:

$$C = \sum (c_j \cdot d_j) + W \cdot \sum (l_{avg} - l_i)^2 \quad (13)$$

where c_j is the load of communication between two processes, d_j is the corresponding distance in the processor interconnection network, W is an empirical constant, l_{avg} is the average computational load in the system, and l_i is the load of processor i.

Algorithm

The assignment algorithm uses the hierarchy of the signal flow models to guide its search. Note that the hierarchy of the hardware models are not considered; the flat topology consisting of nodes is used. The hardware nodes are organized into a hierarchy according to their physical location. Two processors on the same board are grouped together, for example. A direct connection between two processors in two different racks is just as important as a connection between two nodes on the same board. The hierarchy of the hardware models do not contain any useful information for the assignment.

Figure 28 shows the hierarchical assignment algorithm in detail. The procedure starts at the top level of the signal flow hierarchy. It considers one compound model at a time at a given level. The hardware topology assigned to the top level signal flow model is the whole network. Successive levels use the part of the network that was assigned to the compound at the previous level. The first step is to check whether an exhaustive search is possible.

```
level = top level
repeat {
  for every SF compound and corresponding HW topology at level {
    while exhaustive search is not possible {
      if (number of HW nodes > number of SF components) then
        cluster HW topology
      else
        cluster SF topology
    }
    try every possible assignment satisfying the constraints
    select the one with the minimal cost
    expand HW topology
    expand SF topology
  }
  if more levels then
    go to next lower level
  else
    break loop
  repeat {
    select best one from every possible (k < K) local transformations
    if there is no cost improvement then
      break loop;
  }
}
```

Figure 28 Hierarchical assignment algorithm

For N signal flow components and M hardware nodes, the number of possible assignments is, in general, M^N. Because of the assignment constraints, the actual number can be considerably less. A constant limit on this number must be set depending on the available time and computing power. If there are too many possibilities to evaluate in a reasonable time, then the size of either the signal flow or the hardware topology needs to be decreased by an appropriate clustering algorithm. The procedure clustering the hardware topology is as follows.

It repeatedly finds the simple node, a node that has not yet been clustered, with the smallest number of simple node neighbors. The algorithm selects one of those neighbors

and merges the two nodes together. It updates their and their neighbors' connection lists accordingly. The procedure stops when no more change is possible. Note that the host node must not be clustered and the procedure may leave other simple nodes unchanged. Whenever the size of the network gets small enough to make an exhaustive search possible, the procedure is aborted. See Figure 29 for an example transformation.

Note that the communication link attribute specifying the speed of the given link may be changed during this procedure. For example, nodes **A** and **B** in Figure 29 have two original links between them, therefore, their speed must be added together. Similarly, the performance attribute of the individual nodes are updated when they are merged.

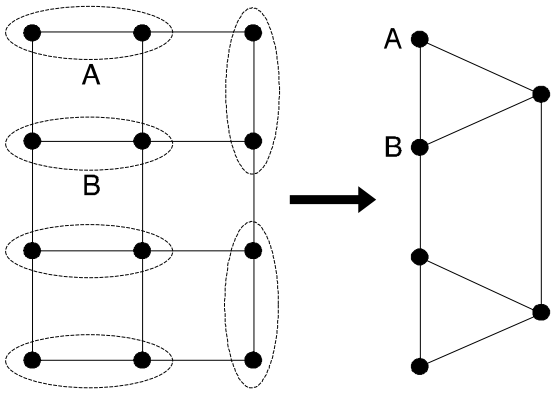

Figure 29 Hardware clustering

Clustering the signal flow graph proceeds differently. The strength of relationships between signal flow components vary. Strongly coupled components need to be identified for clustering. The coupling is defined by a function, which includes the communication rate between the two selected components, the combined communication rate of the two

nodes (other than between each other), and their computational loads. The strength of coupling is increased by more communication between each other and less communication to or from other components. The computational load is included in the function to keep the resolution as high as possible. Components with small loads are better candidates for clustering than blocks with high loads. Note that the assignment constraints must be considered as well. If the lists of available hardware nodes for two signal flow components do not have common elements, then the two nodes cannot be merged at all. The more common elements they have, the better it is to merge them. The following formula is used:

$$Cost_{ij} = W_{cout} \cdot \Sigma(C_{ik} + C_{ki} + C_{jk} + C_{kj}) - W_{cin} \cdot (C_{ij} + C_{ji}) + W_l \cdot (L_i + L_j) - W_n \cdot N_{ij} \qquad (14)$$

where C_{mn} represents the communication rate from signal flow component m to n, L_m is the computation load (e.g. estimated execution time) of signal flow component m, N_{ij} is the number of common hardware nodes the given components can be assigned to, and W_{cout}, W_{cin}, W_l, and W_n are constants. The minimum of this function, which may be negative, represents the best candidate for merging. Note that if $N_{ij} = 0$, then the nodes cannot be clustered.

After merging two components, two things are checked. If an exhaustive search is possible for the assignment, then the clustering is aborted. Otherwise, if the number of hardware nodes becomes greater than the number of signal flow components, the hardware topology clustering algorithm is started.

When an exhaustive search becomes feasible, every possible assignment that satisfies the assignment constraints is evaluated. The one with the minimal cost is selected. Note that in one case, i.e. when no clustering is necessary before the exhaustive search, the

number of hardware nodes may be greater than that of the signal flow components. Since the assignment procedure does not cluster hardware nodes, i.e. it assigns a signal flow component to one hardware node, this would result in nodes with no assigned components. Instead, the direction of the assignment is reversed: hardware nodes are assigned to signal flow components. The only constraint that needs to be satisfied is that the signal flow primitives are not divisible, they must be assigned to a single hardware node. After the optimal solution has been found for the given level and compound, the original signal flow and hardware topologies, i.e. the ones before clustering, are restored. Then the process is repeated for the next compound at the current level.

After the level has been finished, the signal flow topology is refined by going to the next lower level. First, the current assignment is improved by local transformations. Even if the optimal solution was found for all the compounds at the previous level, there is a good chance for improvement, since the granularity is finer at this point. A local transformation is defined as follows.

A signal flow component at either end of a connection spanning different hardware nodes is moved to the other node. The new cost is computed incrementally. All possible ($k < K$) transformations are performed and evaluated. The constant K is typically not more than 2, because of the large number of possible combinations. The best set of transformations is selected, i.e. the one which improves the cost by the largest amount. This is commonly called a hill climbing procedure.

A simple example of the assignment procedure is shown in Figure 30. The top level signal flow model consists of two components **A** and **B**. **A** consists of **C**, **D**, and **E**. **E**, in turn, has two more components **F** and **G**. The load for each component is shown as

well. The communication load is uniform for every connection. The hardware topology is a 2 by 2 mesh. At every step, an exhaustive search is possible, there is no need for clustering any of the topologies.

At the top level, **B** is assigned to node **W**; the remaining three nodes are assigned to **A**. The load is well balanced. Hence, no local transformations are carried out after going to the next level. At this point, **B** is left alone, since it is a primitive. **X** and **Y** are assigned to **E**, **Z** to **C** and **D**. At the lowest level, no local transformations are necessary. **X** is assigned to **F**, and **Y** to **G**. The resulting assignment is optimal.

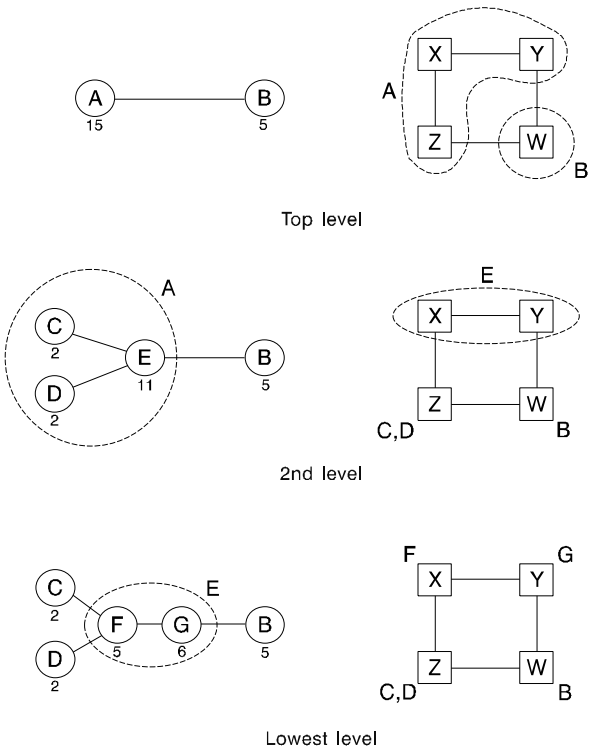

Figure 30 Assignment example

Complexity Analysis

The size of the assignment problem is hard to characterize in this case. Let n_{sf} denote the number of signal flow primitives. Let n_{hw} denote the number of hardware nodes. A compound model typically contains at least two user-defined components, i.e. compounds and/or primitives. In theory, a compound may contain only one primitive adding an artificial level to the hierarchy. A compound containing only one compound is meaningless. Therefore, the number of hierarchy levels is $O(\log n_{sf})$. The number of components in a compound may be $O(n_{sf})$ in the worst case. The most likely candidate for such a case is the top level signal flow compound model.

Clustering the hardware topology once takes $O(n_{hw})$ time. Note that the number of edges in the network is $O(n_{hw})$ because of the fixed degree of the nodes. Since the resulting graph has approximately half the size of the original one, there are at most $\log n_{hw}$ steps necessary. Finding and merging two signal flow components takes $O(m_{sf})$ time. That step reduces the size by one. Therefore, the time complexity of clustering the signal flow graph is $O(m_{sf}^2)$. Note, however, that clustering the whole signal flow graph is never attempted. Only one compound is considered at a time. The size of that compound, however, may be $O(n_{sf})$ making the complexity the same. Note that the average case is much better, i.e. constant.

The exhaustive search takes constant time, since the search space is limited. Note that the limiting constant is potentially large. The procedure optimizing the assignment by local transformations has exponential time complexity in the worst case. On the other hand, the hill climbing procedure going through an exponential number of steps is extremely unlikely. The starting assignment is already good, there is not much room for

improvement. Nevertheless, the number of transformations must be limited. For one step, a single transformation is selected from the m_{sf}^K possible ones. A constant number of steps are carried out at most. Since K is typically 2, the time complexity is $O(m_{sf}^2)$.

For one hierarchy level, the hardware topology clustering is $O(n_{hw} \cdot \log n_{hw})$, the signal flow topology clustering is $O(m_{sf}^2)$, and the optimization by local transformations is $O(m_{sf}^2)$. Since a signal flow primitive cannot be divided further, $n_{sf} \geq n_{hw}$. In addition, $m_{sf} = O(n_{sf}^2)$, since there is no limit on the degree of a signal flow primitive. Furthermore, there are $\log n_{sf}$ hierarchy levels. The overall time complexity is, therefore,

$$O(n_{sf}^4 \cdot \log n_{sf}). \quad (15)$$

Evaluation

The hierarchical assignment algorithm has been implemented as part of the model interpreter. Thorough evaluation of the approach is a hard problem for two reasons. While it would be beneficial to compare the results produced by the algorithm to the optimal solutions for a different set of problems, it is not feasible. Optimal solutions can be attained for small problems only because of the exponential time complexity. The hierarchical assignment algorithm performs an exhaustive search whenever it is feasible. Therefore, it finds the optimal solutions for small problems. There is nothing to be gained from this kind of comparison.

Second, a comparison with another heuristic approach or a general optimization technique, e.g. simulated annealing or a genetic algorithm, would provide valuable information about the characteristics of the hierarchical assignment algorithm. However, none of these techniques has a readily available implementation. The unique requirements

of the Model-Integrated Application Synthesizer (MIPAS), especially the explicitly modeled assignment constraints, make it even more difficult to port an existing algorithm and integrate it into the MIPAS. Implementing another approach from scratch would require as much effort as the implementation of the hierarchical assignment algorithm did. Therefore, the only tests that could be carried out were comparisons with results produced by manual assignment.

The problem size was limited, though not as severely as for optimal algorithms, because the complexity of the problem can become overwhelming for humans. The signal flow model of the test cases was that of the example application described in the next chapter. It contains approximately 50 primitives, i.e. actornodes. The hardware configuration varied from as low as 5 nodes up to 25.

The test results look promising. For small hardware configuration the automatically generated assignments were almost identical to the manual ones. The small variations were mainly due to the optimization by local transformations. Actors with small loads were moved to neighboring nodes because the cost function had a slightly smaller value.

For bigger hardware configurations, the automatically generated assignments were significantly different than the manual results. The hierarchical assignment algorithm even left some hardware nodes idle because the advantages of less interprocessor communication outweighed the advantages of better computational load balance. Note that the application has some actors with very small loads but relatively high communication rates.

Manual analysis of the automatic assignments did not reveal any problems. The algorithm does not follow the same path that human intuition suggests. Hence, the results

are different. In terms of the cost function, however, the hierarchical assignment algorithm produced better results than the manual process. Note that with a small number of strategically placed assignment constraints, the algorithm can be forced the follow the general structure of the manual solution. Further quantitative analysis is necessary to identify the weaknesses of the algorithm. A qualitative evaluation leads to the following observations.

The cost function is reasonably accurate and computationally cheap. Its most important drawback is that it does not consider the actual message paths, only their minimal lengths. This problem is corrected by the message routing strategy that minimizes message contention. Using the signal flow hierarchy as a heuristic to cut the search space is very useful. It makes the heuristic itself application dependent, since it utilizes the knowledge of the user about the application.

In most cases, an exhaustive search can be used directly. Sometimes, most probably at the top level, clustering one or both of the hierarchies may be necessary. The merging of signal flow components utilizes all the available information, i.e. the computation and communication loads. The weak point of the algorithm is the hardware topology clustering procedure. It is ad hoc in that it does not utilize any assignment related information. A better algorithm is needed to reduce the size of the hardware network when it is necessary.

Hardware Topology Synthesis

An alternative approach to the assignment problem is hardware topology synthesis. The idea is to automatically match the topology of the processor interconnection network to that of the signal flow graph. This approach is only possible if the topology of the available hardware is flexible. Smitley and Lee show that the problem is NP-complete, even with strong simplifications [36].

A heuristic hardware topology synthesis algorithm has been developed as part of the MIPAS. The applied heuristic is based on the hierarchical structure of the signal flow models, similarly to the assignment algorithm. For the sake of simplicity, a homogeneous set of processors is assumed (there are no computational or communication performance differences) and the assignment constraints are not considered.

Cost Function

The cost function is simpler than the function applied in the assignment algorithm. The first phase of the algorithm does not consider the topology of the hardware. It optimizes the computational load balance, i.e. it determines the number of processors that need to be assigned to the different (clusters of) nodes. It assumes that if two signal flow components are assigned to different nodes, then those nodes are directly connected. Topological constraints are considered in the second phase only. Therefore, the distance of hardware nodes is not included in the function. The cost is a combination of the sum of interprocessor communication loads and the variance of the computational loads. The formula is

$$C = \sum c_j + W \cdot \sum (l_{avg} - l_i)^2 \qquad (16)$$

where c_j is the load of communication between two processes assigned to different nodes, W is an empirical constant, l_{avg} is the average computational load in the system, and l_i is the load of processor i. Note that a signal flow component may have more than one hardware nodes assigned to it, in which case, the load of the signal flow block is divided equally among the nodes.

Synthesis Algorithm

Figure 31 shows the hierarchical hardware topology synthesis algorithm in detail. The procedure is started at the top level of the signal flow hierarchy. It considers one compound model at a time at a given level. First, the algorithm assigns every available node to the top level signal flow model.

The average load of a signal flow component of the given compound model is computed by adding all the loads together and dividing the sum by the number of assigned nodes. Then the load of every component is compared to the average load. If the load is greater than K times the average load for the greatest possible integer K, then K nodes are assigned to the given block. Furthermore, the number of available nodes is decreased by K. After this procedure is completed, there are typically some more hardware nodes available.

The goal is to perform an exhaustive search. This may not be possible because of the exponential search space. The number of different assignments is M^N / M!, where N is the number of signal flow components, M is the number of hardware nodes. The reason for dividing by M! is that differentiating between processors is not necessary.

```
level = top level
assign the number of available HW nodes to the top level SF compound
repeat {
  for every SF compound and assigned number of HW nodes at level {
    compute avgload
    for every SF component in current compound {
      if (load >= K · avgload AND load < (K+1) · avgload) then {
        assign K nodes to component
        decrease the number of available HW nodes by K
      }
    }
    while exhaustive search is not possible
      cluster SF components
    try every possible assignment
    select the one with the minimal cost
  }
  if more levels then
    go to next lower level
  else
    break loop
  repeat {
    select best one from every possible (k < K) local transformations
    if there is no cost improvement then
      break loop;
  }
}
create optimal HW topology
find spanning subgraph with maximum degree D
```

Figure 31 Hierarchical topology synthesis algorithm

If the number of possible assignments is too big, the search space needs to be reduced. This is accomplished by clustering signal flow components incrementally until an exhaustive search is possible. A cost function is defined to select the best candidates for clustering. There are two factors worth considering. First, two signal flow components are strongly coupled if the amount of communication between them is relatively large. Second, the sum of the loads of the components needs to be close to an integer multiple

of the average load. The cost function used in this phase of the algorithm is based on these two factors. The algorithm evaluates every possible pair of signal flow components, selects the one with the minimal cost, and merges the blocks. This step is repeated until an exhaustive search is possible.

The cost function mandates some restrictions on the assignment strategy. If every signal flow component was clustered into a single group and every node was assigned to it, the cost would be minimal because the load would be evenly distributed and there would be no interprocessor communication, since no communication is considered inside a cluster. In a previous phase, nodes were assigned to components having loads greater than the average load. Those can be called "big" components and the blocks with loads lower than the average "small" blocks. The following restrictions must be made. If there are only small components and K available hardware nodes, exactly K clusters must be created. Consequently, each cluster will have one assigned node. If there are a mixture of small and big components, then only one big component per cluster is allowed. Furthermore, any cluster can have only one additional hardware node assigned to it. This means that a cluster consisting of small components can have only one assigned node. A cluster with one big and some more small components can have the same number of nodes originally assigned to the big component or one more. Note that these restrictions are not severe, they are logical. Furthermore, they are absolutely necessary to produce some results at this level of the hierarchy. Allowing bigger clusters would push the problem down the hierarchy where the search space is much bigger.

During the exhaustive search, every possible clustering and assignment satisfying these restrictions are evaluated and the one with the minimal cost is selected. The load balance

may not be good at this point because of the granularity. At the next level, the signal flow graph is refined and the assignment is improved by using the same method based on local transformations as was used in the hierarchical assignment algorithm. The only difference is the cost function.

This procedure goes on until the lowest level of the hierarchy is reached resulting in the signal flow graph with primitives clustered into as many groups as there are available hardware nodes. The topology of the processor interconnection network can be constructed by assigning one node to each cluster of signal flow primitives and connecting two nodes together if there is a connection between the corresponding clusters in the signal flow graph. This topology would be optimal, every pair of communicating processors would have a direct link. Message routing would not even be necessary. However, the number of communication links of a single node is limited. In other words, the degree of each node in the network is bounded by a constant. Therefore, a degree-bounded, connected, spanning subgraph of the network must be found.

This problem is NP-complete [27]. Assuming that there are no hardwired connections between any processors, the easiest way to find a degree-bounded subgraph is to delete the appropriate number of links at every node having a higher than allowed degree. Deciding which links to delete is an optimization problem requiring a cost function.

Link with small communication loads are the best candidates for deletion. However, that communication still needs to take place. The consequence of a link deletion must be considered. A good and simple measure is the distance of the two nodes after the deletion of their common link. Multiplying the communication load and the resulting distance constitutes a reasonable cost function. There is one more factor worth considering. A link

connecting two nodes both having higher than allowed degrees is a better candidate than a link with only one such node. The deletion of the former decreases the degree of two nodes. The applied cost function is

$$C_{ij} = l_{ij} \cdot d_{ij} \cdot w_{ij} \tag{17}$$

where l_{ij} is the communication load between nodes n_i and n_j, d_{ij} is the distance after deleting the link, and w_{ij} is 0.5 if the degree of both n_i and n_j is greater than the limit, infinite if the degree of none of them is greater than the limit, and 1 otherwise.

Figure 32 shows the algorithm in detail. In the first phase, the search process finds the link with the minimum cost and deletes it. This step is repeated until the degree bounded subgraph has been found. This phase of the algorithm could be made more complicated. A link deletion not only affects the given link, but possibly any communication with a previously deleted link. How deleting a link changes the distance of every pair of nodes whose common link has been deleted before could be checked. This extra step would make the time complexity less attractive and is not worth doing.

At the end of Phase I, there can be nodes with too high degree and only such links that deleting any one of them would disconnect the graph. Consequently, their costs are infinite, since the distance after deleting such a link would be infinite. In this case, the problem is redefined. Instead of finding a subgraph, new links are added that were not present in the original graph.

At the beginning of Phase II, the only nodes that have high degree are these special ones because any lower cost links are already deleted. One of these links with infinite cost is selected and deleted and a new link is added to connect the two components. When selecting the location of the new link, the cost is not considered because this case

is very unlikely in real applications.

```
Phase I:
    create original graph based on SF clusters
    while there are nodes with higher than allowed degree {
        find the link with the minimal cost
        if cost is infinite then
            break loop
        delete link
    }

Phase II:
    for every node with higher than allowed degree {
        while degree is high {
            delete a link creating two components
            for each component {
                find a node with lower than allowed degree
                if not found then {
                    find a node with allowed degree that
                        has a link that can be deleted
                    delete the link
                }
            }
            connect nodes
        }
    }

Phase III:
    while there are more than one node with lower than allowed degree {
        for every link that can be added {
            add link
            compute distance for every node pair
               that had a link in original graph but not any more
            delete link
            compute distances again
            compute the difference
        }
        select link with greatest difference
        add it to the graph
    }
```

Figure 32 Degree-bounded graph generation

If no new link can be added because every node in one component has the highest

allowed (or higher) degree, then a link is deleted in that component. Note that there must be a link that can be deleted in the component. It is not possible to have a graph with all the nodes having the maximum degree and with every link being and edge-cut. This procedure is repeated until a degree-bounded graph is reached.

At the end of Phase II, an appropriate hardware topology is available. However, there can be nodes with less than the allowed degree. Links can be added to the graph just by connecting these nodes together as they are found. Phase III follows a better approach. The best link to add can be chosen by comparing the distance of two nodes that were connected in the original graph, before and after adding the given link for all such node pairs. The one decreasing the sum of the distances the most is selected. The load can be factored in as well.

The situation is more complicated when there are hardwired connections between hardware nodes. In this case, there are small networks of nodes instead of individual processors. These graphs are ordered based on the number of links each has. The procedure starts with the one which has the greatest number of links and tries to find a subgraph of the desired network generated by the hierarchical topology synthesis procedure, which is isomorphic to it. If a perfect match cannot be found, the best one is selected, i.e. the one with the highest number of matching edges. The non-matching links are added to the network. The value of w_{ij} of each of the links in the subgraph is set to infinity. In other words, those links cannot be deleted. This procedure is repeated for every small hardwired network. Then the same search process is followed as shown in Figure 32 until a degree-bounded graph is generated.

Complexity Analysis

Most of the same assumptions as in the analysis of the hierarchical assignment algorithm can be made. Let n_{sf} denote the number of signal flow primitives. Let n_{hw} denote the number of hardware nodes. The number of hierarchy levels is $O(\log n_{sf})$. The number of components in a compound is $O(n_{sf})$.

Computing the average load and comparing it to the individual loads takes $O(n_{sf})$ time. The complexity of the incremental clustering algorithm is $O(n_{sf}^2)$. The exhaustive search needs to be limited by a constant. The optimization step based on local transformations needs $O(m_{sf}^2)$ time, similarly to the assignment algorithm. The time complexity of the first phase is, therefore,

$$O(m_{sf}^2 \cdot \log n_{sf}) = O(n_{sf}^4 \cdot \log n_{sf}). \tag{18}$$

Originally, there are $O(n_{hw}^2)$ links in the hardware topology. Links need to be deleted to get $O(m_{hw}) = O(n_{hw})$ edges. The number of deleted links is $O(n_{hw}^2)$ in the worst case. The time complexity of deleting an edge is $O(n_{hw})$. Deleting all the necessary edges takes $O(n_{hw}^3)$ time.

In the next phase, there may be $O(n_{hw}^2)$ edges with infinite cost that need to be deleted. Finding another link to connect the components takes $O(n_{hw}^2)$ time. The complexity is then $O(n_{hw}^4)$. In the last phase, there are at most $O(n_{hw})$ edges to add. The distances of $O(n_{hw}^2)$ node pairs are checked. That takes $O(n_{hw})$ time for one pair, $O(n_{hw}^3)$ for all the pairs. Altogether, creating the degree bounded graph takes $O(n_{hw}^4)$ time. The overall time complexity is $O(n_{sf}^4 \log n_{sf} + n_{hw}^4)$. Since $n_{hw} < n_{sf}$, the result is

$$O(n_{sf}^4 \log n_{sf}). \tag{19}$$

If there are hardwired connections, the maximum number of nodes in any one

hardwired network needs to be limited because of the exponential time complexity. Based on typical board configurations, a reasonable limit is 4. Finding one matching subgraph takes at most $O(n_{hw}^4)$ time. There are $O(n_{hw})$ hardwired networks. The time complexity is $O(n_{hw}^5)$, which may be greater than $O(n_{sf}^4 \cdot \log n_{sf})$.

Evaluation

The hardware topology synthesis algorithm incorporates several novel ideas. Using the hierarchy of the signal flow models to reduce the search space makes the core of the algorithm, the heuristic, application dependent. Furthermore, the user has control over the algorithm this way. Breaking the problem into two distinct phases, i.e. balancing the computational load and minimizing communication overhead, greatly simplifies the problem and does not have any significant drawbacks. The time complexity is polynomial, but high. Note, however, that the complexity of the average case is much better.

The hierarchical hardware topology synthesis algorithm has not been implemented yet. Generalization of the strategy to work with a heterogeneous set of processors and satisfy the assignment constraints is the subject of future research.

CHAPTER VIII

EXAMPLE SYSTEM

The technique of automatically synthesizing large-scale, parallel instrumentation systems can be best illustrated through an example. The Parallel Signal Analyzer (PSA) program is a multi-channel signal processing/instrumentation system running on transputer or C40 networks with a PC host. The PSA is automatically synthesized from high-level models utilizing a general signal processing library and an application-specific user interface.

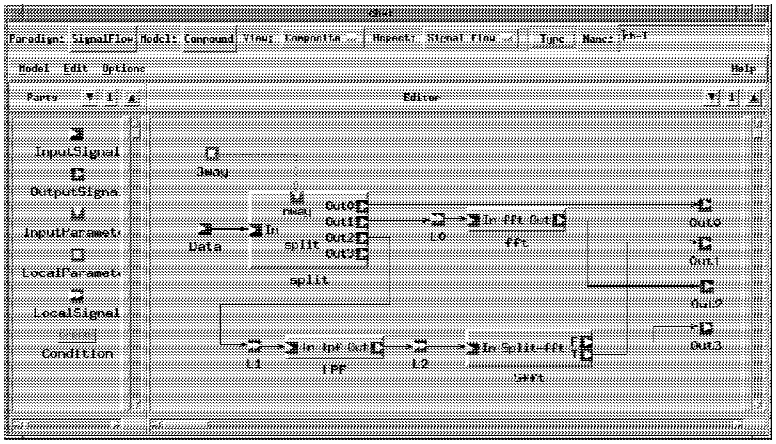

Figure 33 PSA channel I signal flow model

Figure 33 shows the signal flow compound model for one channel in the Graphical Model Builder (GMB). Data is coming through input signal **Data**. It enters primitive **split**, which splits the data stream as many ways as specified by local parameter **3way**. In this

case, its value is 3. One data stream leaves through output signal **Out0**. Another one goes through local signal **L0** to primitive **fft**. The output spectral data goes to output signal **Out2**. The third data stream of the splitter goes through local signal **L1** to primitive **lpf**, which represents a low-pass filter. The output of the filter propagates through local signal **L2** to compound **Sfft**. This model has two outputs. One is the original input data, the other is its spectrum. They propagate to output signals **Out1** and **Out3**, respectively.

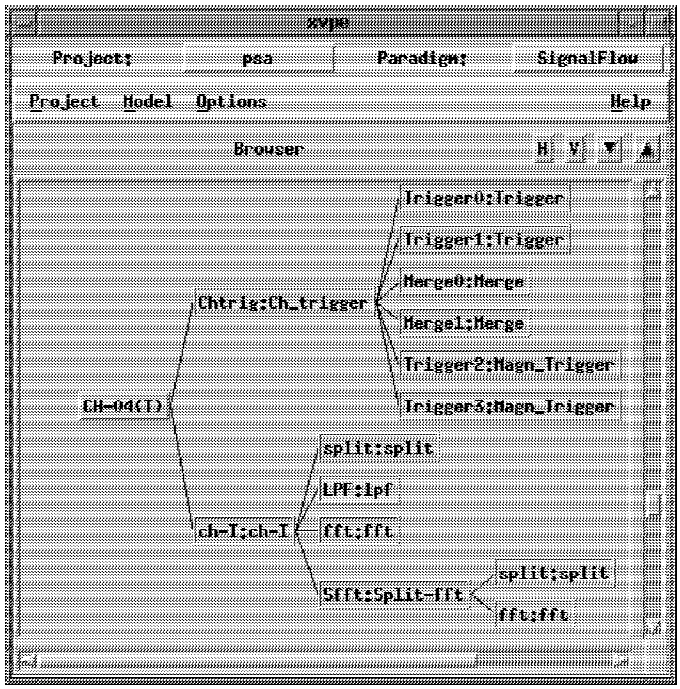

Figure 34 PSA signal flow models in the model browser

The same model of channel I and the corresponding plot generation modules are shown in the model browser in Figure 34. The model identifiers have the form "instance name : type name". The compound **CH-04** contains two compound models: **ch-I** and **Chtrig**. **ch-I** contains all the components described above. Note that by using types,

several instances of the same model can be used, even at different levels of the hierarchy. See, for example, **fft** directly under **ch-I** and in **Sfft**. **Chtrig** contains several plot generator primitives (called **Triggers**), one for each output of **ch-I**. Altogether, **CH-04** contains 11 primitives.

This example configuration of the PSA contains 4 channels, each with 10-15 primitives, a user interface, and a plotter. Altogether, the system consists of approximately 50 primitives. Adding an arbitrary number of additional channels is easy. Their signal flow must be modeled and added to the top level model. The model interpreter generates the new configuration automatically.

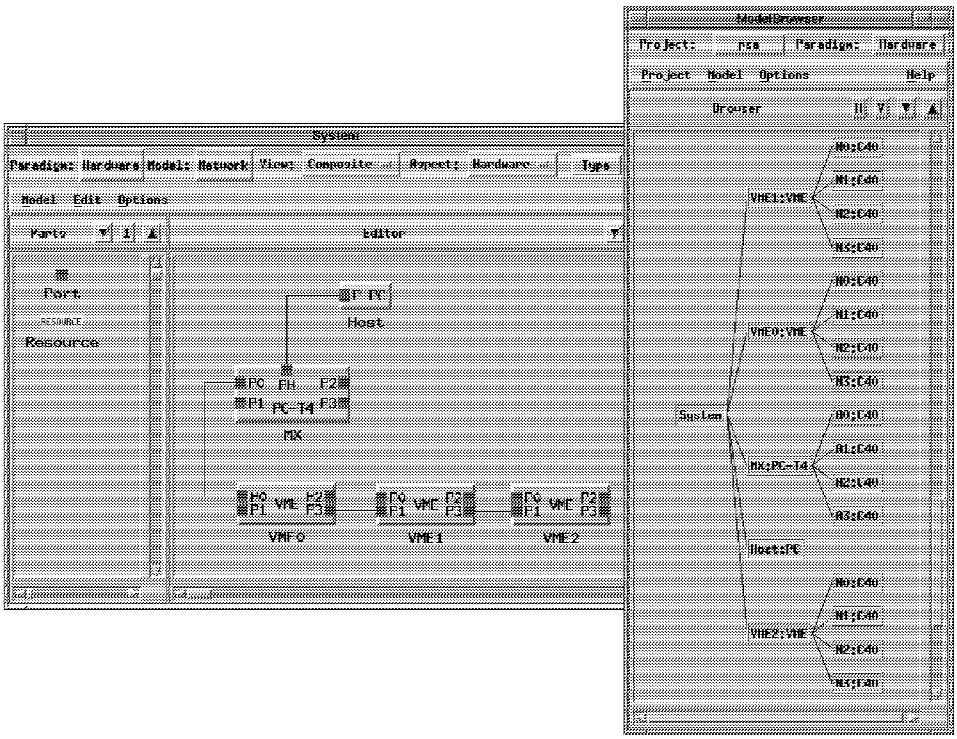

Figure 35 Hardware model

The PSA can be executed on arbitrary transputer or C40 networks with a PC host. An example hardware configuration is depicted in Figure 35. The model browser shows the whole hierarchy. The network model **System** consists of a node **Host** of **PC** type, a network **MX** of **PC-T4** type, and three additional network models of **VME** type, each containing four **C40** nodes. There are 16 C40 processors and a PC in the system.

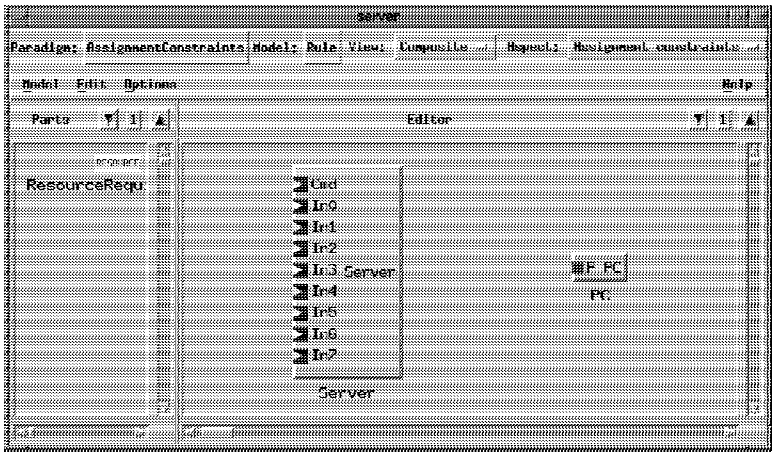

Figure 36 Assignment constraints rule model

There are two constraints the automatic assignment strategy needs to satisfy. The user interface and the plotter must run on the host, since only that node has a display. The assignment constraint rule specifying the latter requirement is shown in Figure 36. The signal flow reference **Server** corresponding to the plotter is associated with the hardware model reference **PC** corresponding to a **PC** type model. Note that hardware type reference is used, since the plotter does not have to run on the host, it could run on any other PC if there were more in the system. The top level assignment constraint model contains this rule and another one corresponding to the assignment of the user interface. Note that this configuration of the PSA collects data generated by a program, not sampled by A/D

converters. Therefore, the data input do not create any additional assignment constraints.

The model interpreter, the Parallel Application Builder (PAB), evaluates the three aspects of the system models. It automatically assigns the primitives of the signal flow models to the nodes of the hardware models as described in the previous chapter. The PAB generates two output files. One is written in the Multigraph Kernel Command Language. A small fraction of the generated file is shown in Figure 37.

```
tsk_attach "host" 1 -> tsk1
tsk_attach "host" 2 -> tsk2
tsk_attach "host" 3 -> tsk3
...
act_create "trigger_script" 2 2 AT_IFANY tsk7 -> Trigger0_act58
act_setcontext Trigger0_act58 6
act_create "split" 1 4 AT_IFALL tsk3 -> split_act51
act_setcontext split_act51 3
act_create "cfft" 1 1 AT_IFALL tsk3 -> fft_act53
act_setcontext fft_act53 7
...
dnd_create 10 DT_STREAM DS_DEALLOC tsk6 -> d1_dnd97
dnd_create 10 DT_STREAM DS_DEALLOC tsk5 -> d2_dnd30
dnd_create 10 DT_STREAM DS_DEALLOC tsk7 -> c00_dnd96
...
nde_connect c33_dnd25 Trigger3_act8 0 CM_NORMAL
nde_connect c20_dnd91 Trigger0_act25 0 CM_NORMAL
nde_connect c21_dnd24 Trigger1_act24 0 CM_NORMAL
...
dnd_enable L1_dnd15
dnd_enable L2_dnd11
dnd_enable L3_dnd14
...
act_activate split_act11
act_activate bpf_act10
act_activate Trigger0_act7
...
```

Figure 37 Generated Multigraph command file

First, one task for each hardware node is generated. Then one actornode for each signal flow primitive is created. Each actornode is assigned to a specific task, i.e. processor. Its script and number of input and output ports are specified. A unique name is assigned to each actor. Their contexts are set according to the input parameters of the signal flow primitives. Then the datanodes are generated. Their length, type, task, and unique name are specified as well. Next, the connections between datanodes and actornodes (and vice versa) are created. Finally, the datanodes and actornodes are enabled. That step starts up the application. The Multigraph command file for this 4-channel PSA consists of approximately 1000 lines.

The other output of the PAB is a hardware description file along with a list of communicating processor pairs. This information can be loaded into the hardware model analyzer tool, the Graphical Configuration Manager (GCM). The hardware topology of this configuration of the PSA inside the GCM is shown in Figure 38. The GCM compares the models to the actual hardware network, generates network loader information and deadlock-free message routing maps, as described in Chapter V.

The user interface of the PSA is shown in Figure 39. There are three main areas of the screen. The window in the upper left corner is used to set up the plots. The data source, scaling information, labels, and plot attributes can be specified. The window in the lower left corner is used to set screen configurations. The signal display window can be set to contain a single large window or any n by m configuration up to 4x4, using the graphical buttons. Screen and window configurations can be stored to disk.

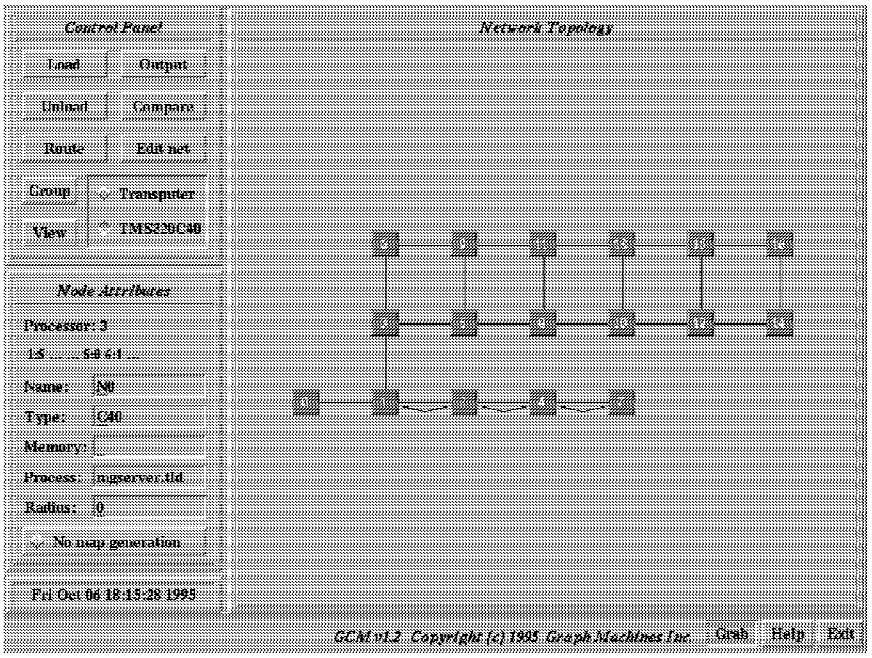

Figure 38 Hardware topology shown in GCM

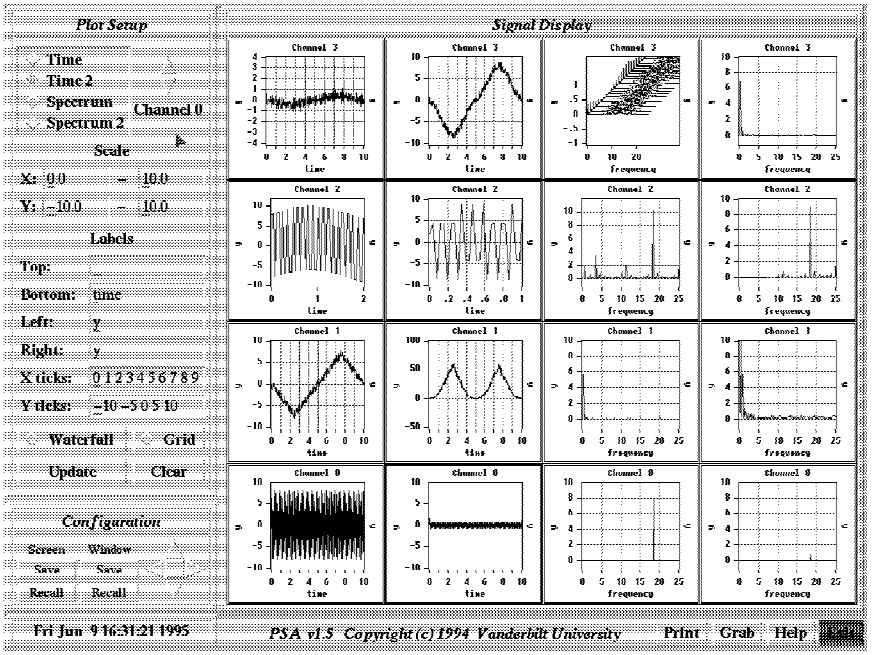

Figure 39 The Parallel Signal Analyzer

The signal display in Figure 39 is configured to contain a plot for every data source available in this configuration of the PSA. The lowest row of windows contains the plots corresponding to channel 0 (ch-I in the signal flow models). The leftmost window contains the original input data, a sweeping sine wave. The next window shows the data after the low-pass filter. The two rightmost windows display the corresponding spectrums.

CHAPTER IX

CONCLUSIONS

This dissertation has discussed the adoption of a model-integrated programming environment, the Multigraph Architecture, to the parallel instrumentation and signal processing domain. The target hardware architecture is distributed memory multiprocessors with flexible interconnection topology. While certain aspects of the problem have been addressed before, crucial parts were missing. The declarative modeling methodology of the Multigraph Architecture (MGA) has been extended by generative capabilities providing a powerful new paradigm. This dissertation has solved the previously open problem of deadlock-free wormhole routing in networks with arbitrary topologies. Process assignment has been solved automatically by a non-optimal, heuristic search procedure and by hardware topology synthesis. Both algorithms utilize the hierarchical structure of the system models providing a user-defined, application-specific heuristic. The next section summarizes the specific contributions of this work.

Contributions

Modeling Paradigm

Generative modeling. The Multigraph Architecture supports declarative modeling of complex systems. While it is a powerful technique, declarative modeling alone cannot manage the growing complexity of today's large-scale applications. Generative modeling

can augment the capabilities of declarative model building. With generative modeling, model components and structure can be specified using a programming language. It is most useful for the specification of repetitive and conditional model components. This dissertation introduced a novel technique to interface generative modeling to the declarative paradigm of the MGA. The mixed declarative (graphical) and generative (textual) models can be transformed to purely graphical representation for verification and debugging. Generative modeling in the Multigraph Architecture is still at a conceptual level and has not been implemented yet.

Assignment constraints. Explicit modeling of the assignment constraints, i.e. specific hardware requirements of software modules, provides a general approach toward the synthesis of real systems. Previous efforts hardcoded such constraints into the model interpreter mandating code change every time a new requirement arose. Providing an additional modeling aspect for this purpose greatly improves flexibility. Furthermore, this approach helps in automatic process assignment. The user can force the assignment of certain modules to hardware nodes that can execute them efficiently and avoid nodes where performance would suffer. Moreover, the user can guide the search through strategically placed assignment constraints. This can effectively cut the search space, speed up the procedure, and provide a better solution.

Model Interpretation and Analysis

Message routing. Previously, two approaches have been used for deadlock avoidance with deterministic wormhole routing. Virtual channels require hardware support not available on the target platforms of this work. Topology-based deadlock avoidance applies

tree-, mesh-, or hypercube-based architectures. This dissertation has identified a new class of graphs with deadlock-free minimal routing. In fact, it has been shown that the only inherently deadlock-free class of topologies are chordal graphs. Chordal graphs are the only class of graphs where any minimal routing is deadlock-free. However, they still do not provide enough flexibility for the domain. Partially connected routing suits the domain well because the system models contain the specific communication requirements of the application. However, this dissertation has shown that the minimal, partially connected, deadlock-free routing problem is NP-complete. Therefore, a non-minimal routing strategy has been introduced that guarantees deadlock-freedom and optimizes the message routes by minimizing message contention. The algorithm generates a deadlock-free routing in a chordal subgraph of the network. A message path from this routing is used only if there is no path with less cost in the whole network that do not result in a deadlocked configuration. The cost function is a combination of the average load on all communication channels and the load on the channel with the highest load representing a bottleneck in the system. The algorithm has polynomial time complexity. In most cases, the quality of the results are better than that of the optimal minimal algorithm, which does not guarantee deadlock-freedom and has exponential time complexity. This is possible because the new algorithm is not forced to use shortest paths and, therefore, it is able to avoid hot spots in the system.

Assignment. A heuristic search strategy has been proposed to assign signal flow primitives to hardware nodes. The algorithm follows the hierarchy of the signal flow models. It assigns models one level at a time, greatly reducing the search space. At any one level, the optimal solution is found by exhaustive search, if feasible. The algorithm

finds the global minimum at a low resolution, then increases the resolution and optimizes parts of the problem locally. Before starting a new level, the algorithm improves the solution globally by transformations at the new resolution. The algorithm and the applied cost function utilize all the available information in the models: the estimated load of signal flow components, the required bandwidth for communication between them, the performance of the hardware nodes and channels, the assignment constraints, and the hierarchy of the signal flow models. The latter makes the heuristic of the strategy tunable. The user can build knowledge of how the structure of the models will affect the assignment into the signal flow models.

Hardware topology synthesis. A heuristic processor interconnection network topology synthesis algorithm has been proposed as an alternative to the more traditional assignment strategy. The concept is to automatically synthesize a network topology matching that of the signal flow graph of the application. This approach is only feasible if the hardware topology is not preconfigured. The synthesis strategy is based on the signal flow model hierarchy. This cuts the search space and utilizes the structural information built into the models. Balancing the computational load and generating the topology to minimize interprocessor communication overhead are carried out in distinct phases of the algorithm.

Graphical Configuration Manager. The hardware model analysis tool of the Multigraph Architecture is the Graphical Configuration Manager (GCM). Interestingly, this was the first tool of the environment implemented. It has a built-in graphical editor enabling the use the GCM without the graphical model builder. Recently, the program was interfaced to the rest of the environment. The model interpreter transforms the hardware models into the input format of the GCM. The functions of the program are

essential in building parallel instrumentation systems on architectures with flexible topology. The GCM compares the models to the actual hardware diagnosing the network and validating the models. It generates network information files for different loaders. It also generates message routing maps for store-and-forward message routing. The new, non-minimal, partially connected, deadlock-free routing algorithm has also been implemented and integrated into the Graphical Configuration Manager.

Future Work

There are some aspects of the model-based parallel instrumentation application synthesis for distributed memory multiprocessors with flexible interconnection topology that warrant further investigation. Some of them go beyond the scope of this dissertation. The most important topics for future research are:

Implementation of generative modeling. The Multigraph Architecture with generative modeling capabilities supports a powerful paradigm by combining declarative (graphical) and generative (textual) modeling. The implementation does not pose any significant theoretical difficulties, but it involves extensive programming, including automatic model component and connection layout generation for the model builder user interface.

Topology classification for deadlock-free routing. Chordal graphs, as this dissertation has shown, are the only inherently deadlock-free class of topologies, i.e. any minimal routing is deadlock-free in these graphs only. What are the topologies with at least one minimal deadlock-free routing strategy? Two examples are meshes and hypercubes. Is there a general characterization of graphs with this property? Topology classification for deadlock-free routing is an open problem.

Faster routing and assignment. The message routing, assignment, and topology synthesis algorithms introduced in this dissertation produce high quality results. While their time complexity is polynomial, their speed needs improvement. The hardware topology synthesis algorithm needs to be generalized for heterogeneous processor sets. It must be modified to satisfy the assignment constraints as well.

Debugging support. Unified debugging support is still missing from the MGA. A process involving the system models in the debugging is desirable, especially for parallel processing.

REFERENCES

1. "A Routing Scheme for TMS320C40-Based Systems," Application Note, Texas Instruments, available on the TI Bulletin Board

2. *CRAY T3D System Architecture Overview Manual,* Cray Research Inc., available on the WWW, 1994

3. *Express 3.0 User's Guide,* Parasoft Corporation, 1990

4. *The Starburst Programming Environment*, Loral Rolm Computer Systems, Demonstration at the DSP World Expo, Dallas, TX, Oct. 1994

5. Abbott, B. : *"Model-Based Automatic Software Synthesis,"* Ph. D. Dissertation, Dept. of Electrical Engineering, Vanderbilt University, Nashville, TN, 1994

6. Abbott, B. et al.: "Model-Based Software Synthesis," *IEEE Software*, Vol. 10, No. 3, pp. 42-52, May 1993

7. Abbott, B., Ledeczi, A.: "TICK: a TMS320C40 Utility Program," *Proc. of the International Conference on Signal Processing Applications and Technology*, Dallas, Texas, 1994

8. Ahmed, S., Carriero, N., Gelernter, D.: "A Program Building Tool for Parallel Applications," available on the WWW, Yale University, 1993

9. Babaoglu, O. et al.: "Mapping Parallel Computations on to Distributed Systems in Paralex," *Tech. Report UBLCS-92-1*, *University of Bologna*, Jan. 1992

10. Bapty, T. et al.: "Parallel Turbine Engine Instrumentation System," *Proc. of Computing in Aerospace 9*, pp. 423-433, San Diego, CA, 1993

11. Bapty, T.: "*Model-Based Synthesis of Parallel Real-Time Systems*" Ph. D. Dissertation, Vanderbilt University, 1995

12. Beguelin, A. et al.: "Visualization and Debugging in a Heterogeneous Environment," *IEEE Computer,* Vol. 26, No. 6, pp. 88-95, June 1993

13. Biegl, C.: *"Design and Implementation of an Execution Environment for Knowledge-Based Systems,"* Ph. D. Dissertation, Dept. of Electrical Engineering, Vanderbilt University, Nashville, TN, 1988

14. Bokhari, S. H.: "On the Mapping Problem," *IEEE Trans. on Computers,* Vol. C-30, No. 3, pp. 207-214, March 1981

15. Bokhari, S. H.: *Assignment Problems in Parallel and Distributed Computing*, Kluwer Academic Publishers, 1987

16. Bollinger, S. W., Midkiff, S. F.: "Heuristic Technique for Processor and Link Assignment in Multicomputers," *IEEE Trans. on Computers*, pp. 325-333, March 1991

17. Childers, C. A. et al.: "The Multigraph Modeling Tool," *Proc. of the 7th International Conference on Parallel and Distributed Systems*, Las Vegas, Nevada, 1994

18. Dally, W. J., Seitz, C. L.: "Deadlock-Free Message Routing in Multiprocessor Interconnection Networks," *IEEE Trans. on Computers*, Vol. C-36, No. 5, pp. 547-553, May 1987

19. Dally, W. J., Seitz, C. L.: "The Torus Routing Chip," *Journal of Distributed Computing*, Vol. 1, No. 3, pp. 187-196, 1986

20. Dally, W. J.: "Network and Processor Architecture for Message-Driven Computers," in Suaya, r., Birtwistle, G. (eds.) *VLSI and Parallel Computation*, Morgan Kaufmann Publishers, San Mateo, CA, 1990

21. Dally, W. J.: "Virtual-Channel Flow Control," *IEEE Trans. on Parallel and Distributed Systems*, Vol. 3, No. 2, pp. 194-205, March 1992

22. Dixit-Radiya, V. A., Panda, D. K.: "Task Assignment on Distributed-Memory Systems with Adaptive Wormhole Routing," *Tech. Report OSU-CISRC-4/93-TR18*, The Ohio State University, Feb. 1994

23. Duato, J.: "A Necessary and Sufficient Condition for Deadlock-Free Wormhole Routing," *Tech. Report*, Universidad Politecnica de Valencia, October 1993

24. Duato, J.: "A New Theory of Deadlock-Free Adaptive Routing in Wormhole Networks," *IEEE Trans. on Parallel and Distributed Systems*, Vol. 4, No. 12, pp. 1320-1331, December 1993

25. Fernandez-Baca, D.: "Allocating Modules to Processors in a Distributed System," *IEEE Trans. on Software Engineering*, Vol. 15, No. 11, pp. 1427-1435, Nov. 1989

26. Franke, H.: "*Programming Environment for Model-Based Systems*" Ph. D. Dissertation, Vanderbilt University, 1992

27. Garey, M., Johnson, D.: *Computers and Intractability: a Guide to the Theory of NP-Completeness*, Freeman, 1979

28. Goldberg, D. E.: *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley, Reading, MA., 1989

29. Harel, D.: "Biting the Silver Bullet," *Computer*, pp. 8-20, Jan. 1992

30. Hillis, W. D.: *The Connection Machine*, The MIT Press, Cambridge, MA, 1985

31. Hillson, R.: "Support Tools for the Processing Graph Method," *Proc. of the International Conference on Signal Processing Applications and Technology*, pp. 756-761, Dallas, TX, Oct. 1994

32. Holland, J. H.: *Adaptation in Natural and Artificial Systems*, The University of Michigan Press, Ann Arbor, MI, 1975

33. Karsai, G.: "A Visual Programming Environment for Domain Specific Model-Based Programming," *Computer*, March 1995

34. Kirkpatrick, S., Gelatt, C. D., Vecchi, M. P.: "Optimization by Simulated Annealing," *Science*, Vol. 220, pp. 671-680, May 1983

35. Ledeczi, A., Abbott, B.: "Parallel Architectures with Flexible Topology," *Proc. of the Scalable High Performance Computing Conference*, pp. 271-276, May 1994

36. Lee, I., Smitley, D.: "A Synthesis Algorithm for Reconfigurable Interconnection Networks," *IEEE Trans. on Computers*, Vol. C-37, pp. 691-699, June 1988

37. Lee, S., Aggarwal, J. K.: "A Mapping Strategy for Parallel Processing," *IEEE Trans. on Computers*, pp. 433-442, Apr. 1987

38. Leiserson, C. E. et al.: "The Network Architecture of the Connection Machine CM-5," available on the WWW, 1993

39. Lin, X., McKinley, P. K., Ni, L. M.: "The Message Flow Model for Routing in Wormhole-Routed Networks," *Tech. Report MSU-CPS-ACS-78*, Michigan State University, January 1993

40. Ma, P. R., Lee, E. Y. S., Tsuchiya, M.: "A Task Allocation Model for Distributed Computing Systems," *IEEE Trans. on Computers*, Vol. C-31, No. 1, pp. 41-47, Jan. 1982

41. May, M. D., Thompson, P. W., Welch, P. H. (eds.): *Networks, Routers and Transputers: Function, Performance, and Applications*, Inmos Limited, 1993

42. Ni, L. M., McKinley, P. K.: "A Survey of Wormhole Routing Techniques in Direct Networks," *Computer*, Vol. 26, No. 2, pp. 62-76, Feb. 1993

43. Ni, L. M., Panda, D. K.: "Sea of Interconnection Networks: What's Your Choice," *Report of the ICCP '94 Panel Discussion*, available on the WWW, Aug. 1994

44. Noakes, M. D., Wallach, D. A., Dally, W. J.: "The J-Machine Multicomputer: An Architectural Evaluation," *Proc. of the 20th International Symposium on Computer Architecture*, May 1993

45. Perry, D.: *VHDL*, McGraw-Hill, 1991

46. Reed, D. A., Fujimoto, R. M.: *Multicomputer Networks: Message-Based Parallel Processing*, MIT Press, Cambridge, Mass., 1987

47. Schwiebert, L., Jayasimha, D. N., Tseng, Y.: "A Necessary and Sufficient Condition for Deadlock-Free Wormhole Routing," available on the WWW

48. Schwiebert, L., Jayasimha, D. N.: "Optimal Fully Adaptive Wormhole Routing for Meshes," *Proc. of Supercomputing '93*, pp. 782-791, 1993

49. Shen, C., Tsai W.: "A Graph Matching Approach to Optimal Task Assignment in Distributed Computing Systems Using a Minimax Criterion," *IEEE Trans. on Computers*, Vol. C-34, No. 3, pp. 197-203, March 1985

50. Shen, H.: "Self-adjusting Mapping: a Heuristic Mapping Algorithm for Mapping Parallel Programs on to Transputer Networks, " *The Computer Journal*, pp. 71-80, Feb. 1992

51. Siljak, D.: *Decentralized Control of Complex Systems*, Academic Press, 1991

52. Simar, R. et al.: "Floating-Point Processors Join Forces in Parallel Processing Architectures," *IEEE Micro*, pp. 60-69, August 1992

53. Stone, H. S.: "Multiprocessor Scheduling with the Aid of Network Flow Algorithms," *IEEE Trans. on Software Engineering*, Vol. SE-3, No. 1, pp. 85-93, Jan. 1977

54. Stunkel, C. B., et al.: "The SP2 High-Performance Switch," *IBM Systems Journal*, vol. 34, no. 2, 1995

55. Talbi, E., Muntean, T.: "General Heuristics for the Mapping Problem," *Proc. of the World Transputer Congress*, IOS Press, Aachen, Germany, Sep. 1993

56. Turcotte, L. H.: "A Survey of Software Environments for Exploiting Networked Computing Resources," *Tech. Report, MSU-EIRS-ERC-93-2*, Mississippi State University, Feb. 1993