MODEL–INTEGRATED PROGRAM SYNTHESIS FOR REAL–TIME IMAGE PROCESSING

By

Michael S. Moore

Dissertation

Submitted to the Faculty of the

Graduate School of Vanderbilt University

in partial fulfillment of the requirements

for the degree of

DOCTOR OF PHILOSOPHY

in

ELECTRICAL ENGINEERING

May 1997

Nashville, Tennessee

Approved: Date:

_____ _____

_____ _____

_____ _____

_____ _____

_____ _____

ACKNOWLEDGEMENTS

I must acknowledge the many people who have contributed either directly or indirectly to this work. The first of these is my wife Shannon, to whom I am forever indebted. It is no overstatement to say that without her love, her sense of humor, and her disarming, light–hearted presence, I would not have gained the self confidence and perspective I needed to finish this work. I would also like to thank my family for their years of support.

I have special graditude for my advisors, Janos Sztipanovits and Gabor Karsai, Dr. Sztipanovits for his vision and motivation, his perpetual optimism, and his belief in my abilities, and Dr. Karsai for his technical guidance and his seemingly endless patience.

Thanks also to my other committe members, Alan Peters, Benoit Dawant, and Bob Galloway for their helpful suggestions and advice.

Special thanks go to Stephan Rosner for his assistance in designing many of the figures both in this document and in the defense presentation slides, and also for working ridiculously late (I always had someone to talk to).

Thanks to the others in the Measurment and Computing Systems Lab, Ted Bapty, Csaba Beigl, Richard Davis, Akos Ledeczi, Amit Misra, Greg Nordstrom, and Jason Scott for their input and for putting up with me and my silly sense of humor for the past several years.

I appreciate the organizations which provided financial support for this research: Vanderbilt University's Electrical and Computer Engineering department for providing teaching assistantships, and the US Air Force and the Air Force Office of Scientific

Research for research grants. In particular, I thank Jim Nichols of the Arnold Engineering Develpment Center both for sponsoring and participating in the project.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER I

INTRODUCTION

Digital imaging application require huge computational performance due to the large data sets involved. Applications such as robotics, animate vision, autonomous vehicle control, and on–line video processing require sequences of images to be processed in *real–time*. Image sequences of normal resolution (640 x 480 pixels) and standard frame rate (30 $\frac{frames}{sec}$) with 256–level gray scale ($8\frac{bits}{pixel}$) represent a data rate of 8.8 Megabytes per second. A color sequence ($24\frac{bits}{pixel}$) at the same pixel resolution produces $26.4\frac{Mbytes}{sec}$. Typical applications require on the order of hundreds or even thousands of operations per pixel in order to enhance, segment, and extract features from the image sequence [57]. This translates into tens of Giga–operations per second. It has been estimated that future applications, such as dynamic scene interpretation, may require on the order of hundreds of Giga–operations per second [56]. Hardware architectures consisting of a single general–purpose processor are incapable of delivering this level computational performance. Even smaller problems, such as video enhancement, require hundreds of Mega–operations per second. For these reasons, most applications in which real–time imaging has been successfully applied have usually employed custom hardware designed to perform fixed sets of specific image processing algorithms. Although such specialized hardware solutions have met computational requirements of some real–time image processing applications, there are many drawbacks to this approach. Hardware implementations are expensive, and

real–time performance is achieved by sacrificing end–user programmability and flexibility. Also, the scope of success has been limited to those applications in which the computational needs could be accommodated by the fixed and limited capabilities of the available hardware.

The need has been stated for research efforts targeted toward producing real–time image processing support for recent applications such as remote command and control, High Definition Television (HDTV), virtual reality modeling, military target tracking, and rapid image identification [28]. These applications require more flexible and scalable real–time image processing solutions than are currently available. The requirement of high performance and scalable hardware, however, should not occlude the need for flexibility and ease of use. A hardware architecture with adequate performance will not be practical without high–level programming environments and tools designed for building image processing applications. This fact is well demonstrated by noting that research in parallel processing has produced many architectures which boast incredible numerical benchmarks. However, since parallel machines are inherently difficult to program, it is more rare to find applications in which a programmer without parallel processing expertise has successfully and cost–effectively exploited the technology in a real–world application. A parallel programming environment which insulates the user from the underlying parallel implementation details is crucial.

In this dissertation, I show that by using Model–Integrated Program Synthesis (MIPS) and taking advantage of the natural parallelism present in image processing computations, a high–level, graphics–based, parallel programming interface is possible. With this approach, real–time parallel implementations of image processing

computations can be automatically synthesized, effectively rendering the complex details inherent to parallel software development transparent to the programmer, enabling the cost–effective exploitation of parallel hardware architectures for building more flexible and powerful real–time imaging systems than are available today.

I examine the inherent parallelisms present in image processing algorithms, and formally define a quantitatively specifiable description of image processing algorithms that is sufficient to determine the ways in which a particular algorithm can be parallelized invariantly (i.e. without changing the computed results). This information is also necessary to quantify the performance which will result as the parallelism is scaled and the computation is mapped to the underlying resources. Note that the algorithm description is necessary, but not sufficient, to determine the resulting performance of the parallel implementation. The performance depends not only upon the computations, but also upon the qualities of the target hardware and run–time resources, and the mapping of the parallelized computation to those resources. The following interrelated problems must be solved simultaneously:

- Decomposing computations: The computation must be broken down into sub–computations which can be performed concurrently. This process involves decomposing the problem (parallelization) and scaling the parallelism to meet the requirements.

- Mapping computations to resources: The general mapping problem is to decompose and allocate the computation blocks to the available resources. This involves assignment of processes to processors, load balancing, and communication routing.

- Forming performance models: Accurate models of the performance that will result from a particular decomposition and mapping of a computation to the available resources are needed in order to predict whether or not the performance goals will be met.

- Supporting parallel execution: Run–time support for parallel scheduling, communication, and synchronization must be provided.

Note that in most research–oriented studies of parallel or real–time systems, the approaches attempt to solve only one, or possibly a few of these problems. However, the implementation of a real–world embedded imaging system requires these issues to be addressed simultaneously.

I show that, with key knowledge about the behavior of the run–time system and the processor architecture on which the hardware network is based, it is possible to accomplish this simultaneous solution, automatically generate mappings of parallel computations onto the resources and build accurate performance models for the resulting implementation. As proof of this concept, I have developed MIRTIS (Model Integrated Real–Time Image Processing System). MIRTIS employs the Multigraph Architecture (MGA), a framework and set of tools for building MIPS systems, to generate image processing applications which run under the control of a parallel run–time kernel on a network of Texas Instruments TMS320C40 DSPs (C40s). MIRTIS is configured by building graphical models representing (1) the computations to be performed, (2) the C40 network configuration, and (3) the performance constraints of the target application. The MIRTIS *model interpreter* reads in the graphical models and automatically determines the feasibility of a real–time implementation, parallelizes,

scales, and maps the given computation to the resources such that the real–time constraints will be met, and configures a graphical user interface with which the user can adjust processing parameters dynamically while the system runs.

MIRTIS is a clear example of how parallel real–time image processing systems can be built which are cost–effectively programmable, flexible, scalable, and built from Commercial Off–The–Shelf (COTS) components. Although the mapping algorithm makes assumptions about the underlying hardware architecture and run–time system, the formal specifications of applications in terms of models provides a high level of architectural independence. Much of the system can be reused when the target hardware platform evolves with the availability of faster and cheaper hardware.

CHAPTER II

BACKGROUND

Computer Vision and Image Processing

Vision is the ability to form images from illumination and derive from these images information about our surroundings. The vision process is complex and not well understood, so the success in constructing machines which employ vision effectively has been limited. However, the potential power and advantages of machine vision drive many research efforts in application domains from military target tracking to non–invasive medicine.

Computer vision systems take in data from *image* sensors, which may sense visible, infrared, or even magnetic radiation, and attempt to construct some model of the surroundings which may be used in formulating controls over the environment and/or presented to a human for interpretation. The approach used in computer vision can be roughly broken down into two sequential steps: (1) *image analysis*, or *early vision*, and (2) *scene analysis* [23]. Image analysis usually involves massaging the image data to reduce noise, enhance detail, or manipulate contrast, then locating features such as edges, corners, and surfaces, and finally identifying patterns and objects. Scene analysis involves making inferences of relationships between objects and deriving understanding of the environment from the scene (refer to figure 1).

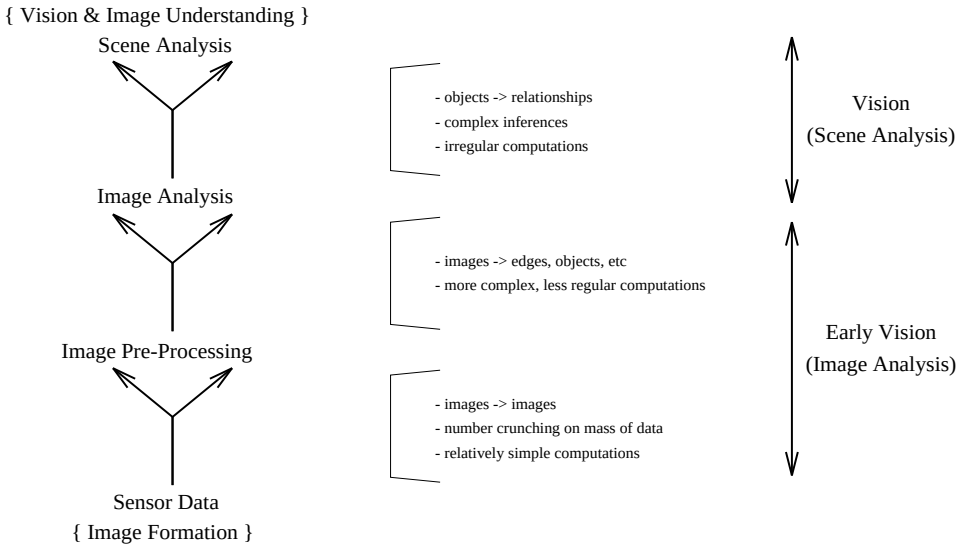

Figure 1: The Computer Vision Process

Image Processing Algorithms

The early vision steps are largely made up of algorithms which operate on image data and produce images, transformations of images, or simple data structures describing images. These are *image processing* algorithms[1], sometimes referred to as *low level vision* [13] because the role they play in vision is to pre–process images for transformation into symbolic data (edges, connected components, etc). Image processing algorithms usually work toward one or more of the following goals:

- noise reduction
- contrast enhancement
- detail enhancement
- edge detection
- segmentation (identification of objects)

[1]Note that in this dissertation, only *digital* systems are considered, so an image implicitly means a *digitized* image and *image processing* implicitly means *digital image processing*

- mensuration (measurement of features)
- depth approximation (3-D vision)
- motion estimation
- object location

Table 1 shows some classes of image processing algorithms which were taken from [38].

Algorithm Class	Example Algorithms
Adaptive Filters	Adaptive Min Mean Squared Error Filter (MMSE Filter)
Graphics Algorithms	Dither, Morph, Rotate/Translate, Warp, Zoom
Histogram Techniques	Histogram Stretch, Histogram Equalize, Local Histogram Equalization
Mensuration Algorithms	Area, Centroid, Moments, Perimeter
Morphological Operations	Opening, Closing, Dilation, Erosion, Outline, Skeleton
Nonlinear Filters	Maximum Filter, Median Filter, Geometric Mean Filter
Pixel Operations	Scale/Offset, Gamma Correct
Segmentation	Line Detection, Threshold
Spatial Filters (Linear Convolution)	Laplacian, Gaussian, Gradient, Sobel, High Pass, Low Pass
Spatial Frequency Filters	Homomorphic Filter, Least Mean Squares Filter, Wiener Filter
Transformations	Discrete Cosine Transform (DCT), Fast Fourier Transform (FFT), Hough Transform

Table 1: Some Classes of Image Processing Algorithms

The mathematical properties of these algorithms are well understood, and detailed descriptions and even source code for implementation can be found in most image processing texts or reference books. For example, see [38]

or [23]. The focus of this work is not the details of particular image processing algorithms, but the intrinsic qualities of image processing algorithms affecting the ways in which they can be implemented.

Higher Dimensional Image Processing Algorithms

The aforementioned image processing algorithms operate only in the two spatial dimensions of the image (2-D image plane). In addition to 2-D image processing operations, there are also useful algorithms which operate on groups of images simultaneously, interpreting them as either time sequences or volumes.

In video processing, the images are organized into a data model containing two spatial coordinates plus at third time coordinate.

$$I_0(x,y), I_1(x,y), ..., I_t(x,y) \quad \longrightarrow \quad V(x,y,t) \tag{1}$$

It is common when processing video to perform calculations which operate along the time domain. In other words, an output pixel in the current frame is calculated based on the pixel value at that position in several consecutive frames.

$$O_{(x,y)}(t) = F(I_{(x,y)}(t), I_{(x,y)}(t-1), I_{(x,y)}(t-2), ...) \tag{2}$$

For example, ensemble averaging (averaging several frames together) is used for noise reduction, and single time step differencing is a simple method of motion detection. The equations for ensemble averaging and differencing are given below.

$$Ave_{x,y}(t) = \frac{\sum_{k=0}^{N} I_{x,y}(t-k)}{N}$$

$$Diff_{x,y}(t) = |I_{x,y}(t) - I_{x,y}(t-1)|$$

Other techniques, such as time–based linear filtering (convolution only in the time domain) are also commonly used.

Some video algorithms operate both in the 2–D image plane and in the time domain simultaneously.

$$O(x, y, t) = F(I(x, y, t), I(x, y, t-1), I(x, y, t-2), ...) \tag{3}$$

Examples of such algorithms are optical flow, 3–D linear filtering, 3–D morphology [41], and image warping.

It is also possible to interpret multiple images as a volume by using the image index as a third spatial coordinate.

$$I_0(x, y), I_1(x, y), ... \longrightarrow V(x, y, z) \tag{4}$$

For example, this model is used in medical imaging, where several 2-D scans are "stacked" for volumetric processing or visualization. 3–D algorithms, which operate in all three spatial dimensions, are then used to process and manipulate the data.

General Image Processing Algorithm Model

Image processing algorithms operate on images or transformations of images. In order to accommodate algorithms which operate on single images, sequences of images, or volumes made up of images, I will define a general model of an image processing algorithm to be used in this work. The model development concentrates not upon algorithmic issues, but upon the way in which an image processing algorithm generally forms a results sequence by accessing data from input image sequences and past states of the results sequence.

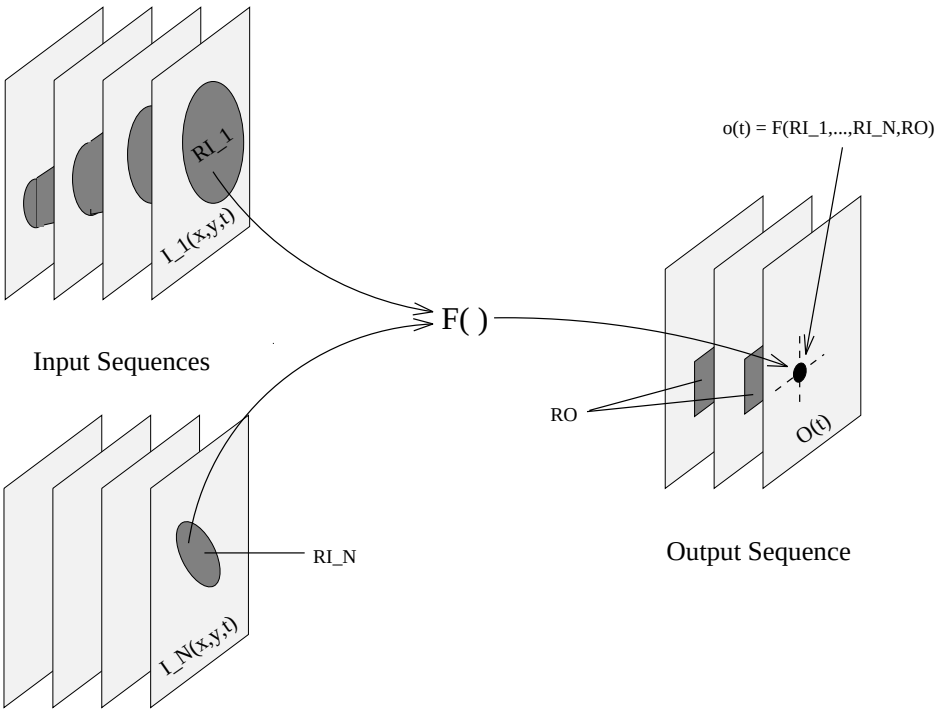

Figure 2: The Image Processing Algorithm Model

Consider image processing algorithm $F()$ which takes N input image sequences $I_1(x,y,t)...I_N(x,y,t)$ and produces an output data structure sequence $O(t)$ (see figure 2). Select a particular data element of $O(t)$, say $o(t)$. (Note that if $O(t)$ is an image sequence, then this is a pixel in the output at time t, and $o(t) = O_{(x,y)}(t)$.) $o(t)$ is computed from some set of data made up of pixels from the inputs sequences, and elements from past values of the output data structure. The corresponding pixels required from the kth input sequence I_k in order to calculate $o(t)$ necessarily lie in a finite set of pixel locations which together form a *region* $\hat{R}_{I_k}$ inside sequence $I_k(t)$.

$$\hat{R}_{I_k} = \left\{ \vec{P} = (p,q,r) \;\mid\; o(t) \longleftarrow I_k(p,q,r) \right\} \tag{5}$$

where $\vec{P} = (p,q,r)$ is the pixel location designated by row p, col q, of frame r. The arrow pointing from the input location to $o(t)$ specifies that $o(t)$ *depends upon*

$I_k(p,q,r)$, which means that in order to compute $o(t)$, the algorithm requires direct knowledge of the pixel value at location $\vec{P}$ in I_k.

The union of the required regions from the input sequences with the required region from the past values of the output sequence $O(t)$ forms the *total data dependency set* of output data element $o(t)$, $\hat{D}_{o(t)}$.

$$\hat{D}_{o(t)} = \left\{ \bigcup_{k=1}^{N} \hat{R}_{I_k} \right\} \bigcup \hat{R}_O \tag{6}$$

Image Processing Algorithm Definition:

For the purposes of this dissertation, a function $F()$ operating on N input image sequences $I_1(x,y,t)...I_N(x,y,t)$ and computing output data structure sequence $O(t)$ is an image processing algorithm if and only if

- $F()$ is causal. For each output data element $o(t) \in O(t)$

  $$r \leq t \quad for\ each \quad \vec{P} = (p,q,r) \in \hat{R}_{I_k} \quad k = 1...N$$

  i.e. The data dependency set for a current output data element $o(t)$, $\hat{D}_{o(t)}$, contains only input data locations from the current and past input images.

- $F()$ is non–recursive. For each output data element $o(t) \in O(t)$

  $$!\exists \vec{P} \mid \vec{P} \in O(t)\ and\ \vec{P} \in \hat{D}_{o(t)}$$

  i.e. The data dependency set for a current output data element $o(t)$, $\hat{R}_{o(t)}$, contains no data locations in the current output structure $O(t)$.

- For each output data element $o(t) \in O(t)$, the total data dependency set $\hat{D}_{o(t)}$ required to compute $o(t)$ is finite, and a bounding set can be determined a priori

to the start of computation. The computations need not be fixed, as long as both the amount of computations and the regions required to compute each output have specifiable upper bounds.

This definition is quite non–restrictive, especially in light of the types of computations which are usually performed in low–to–mid level vision. Recursive algorithms are not commonplace, and non–causal algorithms are not realistically implementable in a real–time vision system.

Specialization of Image Processing Algorithms

Image processing is a highly developed research area, with many current efforts on both theoretical and applied fronts. There are conferences and journals dedicated exclusively to image processing, in which theorists and application engineers present general techniques as well as specialized algorithms for very specific applications. For example, in an issue of "Optical Engineering", a researcher presented an algorithm developed specifically for automated recognition of raised characters on rubber tires. An edge detection algorithm was developed which takes advantage of the properties the developer observed in the data sets. This specially designed algorithm was shown to be much more effective for the application than standard edge detectors, such as the Sobel, LOG, and morphological edge operators[22]. However, on general data sets, it is doubtful that the algorithm would be very useful. The notable point is that the level of success achieved in applying image processing to real applications is often directly related to the development of specialized algorithms by the people with intimate knowledge of the application. The most successful solutions are obtained when the developer can interactively rapid–prototype and experiment with algorithms

to come up the best solution for the given problem. This explains the wide use of development environments such as Khoros, PVWave, and Matlab, which provide high–level, interactive interfaces and promote rapid prototyping and experimentation with non–standard computational techniques.

> *High–level development and rapid–prototyping interfaces play an integral part in the image processing algorithm development cycle, and should be considered a non–expendable option in any suitable image processing programming environment.*

Real–Time Image Processing

A *Real–Time* system is one which, due to interaction with its environment, must produce outputs which are not only numerically correct, but which also meet timing constraints. Such systems are said to be *embedded* into their environments. The relevant environmental interactions for image processing systems are receiving data from sensors or other systems, and outputting data to displays, plots, devices, or other systems, which may apply controls to the surroundings directly (e.g. a vision system might generate controls for a robotic arm).

Relevant Timing Constraints

Timing constraints are introduced by these interactions. The system may be required to service the input devices as quickly as the data is produced, produce outputs at a certain rate, or to produce an output from each particular input within a constrained amount of time. The two relevant types of temporal constraints in real–time image processing systems are:

- *throughput*: the net system data rate. The rate at which results are produced by the system, usually quantified in $\frac{frames}{sec}$.

- *latency*: the total time between the sensing of a particular image and the results of that image leaving the system, usually quantified in *seconds*, and sometimes in *frames*.

  Either or both of these temporal constraints may be required by a particular real–time imaging application, so a general system must have methods of controlling throughput and latency.

For example, in real–time video enhancement throughput is important, but latency may not be. It is more critical that all of the data be processed with no down–sampling. However, in robotics latency is more important. The determination of the robot's action must be made as soon as possible after the visual sensing, and the action must be performed before the next sensing.

Real–Time imaging applications can have *hard real–time*, or *soft real–time* constraints. A hard real–time approach requires the absolute guarantee that the specified constraints will be met. If they are not met exactly, the system fails. In a soft real–time application, it may be acceptable to operate below the specified performance if the constraints cannot be met. In this case a *best effort* approach should be taken.

Since both hard and soft real–time image processing applications exist, a general approach must include both hard and soft real–time constraints.

In order to pursue a deeper understanding of the nature of these timing constraints, I examine various applications which use embedded imaging systems in the next section.

Real–Time Image Processing Applications

A major application of real–time image processing is video enhancement. In addition to real–time video enhancement, there are several contemporary, specialized applications requiring real–time image processing that were discussed by Laplante [28].

- *Video Enhancement:* Image sequences are filtered, cleaned, and otherwise enhanced to increase the signal to noise ratio, enhance details, detect phenomena, or merely improve the visual appearance. Image quality and throughput are both very important.

- *Remote command and control:* Images are transmitted over a channel to provide visual information at a remote site, which may be used for remote control of a device or vehicle. The quality of the images need only be good enough for the control process, but both throughput and latency can be crucial.

- *Broadcast and multimedia communications (eg. HDTV):* Image data is transmitted over a channel, and both image quality and throughput are of highest importance.

- *High speed modeling:* Images are generated and displayed sufficiently fast by a simulation or by a virtual reality model. Image quality and throughput are of highest importance.

- *Rapid image identification:* These applications include military target tracking and threat identification. Both image quality, latency, and throughput are crucial.

– *Medical imaging:* Medical applications may involve both rapid image identification and real–time image synthesis. Examples are mammography, ultrasound, MRI, and CT. The real–time nature of these applications varies.

– *Robotics:* Visual information is used in calculating controls for a robot. Both throughput and latency can be crucial, but latency is more often the critical constraint.

An Example Real–Time Image Processing Application:

Here, a specific example application will be explained in detail. The application involves the instrumentation of turbine engine tests, and falls under the category video enhancement. See figure 3.

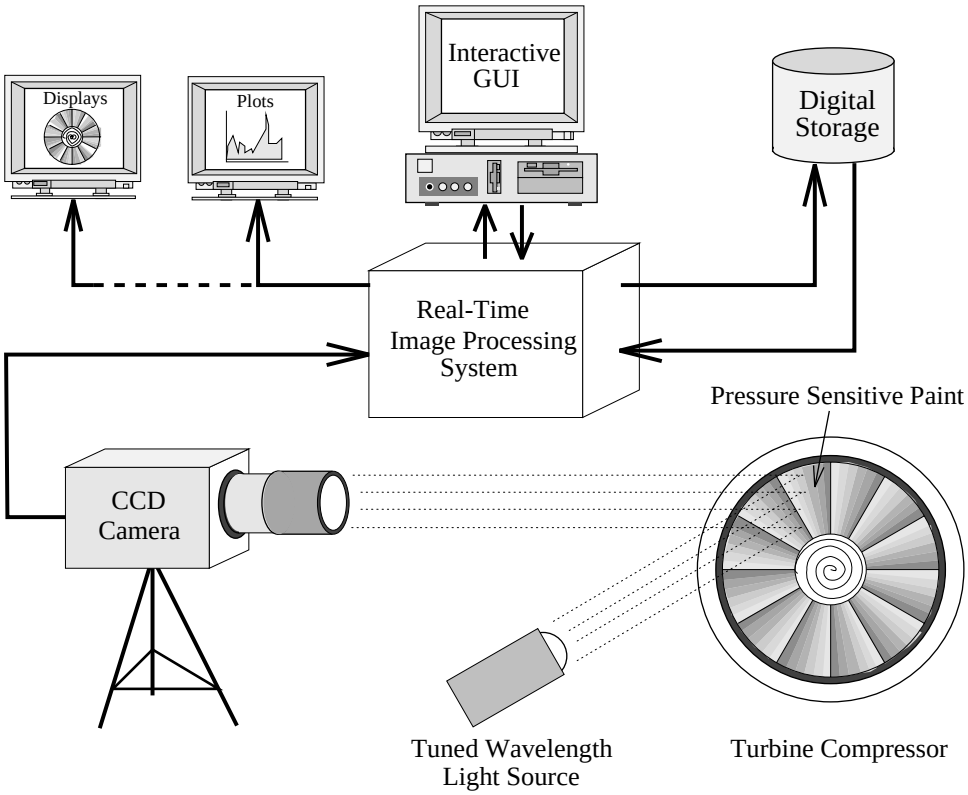

Figure 3: An Example RTIP Application

In analyzing and testing turbine engines, information about the physical phenomena occurring within the internal components of the running engine is needed in order to evaluate the performance, health, reliability, and efficiency of the engine. Non–invasive instrumentation of a turbine engine, however, is difficult. While measuring structural stresses with strain gauges has been successfully practiced, until recently there was no method of measuring fluid pressure patterns on the moving turbine blades.

Recent efforts have begun using *Pressure Sensitive Paint* (PSP) to effectively instrument the turbine blade surfaces to obtain this valuable data. Pressure sensitive paint[2] is a fluorescent paint which, when excited with the correct wavelength of light, luminesces with a varying intensity which is functionally dependent on the oxygen depletion at the paint's surface. By (1) applying PSP to the turbine blades of jet engine's compressor stage, (2) exciting the paint with a tuned wavelength light source, and (3) strobing a CCD array camera so that it is synchronized with the motion of the blade, visual data representing the pressure field on the moving blade can be obtained (see figure 3) . The pressure field data must be digitally processed and stored in real–time. The reasons that a real–time system is needed are: (1) the contrast produced by the paint is so low that it is difficult to discern without enhancement, and would be lost in a normal analog recording of the video, and (2) the plots and enhanced data could be useful during the test. The processing system is needed to enhance the images, extract information from the sequence, produce plots and image displays, and provide real–time disk storage. The types of computations to be performed would include frame–based noise reduction, contrast enhancement, time gradient detection,

[2] *Pressure sensitive paint* is actually a misnomer. It should be called oxygen sensitive paint.

multiple view registration and/or warping, spatial frequency analysis, and lossless compression.

The sensor output(s) from a high quality visible light camera would be digitized into frames of 512x512 16–bit pixels at a rate of 30 frames per second, producing $15\frac{Mbytes}{sec}$ of data. Supporting the 30 frames per second frame-rate is a major constraint, since short–lived phenomena are important to the turbine engine analysts. Thus, computational throughput is a hard real–time constraint. Latency, however, could be relaxed if necessary, since no direct control over the engine will be performed by the system.

Challenges of Real–Time Image Processing

In this section I will examine what makes general real–time image processing a difficult problem. Real–time systems are inherently complex and difficult to implement. Specifying, proving the correctness of, implementing, scheduling, assigning, and integrating real–time systems are unsolved problems in general. For instance, the general assignment problem is NP–complete. For these reasons, the study of real–time systems is a significant branch of current research.

> *To make the problems of specifying and guaranteeing timing constraints for real–time image processing tractable, it is mandatory to build a real–time system specially for image processing which takes advantage of knowledge about the image processing problem domain.*

Real–time imaging applications are characterizable as requiring the following:

- high I/O bandwidth

- high computational throughput
- low latency
- predictable and controllable performance behavior
- interaction with the environment through sensors

I/O Bandwidth:

Image processing algorithms are by nature computational intensive. This is due in part to the large size of image data. Even transferring data in and out of a system at the required rates is a challenge. Table 2 shows data rates resulting from some common video digitization resolutions and frame rates. In particular, consider the following example. A standard video source[3] produces $30\frac{frames}{sec}$. Digitizing the frames into 640x480, $8\frac{bits}{pixel}$ images produces $8.8\frac{Mbytes}{sec}$ of data. This I/O rate resulting from moderate digitization resolution and only 256 color pixels already surpasses the capabilities of many standard I/O devices, such as hard drives, Ethernet connections, and busses.

> *Note that, although one may argue that there are high performance busses, drives, and communication adapters available which will support* $8.8\frac{Mbytes}{sec}$, $64\frac{Mbytes}{sec}$, *or more, the point is that real–time imaging data rates are large, which requires special attention to the communication and storage hardware bandwidths.*

[3] $30\frac{frames}{sec}$ is now the standard, however, there are sensors which produce $60\frac{frames}{sec}$ or more.

resolution colsxrows	data depth $\frac{bits}{pixel}$	frame rate $\frac{frames}{sec}$	data rate $\frac{Mbytes}{sec}$	$5x5$conv $\frac{Mops}{sec}$
$256x256$	8	30	1.9	98
$512x480$	8	30	7.0	370
$640x480$	8	30	8.8	460
$1024x1024$	8	30	30	1,600
$256x256$	16	30	3.7	98
$512x480$	16	30	14	370
$640x480$	16	30	18	460
$1024x1024$	24	30	60	1,600
$256x256$	24	30	5.6	98
$512x480$	24	30	21	370
$640x480$	24	30	26	460
$1024x1024$	24	30	90	1,600

Table 2: Throughput Requirements Of Real–Time Video Processing

Computational Throughput:

Actually performing mathematical operations at the necessary rates is even more challenging. For example, the last column of table 2 shows the computation rate required to perform a real–time $5x5$ convolution on the corresponding data stream. This computation rate is obtained by multiplying the pixel rate by 50, since a $5x5$ convolution operation requires 25 multiplies and 25 adds for each output pixel (see figure 4). The units are $\frac{Mops}{sec}$, or Million generic math operations per second, which may or may not correspond to a machine cycle, depending upon the processing node architecture. The table shows that performing a $5x5$ convolution on the $7.0\frac{MBytes}{sec}$ data stream requires $370\frac{Mops}{sec}$.

> *The computational performance required to execute standard image processing computations, such as a $5x5$ convolution, on standard digitized video is beyond the capabilities of most current single processor computer*

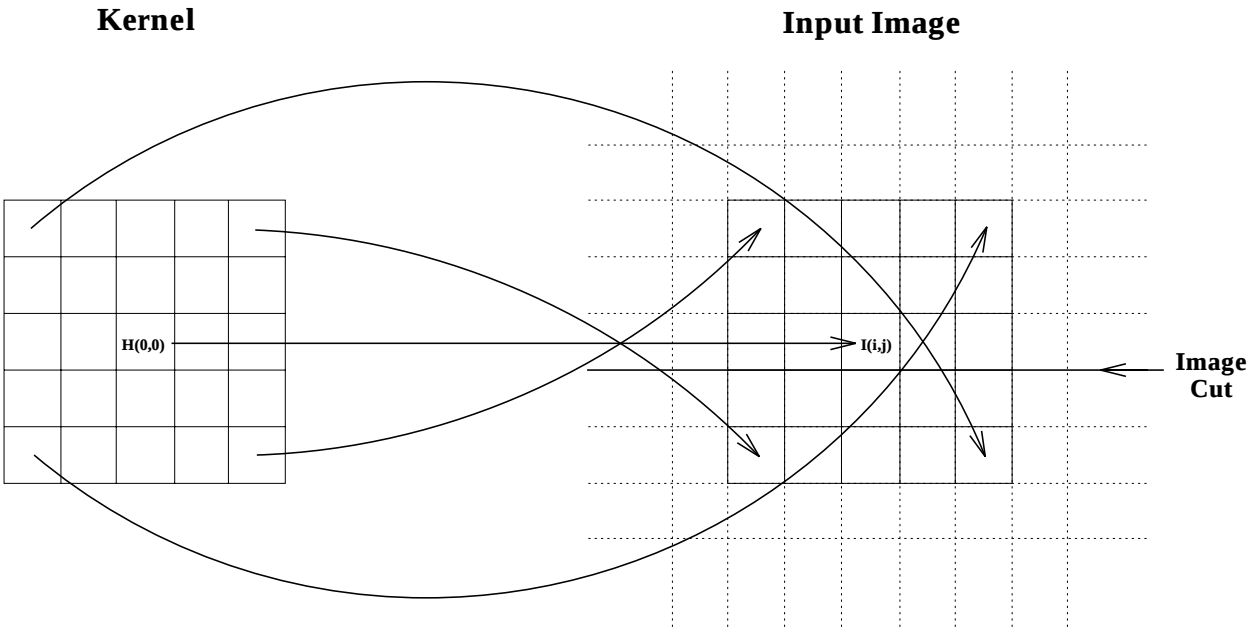

Figure 4: The 5x5 Convolution Algorithm

architectures. Even if a single processor can be found which will support such computation rates, most real applications require that not just a single algorithm, but several such algorithms be performed (eg. gamma correction, then time domain averaging, then convolution, then thresholding, etc). Moreover, many future vision applications may require thousands of mathematical operations per pixel [57] [13], which represents on the order of tens of billions of operations per second for even moderate digitization resolutions. This level of performance is off the scale of current single processor technology.

Latency:

In addition to high communication and computational bandwidths, some applications also require that the computations be performed with very little latency.

The latency experienced in both communication and computation is largely dependent on the processing architecture, and is difficult to control in traditional computational approaches.

Predictable and Controllable Performance Behavior:

It is important to note here that there is a large difference between a *high performance* system and a *real–time* system. A real–time system may not even require fast computations. Of great importance in real–time systems is the ability to predict the performance behavior and guarantee that the constraints will be met. In real–time image processing applications, it is necessary to know a priori to run–time (1) if a computation can be done within the timing constraints on the available hardware, and (2) if so, how to utilize the hardware to realize the real–time system. Predictive models of both throughput and latency are necessary. However, it is generally difficult to accurately characterize the performance behavior of a computation.

Performance models must be developed using knowledge of the algorithms and underlying run–time system. The most reliable methods are based on practical performance benchmarks.

I/O Interfaces:

A real time imaging system must interact with the environment through an efficient interface with sensors [13]. These interfaces must support the high data rates of real–time image processing. There are two relevant types of I/O interfaces.

- Analog video interfaces: input data is acquired by digitizing video with a special video A/D device and output data converted to displayable video by a D/A.

- Digital interfaces: I/O signals may be digital connections to other system components (real–time disks, digital cameras, medical data acquisitions systems, etc.) or other vision systems.

  Synchronizing with and servicing these special I/O interfaces inevitably adds complexity to the implementations of real–time imaging systems. Also, image processing I/O interfaces do not often exist on commercially available high performance computers [54], so special effort must be spent in customizing a machine for real–time imaging.

Approaches Toward a RTIP Solution

The computational requirements of general real–time image processing make it a difficult problem to solve with traditional uni–processor computer architectures. The problem requires a computer system which is both high performance *and* a real–time. There are two approaches which could be taken in building real–time image processing computers to support future vision applications:

1. Implement the computations in specialized real–time hardware.

2. Parallelize the computations and scale the solution to achieve real–time.

The first approach is the one traditionally taken for real–time imaging. This has been due to necessity, since specialized hardware has been the only practical and cost–effective solution. Only recently have VLSI and parallel computing technologies

supplied machines powerful enough to make second option possible. If methods of cost–effectively programming parallel machines can be developed, the parallel solution will be feasible.

Specialized Hardware Solutions

There are many commercially available machines which perform a small set of standard image processing computations at real–time rates. (eg. Matrox, Coreco, DataCube) Some are more general than others. However, all hardware solutions suffer from the same problems:

- They are either not end–user programmable, or offer very limited programmability. The only way to add functionality is through VLSI design, which is a very expensive and slow process. If a specialized hardware solution is used for an application, it is not practical for the user to invent and experiment with non–standard algorithms. As was discussed previously, the ability to rapid–prototype and experiment with algorithms has been proven to be key to the success of image processing applications.

- They offer limited flexibility. The data paths are either hard–wired or have a fixed number of configurations, so the possible ordering of the computations is limited.

- They do not scale. With specialized hardware, it is not always possible to scale the system to meet the performance required for a computation. The amount of computations that can be performed is limited.

- They are difficult and expensive to use. Learning to use specialized hardware systems can require months of training, even for an experienced image processing expert. Training time and cost is a major factor in the economics of computer solutions, since labor is traditionally more expensive than hardware.

- The hardware is expensive. Machines such as the DataCube are very expensive. The minimum initial hardware investment is large, and incrementally building a large system by buying affordable pieces over a time period is usually not an option.

  The use of specialized real–time image processing hardware has proven successful for several applications. However, the inherent limitations have caused the real-time imaging industry to develop the mind-set of trying to fit problems to the fixed capabilities of the available real–time hardware, instead of building integrated solutions to the problems at hand.

This has had the unfortunate effect of isolating the algorithm development community from many real world embedded applications. Algorithm developers have never been able to easily create real–time implementations of new, non–standard algorithms to use in embedded imaging systems, and as a result much of the theoretical image processing developments have gone unused.

Parallel Software

In order to produce a machine which is capable of performing general image processing computations in real–time, a much more powerful approach must be taken.

Software systems are by nature programmable and flexible. However, many real–time image processing applications require far more power than is available from single general–purpose processors.

> *An approach which can produce the necessary performance while retaining the flexibility and programmability of a software system is to develop a parallel computer specifically for image processing [55].*

Difficulties of the Parallel Approach

Developing a general real–time image processing system involves more than building a high performance parallel hardware architecture. Several very high performance concurrent architectures have been developed. However, without high–level programming environments and tools designed for developing image processing applications, these architectures are difficult to program, and thus not cost–effective.

> *The approach must be to build up environments and tools for automatically generating parallel image processing software which can be executed on a number of hardware architectures.*

Problem Statement

The goal of this work is to develop a system for performing image processing computations in real–time. The solution must be as general as possible, in that one flexible architecture should solve many problems.

Image Processing Algorithms

The types of image processing algorithms which are considered are those which

- follow the data access model defined in section II.7.
- have data content independent computation time or an a priori specifiable upper bound on the computation time.

The second requirement is necessary in order to form predictive performance models. Without either data independence or an upper bound on the computation time, this would be impossible.

Synchronous Data Flows

Image processing applications are generally built up by combining several pre–coded algorithms together to form a larger computation. The specification of a computation can be done using a natural and convenient large grain data flow (LGDF) programming technique used in signal processing called Synchronous data flow (SDF). First, a solid definition of a SDF is needed.

A *data flow graph* is a directed graph in which the graph nodes represent processing blocks and the arcs represent communication between the blocks, as shown in figure 5. Such representations are common in signal processing [32], and also in image processing. The Khoros environment, for instance, uses data flow graphs to specify image processing computations [42]. data flow graphs are intuitive and visual, which promotes the integration of high–level graphical programming interfaces.

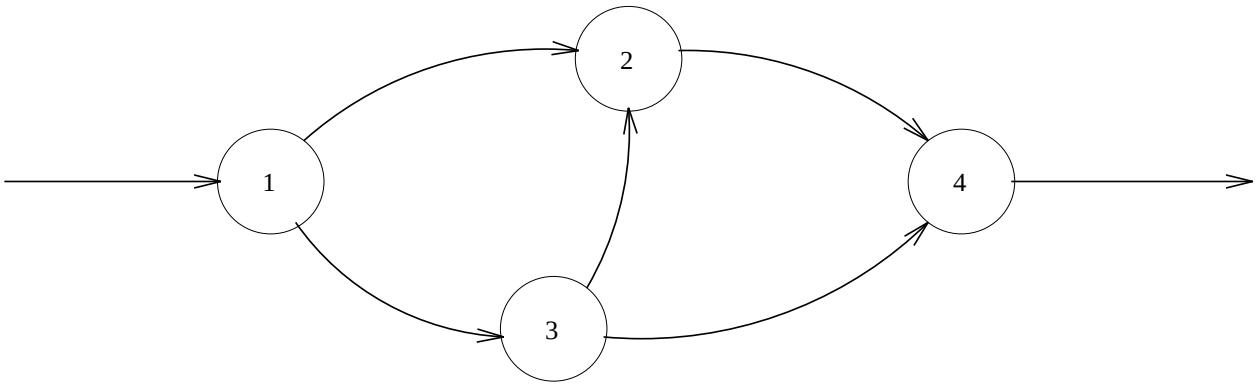

Figure 5: A Computational Data Flow Graph

A computation block is *synchronous* if each time it is invoked, it will consume a fixed number of *data tokens* from each of its inputs and produce a fixed number of data tokens on each of its outputs, and these fixed numbers are known a priori to run–time. The data tokens are so called because the importance is not the size or structure of the data produced or consumed, but the number of times per computation that some data structure will be produced on or consumed from a connection.[4]

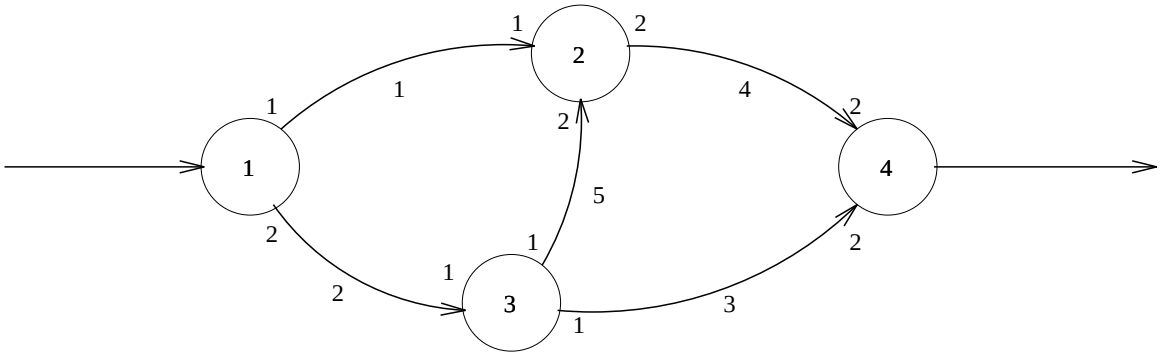

Figure 6: A Synchronous Data Flow Graph

A *synchronous data flow* graph is a network of synchronous computation blocks. An example synchronous data flow graph is shown in figure 6. The number associated with each input and output connection coincides with the number of input tokens

[4]It is assumed that a computation producing a data token on a connection has the same units of *data* as the computations which consume data from that connection.

consumed from or output tokens produced on that connection each time the block runs. For instance, referring to figure 6, when Block3 runs, it consumes 1 token from connection 2 and produces 1 token on each of connections 3 and 5.

In this work, the computations which are considered are synchronous data flows made up of image processing algorithms, or *image processing synchronous data flows*. The work will focus on synchronous data flows that do not terminate, but execute repetitively on an infinite sequence of data. Also, the data flows are restricted to those without cycles (a cycle is a loop in the data flow graph which does not contain a delay element), and without "fan ins" (connections cannot be merged together). "Fan outs" are allowed (a connection can have multiple readers).

Solution Requirements

Real–Time Performance

Since the target applications are real–time embedded imaging systems, the image processing operations must be fast enough to satisfy the environment's timing constraints. The relevant timing constraints are latency and throughput.

As is shown in table 2, real–time imaging requires a hardware architecture which is not only capable of huge computation rates, but which also can support very large system I/O and internal communication bandwidths. Also, the architecture must have facilities for controlling latency in both the communication and computation. I have determined that the most economical way of providing the extremely high computational performance needed by image processing without sacrificing programmability, flexibility, and scalability is to use parallelism [48].

A real–time image processing environment must include facilities to determine a priori to run–time whether the specified throughput and latency goals can be achieved by parallelizing the computations and executing them on the given hardware. If so, then the throughput and latency requirements must drive the decisions about the following:

- the type(s) of parallelism
- the granularity of the parallelism
- the mapping of the decomposed computations to the hardware

If it is determined that the computation cannot meet the performance specifications, decisions must be made about what action to take. The possible actions are

- hard real–time: fail
- soft real–time: make an effort to come as close as possible to the performance goals through compromises
    * down sample the data in time and/or space
    * relax throughput and/or latency requirements

For the environment to make these predictions and decisions transparently, it is necessary that performance models be formed which accurately characterize the behavior of the parallel computations. Difficulties arise because these models depend upon the computation, the processing node architecture, the communication network topology/type, the method/mode of concurrency, and the mapping of the decomposed software to the hardware network. The challenge is to develop concurrent computing

constructs and hardware architectures for image processing for which these critical performance models can be formed.

Cost–Effective Scalability

The system must be scalable to the requirements of many applications. Applications with low computational requirements should be mappable to small hardware configurations containing only a few processors. An increase in the application's computational load should merely require that processors be added to the network.

The system should be both easy and inexpensive to configure. Adding processors to or removing processors from the hardware architecture should require no changes to the software, and the reconfiguration of the system when the hardware is changed should be automatic.

The cost of the system, both in terms of cost per node and cost per performance, should be kept to a minimum. Since the computer hardware industry is constantly producing faster and cheaper processing nodes, it is highly desirable to use hardware architectures constructed by plugging together COTS components.

High–Level Parallel Programming Interface

Parallel systems are much more difficult to program than traditional sequential systems. The lack of suitable programming techniques and environments for parallel systems has historically been the critical technology inhibiting the use of parallelism in real world applications. This is due mainly to the complexity of parallel systems, which can easily overwhelm even an experienced programmer. Task decomposition,

sub–task assignment, sub–task scheduling, inter–task communication, and synchronization are major sources of complexity in parallel systems. A major issue that must be faced in developing a parallel real–time image processing programming interface is how to make these complexities transparent to the programmer by taking advantage of the properties of image processing algorithms.

> *A suitable application programming interface is required which provides the user with transparent access to the parallel facilities.*

Such a programming interface would allow programmers to utilize the parallel hardware architecture easily and efficiently. The complexity of the underlying parallel architecture would be hidden from the programmer, allowing the development of architecture independent applications.

Graphical Programming and Rapid–Prototyping Environment

Image processing is most successful when the programmer is allowed to interact and experiment with computational techniques to find an appropriate solution for the application and the data. An example of an environment which allows this type of interaction is Khoros, a popular software development tool from the University of New Mexico. Khoros provides high–level graphical tools with which the user builds visual data flow *programs*, which are realized and executed automatically. Khoros allows the user to rapidly prototype new techniques and interactively converge on solutions. However, Khoros was developed with flexibly and ease of use in mind, not performance. The resulting implementations, while valuable for algorithm development, do not come close to the performance requirements of most real–time

imaging systems. Khoros is implemented on UNIX workstations, and can be easily be made to run with concurrency at the data flow level. Work has also been done toward automatically generating data parallel implementations of Khoros data flows which run on on Ethernet connected workstations workstation networks [35]. While significant performance accelerations were achieved, the nature of the platform and the overheads of the Khoros system limit the scalability of such solutions. The approach of automatically parallelizing Khoros data flows is insufficient for real–time applications [35].

> *As the success of Khoros has proven, graphical user interfaces are a key feature in today's applications [11]. A programming environment designed for parallel real–time image processing should therefore provide graphical tools with which the developer can program, configure, and control the system.*

Summary of Requirements

The problem to be solved is to build a general real–time image processing system which

- supports synchronous data flows image consisting of image processing algorithms
- is based on a scalable hardware architecture, preferably constructed of COTS parts.
- provides real time performance, with control over both throughput and latency.

- is end–user programmable.
- includes a parallel programming interface which insulates the programmer from the underlying parallel implementation.
- supports the automatic and transparent parallelization of image processing data flows
- provides a graphical rapid–prototyping environment and a graphical user interface with which the user can prototype and interactively experiment with algorithms.

CHAPTER III

APPROACH

The most difficult task to be faced in solving the proposed problem is dealing with the complexities of integrating and programming a parallel system. The approach of directly implementing image processing algorithms using an architecture specific parallel programming language is not feasible. First, image processing algorithm developers cannot be expected to have expertise in parallel programming. Second, writing architecture specific programs eventually leads to being bound to out–dated hardware architectures, since the cost of re–programming a system as new parallel architectures become available is often prohibitive. For parallelism to be a viable option, it is necessary to provide facilities which remove the responsibility of writing parallel programs for the target architecture from the algorithm developer.

Approaches to Parallel Programming

There are various approaches to insulating the programmer from the task of directly implementing parallel algorithms: (1) parallel languages, (2) automatic code translation, and (3) meta–level driven techniques. These are discussed next.

Parallel Languages

There have been several recent efforts to develop architecture independent parallel languages both for general parallel systems and specifically for image processing. The goal is to write programs using a special parallel language, have a compiler

generate the parallel support code, and in some cases, the mapping of the code to the underlying parallel architecture. The complexity in this approach is hidden in the compiler. The parallel languages must support the new parallel architectures as they become available, and theoretically the programs must merely be recompiled to take advantage of the newest technology. Key examples of such parallel languages are given below.

- General parallel languages: A few languages support semi–transparent data parallelism, such as High Performance Fortran (HPF), Parallel C++ (PC++), and OCCAM. HPF and PC++ are extensions of existing sequential languages to which data parallel facilities have been added. OCCAM, however, is a truly concurrent language developed for the INMOS Transputer which is based on the Communicating Sequential Processes (CSP) processing construct.

- Parallel image processing languages: Apply and Adapt were developed at CMU specifically for image processing and support automatic split–and–merge data parallelism on the iWarp architecture.

  These languages provide varying degrees of transparent and automatic data parallelism. However, a major drawback of using a special language is that the language must be ported to the particular hardware architecture being used. Since specialized compilers are usually the last to be implemented for a new architecture, using a parallel language may ensure that a system will always be resigned to employ out of date hardware technology.

Another major inadequacy of these parallel languages for real–time image processing is that each fails to provide the necessary performance predictability and control

over the underlying implementation, so there are no facilities for specifying real–time constraints, and no way of guaranteeing them.

Automatic Code Translation

Automatic translation of sequential programs into concurrently executing implementations involves extracting independent parallel programming units via direct code analysis. The complexity is transferred from the application code to the code analysis and translation engines. This requires the complex and expensive identification of intermodule dependencies to determine where it is safe to split the algorithms. This is especially difficult when there is global or shared data [7].

The automatic translation of sequential code into independent parallel units via direct code analysis is an prohibitively costly approach. Moreover, parallel programs produced by analyzing code are often inefficient, and the performance gains due to the parallelism are difficult to predict [52].

Meta–Level Driven Techniques

Rather than using special languages, or attempting to automatically transform sequential code directly into parallel code, it is perhaps better to develop methods of effectively specifying computations in a general environment and techniques of generating parallel programs from these specifications. The complexity of the parallel implementation is hidden in the programs which translate the specifications into programs. By generating programs from architecture independent high–level specifications, both the application programmer and end–user can be totally insulated from the parallel implementation, or even the existence of parallelism, and the most

recently developed parallel architectures can be exploited [43].

Techniques must be developed to either transform the specifications directly into parallel programs, or use the specifications as an aid in automatically parallelizing existing sequential code. The former approach of direct code generation from high–level specifications, although an attractive solution, is expensive in terms of development time, and does not promote reuse of existing image processing libraries.

As will be seen in a later section, the nature of image processing computations make them particularly well–suited for automatic parallelization. For this reason, it is feasible and efficient to take the approach of first capturing pertinent information about each algorithm in a meta–level information library, and then, with the aid of this information, automatically parallelize the sequential algorithms. Unlike using direct code generation, this approach enables the use of normal sequential programs and compilers, and the use of image processing libraries optimized for the target architecture.

Two important efforts toward using meta–level specifications in this way are described below.

- The CVIP (Computer Vision and Image Processing) environment: This effort at Purdue University is geared toward the specification and implementation of parallel algorithms, and automatically mapping the algorithms to an underlying parallel architecture. CVIP is based on three tools. A program generation tool called *Cloner* allows key information about new algorithms to be declared and stored in a meta–level characteristics set. A tool called *Graph Matcher* is used to map the algorithm onto the target architecture. A third tool called *DISC* (Dynamic Intelligent Scheduling and Control) is used to generate parallel

schedules which implement software data flows made up of the algorithms [24].

- Model–Integrated Program Synthesis: The Multigraph Architecture is a MIPS environment which supports the generation of general complex software systems from high–level graphical models. It has been shown that, in addition to being applicable to various other applications, the MGA can be employed to automatically generate medium–grained parallel implementations of sequential image processing algorithms by taking advantage of data parallel constructs, and that such implementations can be scaled to achieve real–time performance on parallel DSP hardware architectures [36] [37].

The focus of the CVIP effort is in generating valid network schedules for executing vision related tasks in parallel with moderate performance and efficiency. The performance behavior of the algorithm is not part of the algorithm meta–level characteristics set, and the Graph Matcher tool does not take into account latency or throughput goals as it maps the algorithms to the underlying architecture [24]. Also, the system was not designed to include interfaces to live image I/O devices, such as cameras and real–time disks. However, the approach of generating a mapping of algorithms onto a parallel architecture based on high–level specifications has resulted in a semi–architecture independent programming environment.

This type of architecture independent specification is also supported by MIPS, but in a more general way. For instance, the MGA system provides tools for building models in terms of domain specific concepts, and transforming these models into system implementations. The MGA has been shown to be useful for generating real–time instrumentation systems which are flexible, scalable, and can be easily ported to

new processing architectures [4][5]. The same approach can be applied to real–time image processing.

In the case of real–time image processing, the modeling environment should be configured to include the appropriate concepts, such as real–time constraints, algorithm properties, and hardware configuration. An interpreter should be specialized to transform models of real–time image processing applications into mappings of image processing algorithms to parallel hardware architectures which yield real–time performance.

> *Because a major goal in automatically mapping image processing algorithms to parallel hardware is to satisfy real–time performance constraints, it is absolutely essential that the necessary real–time concepts be intrinsic to the meta–level specification environment and the mapping algorithm. For that reason, the approach taken in this work was to use the Multigraph Architecture to develop a meta–level driven software translation system specifically for real–time image processing.*

Before going any further with the description of the approach, a more detailed understanding of MIPS and the MGA is required.

Model Integrated Program Synthesis

MIPS Overview

Model Integrated Program Synthesis (MIPS) is a method of synthesizing software systems from high-level models. MIPS is related to code generation performed by

compilers, but the goal of MIPS is not the generation of machine code. MIPS systems generate instead either code to be executed on a virtual machine, or a configuration of existing computations. Common to all the various existing MIPS approaches is a component called the *model interpreter*, which actually performs the program synthesis. The model interpreter transforms high–level system models, specified in terms of a paradigm, or language, into the system program [1].

The Multigraph Architecture

The Multigraph Architecture (MGA) is a MIPS architecture developed at Vanderbilt University which provides a frame–work and tools for (1) building graphical domain specific models and (2) transforming the graphical models into executable applications [25]. By using domain specific models and interpreters, MGA allows the domain experts to specify a system in familiar terms without dealing with the underlying software engineering details. Referring to figure 7, the MGA consists of the following components:

- A *graphical model builder* (GMB). This is a graphical environment in which domain specific models are built and manipulated. The current MGA model builder is called XVPE.

- A *model database* in which the models are stored. The current implementation uses a public domain *Object–Oriented DataBase* (OODB) called *obst*.

- Domain specific *Model Interpreters*, which translate the system models into the various components of the target system.

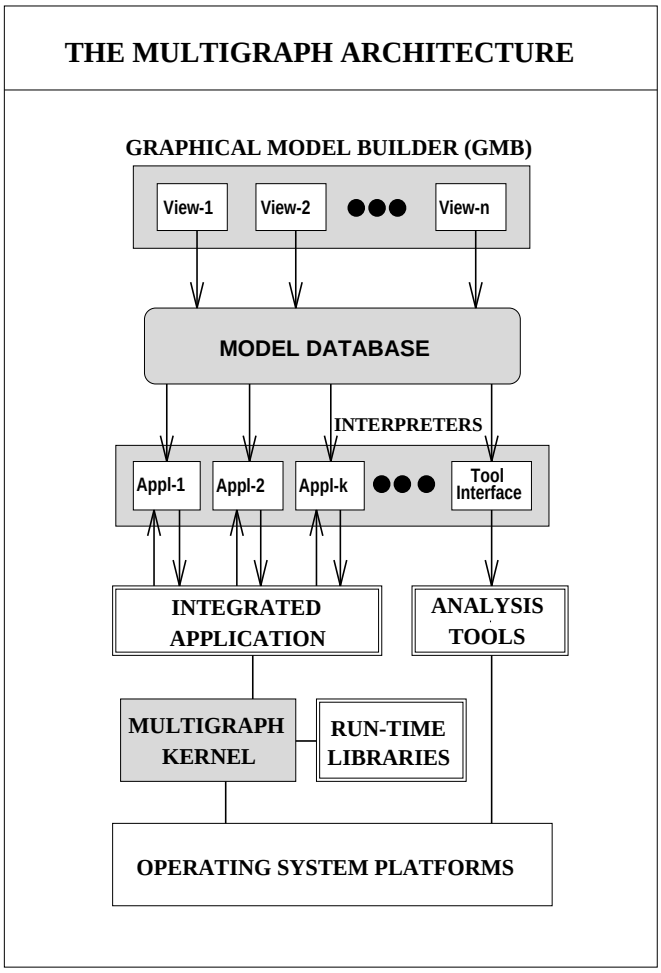

Figure 7: The Multigraph Architecture

– The *Integrated Application* and *Analysis Tools* make up the target software system. In the case of real–time image processing, the integrated application will be the image processing system running in the embedded environment plus an interactive user interface. The analysis tools could be an on–line performance monitoring system which analyzes how well the performance constraints are being met, and possibly reconfigures the system when the constraints or the problem change.

– The *Multigraph Kernel*, *Run–Time Libraries*, and *Operating System Platform* represent the real–time kernel, the image processing libraries, and a set of facilities which provide an interface to the parallel network (tools for loading and debugging the network, such as the *Tick* [2] boot loader for C40 networks). These together represent the run–time environment.

MGA Applications

The MGA has been used to build complex, large scale computing systems. It has been successfully applied to a wide range of domains, such as real–time embedded instrumentation [4], chemical plant monitoring and control [26], simulation [12], fault diagnosis [40], and discrete manufacturing analysis [34]. In particular, the CAD-DMAS (Computer Assisted Dynamic Data Measurement and Analysis System) is a real–time parallel instrumentation system which uses a parallel network of high performance DSPs to solve general signal processing problems [5]. The hardware configuration, software configuration, communication patterns, task–to–processor assignment, and the scheduling are managed through manipulating the system models. A domain specific model interpreter transforms the models into a distributed real–time instrumentation system and a graphical user interface. The complexities of the implementation are hidden in the model interpretation process.

MGA Models

As was discussed, the models are built and manipulated via the XVPE graphical model building environment. For each application, XVPE is configured to work specifically with that application's modeling paradigm. A very brief outline of what

a model is and the types of concepts which can be represented is given below.

The Makeup of A Model

The basic conceptual pieces of a model will be described in this section. In explaining each concept, I give examples from the analog circuitry domain containing circuits made up of voltage and current sources, resistors, capacitors, inductors, transformers, and switches inter–connected by wires.

- Attributes: Simple numerical, textual, or logical descriptions of the model. For instance, some attributes of a *resistor* model would be its resistance, tolerance, power rating, type, and physical size. The attributes of a voltage source would be the voltage/load relationship, type (battery, generator, converter), frequency, and power rating.

- Atomic Parts: Objects which themselves have attributes, but which are not models. Simple circuit elements would be atomic parts.

- Defined Parts: Other models contained by or referred to by this model. If a model contains defined parts, it is a *compound model*. Otherwise, it is a *primitive model*. In a complex electrical system such as an audio amplifier, the circuit sub–systems would be organized into hierarchical models, such as a AC/DC converter circuit, pre–amplifier, filtration, and power amplifier stages. The system circuit model would contain these as sub circuits, and would thus be a compound model. At the lowest level of the hierarchy would be primitive circuit models containing discrete components.

- Link Parts: Any part can be specified as either a *link part*, or not. A link part will be used as a connection point for the model.

- Connections: Atomic parts, defined parts, or link parts of defined parts can be connected together to specify relationships. Connections would be used to represent the electrical inter–connections between circuit components and sub–circuits.

- Conditional Associations: Parts can be *conditionally associated* with other parts, specifying some logical relationship between them. This type of relationship might be used to model the dependence of a voltage source upon another circuit element, or to specify relationships between parts such as switches and relay states.

Model Aspects

If there is a large amount of information in a model, displaying it all in a single view is difficult to comprehend. In order to allow the presentation of information to the user in an understandable fashion, models can be viewed from multiple *aspects*. In each aspect, a subset of the parts can be made visible which relate in a particular way. For instance, a circuit model might have a schematic aspect, in which the parts and inter–connections are specified. It might also have a fabrication aspect for specifying the physical layout of the components and the routing of the runs on each layer of the board.

MGA Models Summary

The modeling concepts available in the MGA system include attributes, parts, hierarchy, connection, association, reference, and multiple aspects. This set of modeling concepts is rich enough to support the needs of the many varying problem domains to which MGA has been applied. The configuration of the editor with a domain specific paradigm is done by first specifying the modeling paradigm in a file using a simple declarative language (Model Description File, or MDF). A tool called *build* is provided which translates the MDF into the XVPE graphical model building environment for the paradigm. Also generated by *build* is a C++ class hierarchy to be used by the model interpreter in extracting information from the model database during model translation. Although the implementation of the actual interpretation algorithm, which is always highly domain specific, is up to the system developer, this task is eased by the automatically generated database access classes.

Approach Taken: Application of MGA to RTIP

The approach followed in this work is to use MGA to build a system in which data flow programs and performance constraints are specified graphically, and both the decomposition of the sequential algorithms and the mapping of the decomposed data flow onto the underlying hardware architecture are performed automatically and transparently with guaranteed real–time performance. Instead of using a parallel language such as HPF, PC++, or Apply to write the image processing algorithms, or trying to directly translate sequential image processing code into parallel code via code analysis, parallel implementations of image processing algorithms are generated

through automatic parallelization of sequential code aided by (1) meta–level information provided about the algorithms in the modeling environment (2) knowledge of the inherent parallelism present in image processing algorithms.

This approach is similar to that taken in the CVIP project at Purdue, with the largest differences being that the CVIP (1) provides tools for specifying and developing parallel programs instead of re–using sequential code, and (2) makes no attempt to model or guarantee the performance behavior of the implementation. The MGA–based approach, on the other hand, is a combination of automatic transformation and meta–level driven program synthesis. The strengths of the approach are: (1) It allows the exploitation of existing image processing libraries optimized for the target architecture. (2) The specification of performance requirements and methods of guaranteeing them can be built into the design, so the real–time issues can be addressed in a natural and elegant way.

CHAPTER IV

TOWARDS A MGA–BASED SOLUTION

This chapter will further develop the approach towards building the specified real–time image processing environment using the Multigraph Architecture for translating graphical models into parallel real–time implementations.

The basic components necessary for an MGA Real–Time Image Processing environment are:

- A library of sequential image processing computations.
- A scalable parallel hardware architecture.
- A Parallel run–time support system to provide the communication, scheduling, and synchronization required to implement decomposed data flows on the underlying parallel architecture.
- A modeling paradigm designed specifically for real–time image processing. The paradigm must contain all the necessary concepts such that models of the software, the hardware, and the real–time constraints are adequate such that an algorithm can be devised to automatically map the computations to the parallel hardware.
- An interpreter to decompose and map the image processing computations to the parallel hardware architecture. The interpreter requires the following:

* Discovery and generalization of the inherent parallelism of image processing algorithms.

* Development of methods of mapping image processing data flows to parallel resources. Mapping involves decomposing (parallelizing) the computations, building performance models of the parallelized computations, scaling the parallelism, and allocating the scaled solution to the processing nodes, and must be done in such a way that the timing constraints are met.

* Development of predictive performance models describing the throughput and latency resulting from the given mapping of the computations to the resources.

The following presents discussions of the major issues which must be addressed in building and integrating these components. The flow of the discussion will first center upon how image processing algorithms can be most easily parallelized, and then will concentrate on selection of an appropriate hardware architecture, and then a run–time system. After that, the remainder of the chapter will focus on developing the MGA components of the system which make the approach novel: (1) the modeling paradigm, which is used for meta–level specification of real–time image processing applications, and (2) the model interpreter, which is used to automatically transform the models into a decomposition and mapping of computations to resources which results in the specified real–time performance.

Discovering the Inherent Parallelism In Image Processing

Image processing has been the most common area for the application of high performance parallel computing [54]. This is because the operations have several characteristics which make them particularly suitable for implementation on parallel computers [55].

- They are regular. In most cases the same operation is performed repeatedly across the image.
- Usually the algorithms perform fairly simple operations to produce each output pixel.
- Most image processing programs traverse the image in raster order. This commonality in procedure makes it easy to parallelize many algorithms with the same technique.
- Images are large data sets which lend themselves easily to data parallelism.
- Processing sequences of images with these algorithms requires high computational power, and many applications (especially the real–time applications) need the acceleration of parallelization.

These basic parallel computing constructs can be defined which best support these characteristics:

- Data–parallel constructs
    * Spatial decomposition
    * Temporal decomposition

– Functional parallelism

The next sections will discuss these parallel processing constructs and how they can be exploited in image processing.

Spatial Decomposition

Because of the properties mentioned above, many image processing algorithms are easily data parallelizable by decomposition of the data in the image plane. A simple data parallel programming technique which is applicable to image processing is the *split–and–merge model*. In this technique, each input data structure is *split* into N

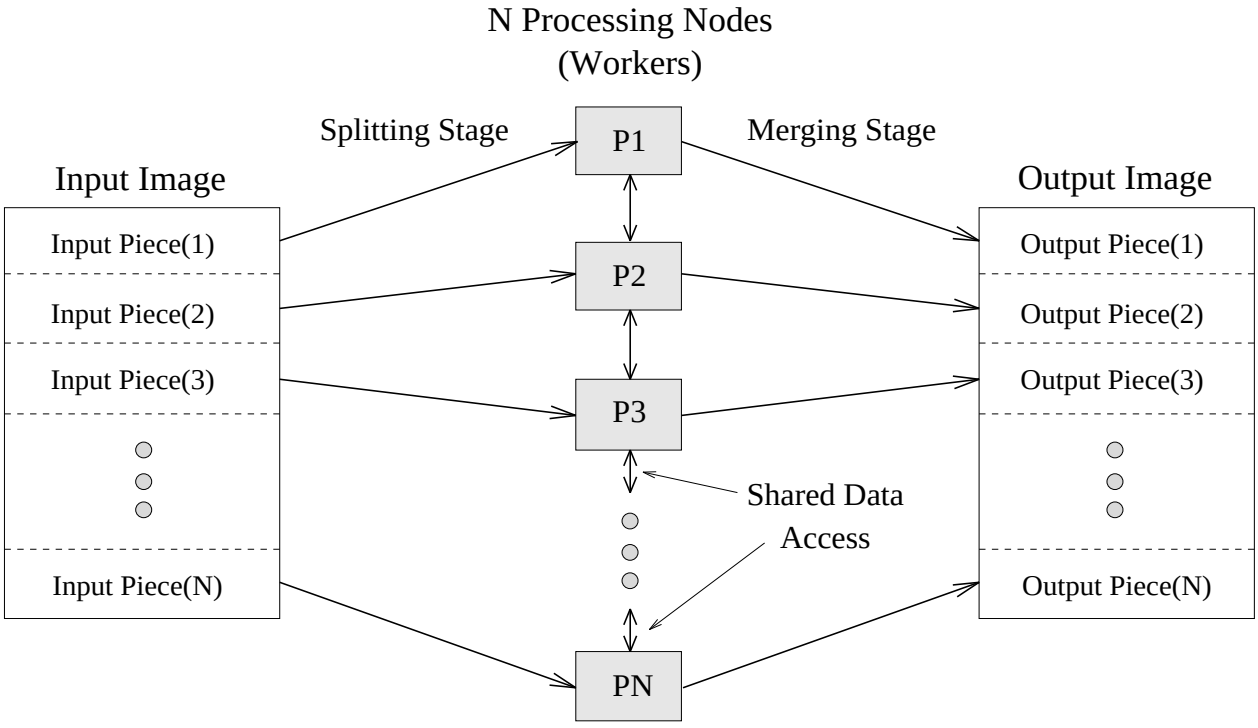

Figure 8: Spatial Decomposition (Split–and–Merge Processing)

pieces, which can be blocks of rows, blocks of columns, panels, overlapping regions, etc. The pieces are distributed across the memories of the N *worker* processors, each which performs the same algorithm on its sub–section of the data. The partial results

are then *merged* to form the output. In figure 8, a split–and–merge computation is shown in which the data has been split into blocks of rows, and the final result is formed by concatenating the partial results.

Applicable Algorithms:

It is important to be able to determine whether or not an operation can be performed with this model. In [55], Webb proves that any operation that can be computed in forward or reverse order over a data structure can be performed with the split–and–merge technique. I will refer to algorithms which meet this criteria as *splittable* computations. Since most image processing computation are splittable, it may be easier to give examples of types of algorithms which are not. Examples of non–splittable computations are error–diffusion half toning and some region–growing segmentation algorithms which require a particular image scanning order [55]. Since these order dependent algorithms are not prevalent in image processing, splittable image processing computations form a very general class of algorithms.

Performance

Even more relevant than knowing which algorithms are splittable is determining the performance as the parallelism is scaled, which will depend upon the communication and computation overheads. Many times a splittable algorithm could be spatially decomposed and mapped to the available architecture, but the resulting overheads would limit the performance gains. A method of quantifying the performance of a spatially parallel computation for a given mapping to resources is necessary.

The acceleration achieved by a split–and–merge computation depends upon (1) the

communication overhead, and (2) the computation overhead. These are explained below.

Communication Overhead

Sources of communication overhead in the split–and–merge processing model are:

- splitting: distributing the image pieces to the processors
- merging: gathering the partial results to be combined
- sharing: communicating *shared* data between worker processors

The splitting and merging communication overheads are obvious. However, the *sharing* overhead requires some explaination.

Sharing Input Data:

Some algorithms require that part or all of the data be available to more than one of the worker processors. Consider, for example, the 5x5 convolution operation, which is implemented by the equation

$$O(i,j) = \sum_{k=-2}^{2} \sum_{l=-2}^{2} I(i-k, j-l) * H(k,l)$$

where H is the $5x5$ convolution kernel.

Referring to the equation and figure 4, suppose that $I(p,q)$ is a pixel lying just above a horizontal cut in the input image. In order to correctly calculate $O(p,q)$, the processor must access data from five different rows in the input image, two of which are below the horizontal cut. These rows were distributed to the local memory of a different processor. They must somehow be available to two different processors, or

shared. If the architecture is a distributed memory multicomputer (as will be determined in a later section), this sharing of data must be provided via inter–processor communication, which results in communication overhead.

The determination of which input data must be communicated to which processors requires specific knowledge of the algorithm's data access patterns. Referring back to the general image processing algorithm model defined in the background chapter, note that knowledge of the *total data dependency set* required by the algorithm in computing each output pixel is sufficient information to determine, for a particular data decomposition, what regions of the input data must be available locally to each processor. The shared pixels lie in the overlap between these regions.

Computation Overhead

Sources of computation overhead in the split–and–merge processing models are:

- merging: combining the partial results to form the output data structure
- non–linear dependence of execution time on data size

Merging Partial Results:

How the partial results must be combined in the merge step varies with the computation. Merging may be merely concatenation of partial results, or may involve other operations such as addition. When the computation is such that each worker processor produces a section of the output image, the partial results are just concatenated together to form the output. This is the case shown in figure 8. However, in some cases the partial result computed from each local sub–image does not correspond to

a spatial decomposition of the output data structure. One case in particular is the histogram calculation. The number stored at each location of a histogram is a count of the number of pixels in the entire image with value corresponding to the location index. For instance, the first position in the histogram, *location* 0, contains the number of pixels in the image for which $value = 0$. If the split–and–merge technique is applied to the histogram calculation with the input being distributed as blocks of rows, the partial results will be histograms of the sub–images. In order to combine these histograms to form the "global" histogram, the "local histograms" must be point–wise added. These extra additions introduce an additional computational overhead which must be factored into the performance models.

Relationship Between Execution Time and Data Size:

A reduction in the execution time for any one image is achieved by decreasing the size of each local data set, and presumably also the number of computations to be done by each worker processor. The tendency is that an algorithm's computation time decreases as the number of input pixels decreases, so generally as N scales, the execution time is reduced. Each of the N processing nodes computes at least $\frac{1}{N}th$ of the result, so the execution time is reduced by a factor of at most N. This translates to a throughput gain factor of at most N (linear speedup is the best case). However, the execution times of all algorithms do not depend upon image size linearly. This deviation from linear speedup can be viewed as computation overhead.

Performance Models

> *The quantities which must be characterized in order to construct performance models for split–and–merge computations are (1) the communication overheads due to splitting and distributing the input data (including shared data) and gathering the partial results, (2) the time required to combine the partial results, and (3) the time required to execute the algorithm on the reduced size data set.*

These characterizations depend upon the following properties of the algorithm: (1) the data dependency characteristics of the algorithm (2) how the execution time varies with image size (3) the method used to combine partial results (4) how the data structure(s) have been split.

This information alone, however, is not sufficient to build complete performance models, since the communication overheads will also depend upon the properties of the run–time system and the communication network. As will be seen later in the development, these characterizations must be tailored to meet the properties of the underlying implementation.

Temporal Decomposition

Algorithms which are not *splittable*, or for which the resulting gains of spatial decomposition would minimal, can still be successfully data parallelized when operating on image sequences by taking advantage of their structure. Image sequences can be decomposed temporally instead of spatially (split the data along the *time domain*, instead of the *spatial domain*). Instead of distributing pieces of an image to worker processors, as shown in figure 8, we can distribute the pieces of the image

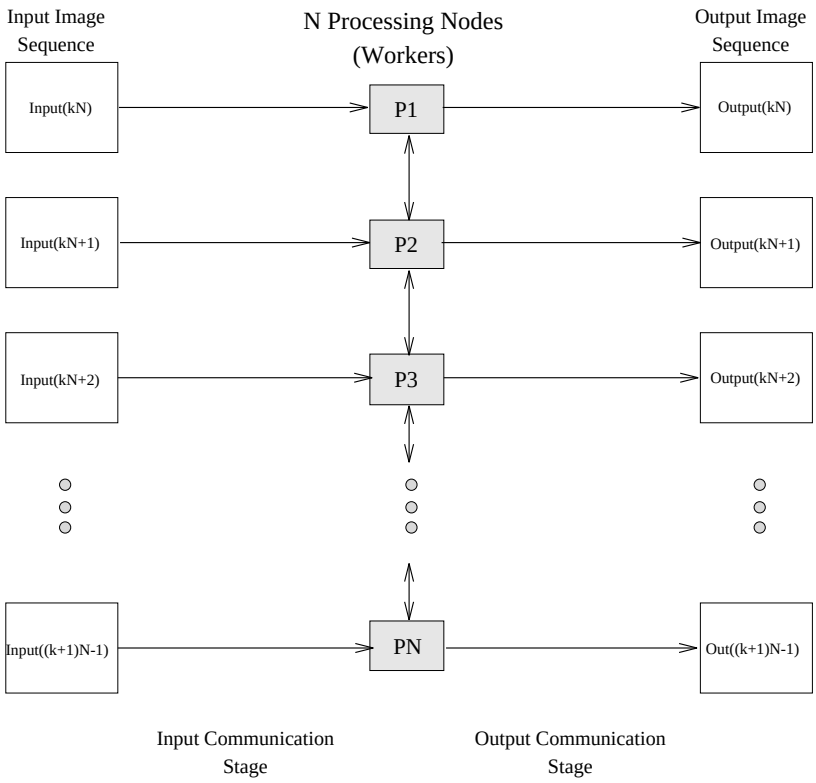

Figure 9: Temporal Decomposition (Sequence Splitting)

sequence (entire images) as shown in figure 9. $Input(i)$ is distributed to processor $(i \; mod \; N) + 1$, where N is the number of processors, and $i \in \{0, 1, \ldots\}$.

Applicable Algorithms:

Since each of the workers processes an entire image, any 2–D image processing algorithm, either splittable or non–splittable, can be parallelized with this method.

Performance

Each worker processes an entire image, so there is no decrease in the time it takes for any one image to be processed (latency). However, images are being processed concurrently, so there is an increase in the rate that images are being processed (throughput).

Methods of quantifying the performance characteristics of temporally decomposed computations for a given mapping to resources are necessary. The performance depends upon the communication overheads, which are:

- distributing the input images to the worker processors
- sharing images between the worker processors
- communicating the results to form the results sequence

Image Sharing Overhead:

Applying temporal decomposition to image sequence algorithms, such as time averaging and 3–D morphology, requires additional communication because these algorithms share data in the time dimension. Two or more consecutive frames are needed to compute the local output, but temporal decomposition processing puts consecutive frames on different processors. Instead of rows, or columns being shared, each entire image is shared by two or more processors. The resulting communication overheads could preclude performance gains. Performance models for temporal decomposition require characterizations of image sharing overheads, which depend upon the image size and the properties of the underlying run–time system.

> *The sources of overhead in temporally decomposed computations which must be modeled are (1) the communication overheads necessary to distribute the images (including the possibility of entire images being shared between processors), and (2) gathering the results.*

Functional Parallelism

Functional parallelism takes advantage of the natural concurrency of an algorithm. The algorithm is broken down into semi–independent sub–computations, which interact by passing data.

$$Sequential\ Computation \longrightarrow \{SC_1, SC_2, \ldots, SC_N\}$$

The sub–computations along with the data passed between them form an implicit data flow computation graph. The sub–computations can be executed concurrently on N processing elements, with data being transferred between the computations via inter–processor communication. Figure 10 shows a computation which has been decomposed into four sub–computations, represented by boxes in the figure. The lines between the boxes represent the exchange of intermediate data. The figure also shows a hardware architecture which mimics the structure of the algorithm, with processing element PE_k executing sub–computation SC_k.

Applicable Algorithms:

Functional decomposition is a very general parallel processing construct. Any algorithm which contains inherent concurrency is applicable. The usefulness of this paradigm, however, is totally dependent on the structure of the computation. The prospects of taking a sequentially implemented algorithm and automatically extracting the functional structure and decomposing it into sub–computations is a complex and costly activity [52], and is not a goal of this effort. Here the assumption is made that any functional parallelism is specified explicitly using a data flow graph representation.

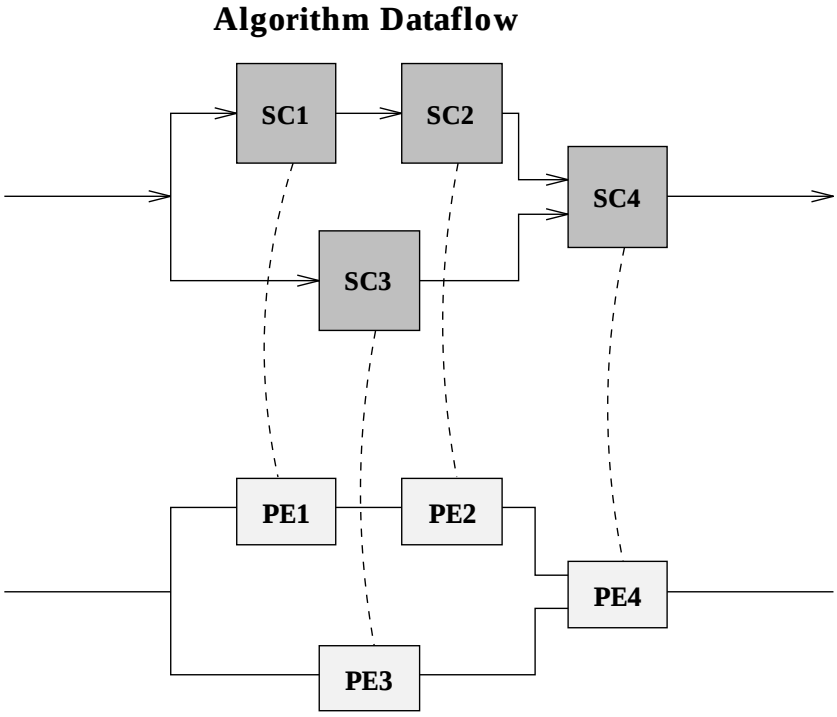

Figure 10: Functional Parallelism

Fortunately, the data flow graph is a comfortable and commonly used method of visualizing a computation. There are commonly used graphics–based tools in which computations are specified as visual data flow *programs* which are implemented using pre–coded software modules [42]. This effort will take a similar approach of specifying computations by building graphical data flows made up of image processing algorithms. With the sub–computation specified explicitly as a graphical data flow, the application of functional decomposition to parallelize the computation is straight–forward.

Performance

The performance gains of functional parallelism are achieved through allowing the sub–computations to execute concurrently, and are bound by the structure of

the data flow. The efficiency is controlled by the relative complexities of the sub–computations. Unless the sub–computations are of similar complexity, the system will not be *load balanced*. Non load balanced systems have performance bottlenecks that result in the inefficient use of the parallel resources and poor performance gains.

> *The major difficulties of using functional parallelism are (1) the success is totally dependent upon the structure of the computation, and (2) the maximum performance gain is controlled by the number of sub–computations, and (3) unless the system is load balanced, any performance gains may be lost to inefficiency.*

Hybrid Parallel Constructs

Since it is natural, and quite common, to specify sequential computations by building data flows of low–level algorithms, it is reasonable to extend this approach and form a hybrid parallel construct which uses two levels of parallel decomposition, with the top data flow level being functionally parallel, and the underlying sub–computations being data parallel. Scaling and load balanced to the target throughput can be achieved by scaling the data parallelism of the sub–computations independently until each achieves the target throughput. This type of hybrid parallel construct can be applied automatically to data flow computations if methods of automatically data parallelizing and scaling the sub–computations are developed.

Summary of Decomposition Techniques

The types of parallelism which are inherently applicable to parallelizing image processing data flows are:

- Spatial decomposition (split–and–merge)
- Temporal decomposition (sequence splitting)
- Functional decomposition (data flow decomposition)
- Hybrid decomposition (top level functionally parallel, lower level data parallel)

Selection of a Parallel Hardware Architecture

The available parallel computer architectures vary widely. The following argument will define a class of architectures which is well suited for the real–time image processing problem domain.

MIMD Versus SIMD

One way of differentiating between parallel architectures is based on how the parallel execution is controlled. The two major types of architectures applicable to image processing are SIMD (Single Instruction, Multiple Data) and MIMD (Multiple Instruction, Multiple Data) architectures [18]. SIMD architectures have traditionally been used for low–level image processing, and MIMD for mid and high–level vision operations.

In a SIMD architecture, all of the processors receive instructions from a single control unit, and the data is distributed across the processing elements (see figure 11).

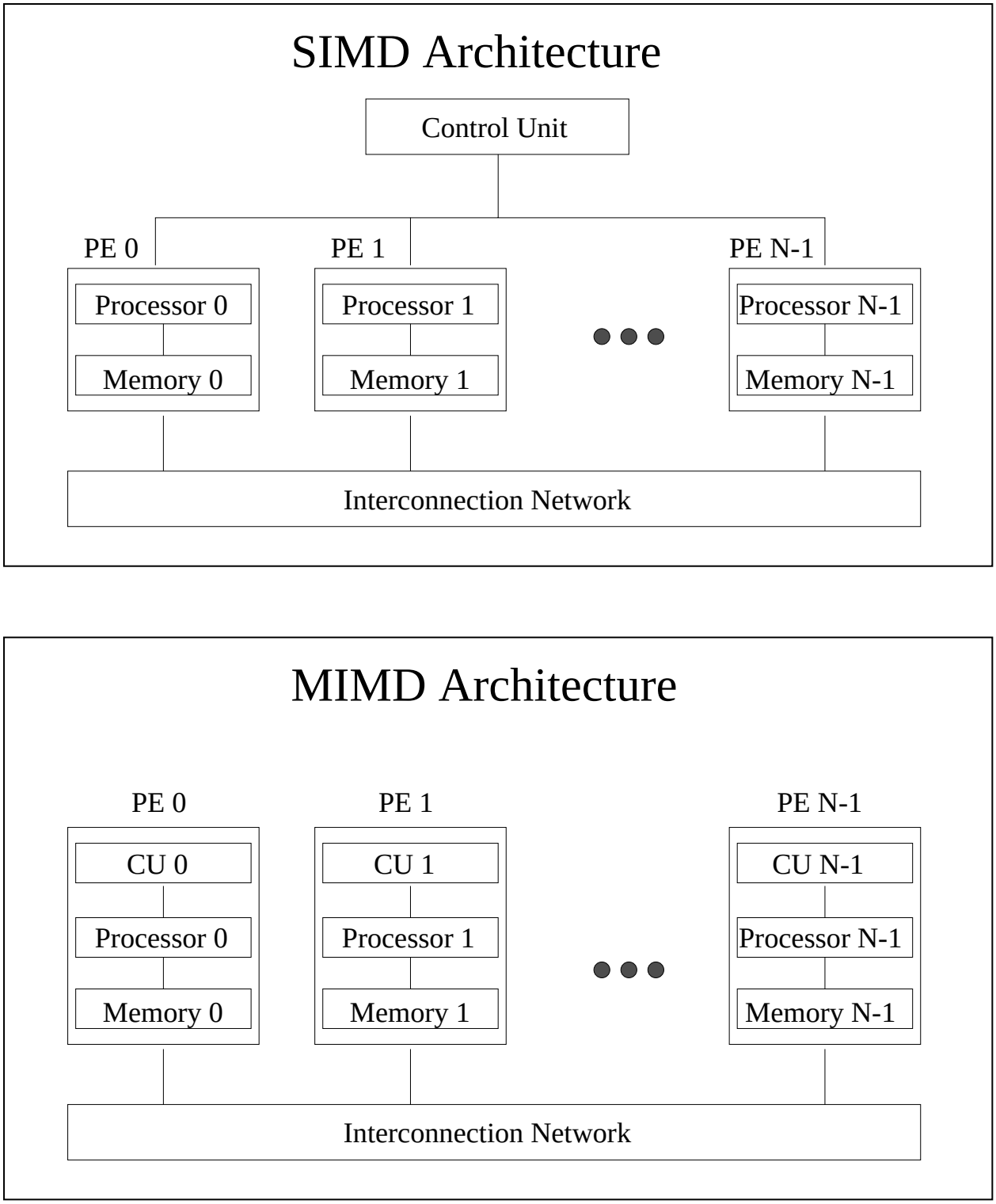

Figure 11: SIMD and MIMD Architectures

Each processor executes the instructions synchronously in a locked–step fashion. Examples of SIMD architectures are CLIP–4, MasPar, and the MP–1 [48]. An MIMD architecture has totally independent processors, each with a control unit. The processors can execute different programs and can operate asynchronously. Examples of MIMD machines are the BBN Butterfly, Cedar, and the Intel N–Cube [48].

At first look, it might seem apparent that SIMD architectures are optimal for the problem domain I have defined, since usually the same simple computation is performed across an image. Fine grained SIMD image processing architectures have been built which take advantage of this property. However, a closer study reveals that SIMD architectures are not well suited for the general problem domain. Not all image processing operations considered map well to this programming model. For instance, histogram operations do not work well on SIMD architectures because they are not neighborhood operations. By choosing a SIMD model, the ability to efficiently perform such computations is lost. Moreover, SIMD architectures do not offer the scalability and flexibility the solution requires. Due to the fact that SIMD machines are usually custom built, it is not always possible to add processors or partition the architecture to match a computation. The desire to use COTS parts also reduces the attraction of such machines, since most SIMD architectures do not.

A more practical, and no less intuitive approach is possible by using coarser grained MIMD architectures with fewer more powerful processors. Using the split–and–merge programming model, which is comparable to a software version of SIMD computation, MIMD architectures can easily be exploited for low–level image processing [55]. Since MIMD architectures can perform any type of computation, the same machine can be used for all operations. By using a MIMD architecture with a moderate number of

more general purpose processors, we gain flexibility and programmability, but lose the simple SIMD programming model. The COTS issue is well served by MIMD machines, since there are several commercially available computer architectures designed to be plugged–together to form parallel MIMD networks.

Shared Versus Distributed Memory

Another way in which parallel architectures differ is in the memory model used. Both *shared memory* and *distributed memory (message passing)* architectures have been used for image processing. Although shared memory systems have the advantage of being easy to program, using a shared memory model has several drawbacks. One is that systems which physically share memory are *fixed* architectures, and therefore are not modular, extensible, or scalable. *Virtual shared memory* architectures, such as the KSR (from Kendall Squared Research), actually have physically distributed memory, but emulate a shared memory space between distributed memory banks by passing messages. A drawback of virtual shared memory architectures is that the scalability is highly problem dependent, and moreover implementation dependent. By the same causes, it is also very difficult in general to form dependable computational performance models for such systems. These models are crucial to the development of a parallel real–time image processing system. Distributed memory/message passing architectures, although they are in general more difficult to program, can be designed to be much more scalable and flexible than shared memory machines. Therefore, I have determined that a distributed memory hardware architecture would better support the needs of general real–time image processing.

Distributed Memory Multi–Computers

The class of hardware platforms to be considered has been limited to Distributed memory MIMD architectures, also know as Multi–Computer architectures [44].

Several such multi–computer architectures have been developed which approach the needs of real–time imaging. Examples are Carnegie Mellon's iWarp architecture [9] and the Kyushu reconfigurable parallel processor [33]. In addition to these, it has also been shown that platforms powerful enough for real–time image processing can be build from processors designed to be parallel processing building blocks, such as parallel DSPs (Digital Signal Processors) [36] [37].

The constraints that the hardware architecture should be flexible and cost–effective to build, configure, and manage, lead to the preference of machines built from COTS parts over those with specially designed processors. Also, since the industry is constantly updating the processor technology, using COTS parts allows the most up–to–date processors can be integrated quickly. Since parallel DSP modules deliver high performance per cost and are widely available COTS, building a multi–computer architecture based on DSPs is a very powerful solution.

Of the many parallel architectures which could have been used for this work, the type of hardware platform chosen is a distributed memory, message passing, multi–computer. Specifically, I chose parallel DSPs because (1) such architectures are appropriate for the problem domain, (2) they can be built from COTS parts, and (3) I was fortunate to have a moderate number of C40s available with which to build a prototype.

Note that the MIPS approach taken could be applied to other hardware platforms, even mixed mode or reconfigurable architectures, with little change in the top–level components.

Parallel Run–Time Support

In this section, the parallel run–time issues involved in implementing parallel image processing data flows on multi–computer networks will be discussed. Decisions will be made as to the type of communication, synchronization, and scheduling which are most appropriate for the problem domain.

When using a distributed memory multi–computer, scheduling, communication, and synchronization must be handled explicitly in the software. A parallel run–time kernel which provides these facilities is necessary. There are several ways to handle each of the tasks, and the solutions must be chosen based on the needs of the application. The following discussion will narrow the scope to run–time support techniques most appropriate for implementing real–time image processing data flows on multi–computers.

Synchronous Versus Asynchronous Communication

The communication system can be *synchronous*, or *asynchronous*. Synchronous communication imply that both the sender and receiver must be ready for a transfer before it can occur. A sender will block until the receiver is ready and the transfer has been performed. Thus the tasks are automatically synchronized whenever they communicate. With asynchronous communication, there is a buffering process between the sender and receiver which queues the communication buffers and allows the tasks

to operate more independently. The buffering process adds overhead in asynchronous communication.

Static Scheduling Versus Dynamic Scheduling

Task scheduling can be done either statically, or dynamically. Static schedulers perform a pre–determined schedule, or sequence of tasks, and introduce very little or no computation overhead. The schedule to be performed on each node must somehow be determined prior to run–time from knowledge of the computation structure. Algorithms and heuristics exist for generating static schedules for synchronous data flow graphs, such as the Lee–Messerschmidt algorithm [32].

With dynamic scheduling, the task to be run on a node is chosen based on the state of the system. Possible methods are (1) priority driven scheduling, and (2) data driven scheduling. In priority driven scheduling, each task is assigned a numerical priority which places it in a scheduling queue. The priority may or may not change each time the scheduler runs. For example, the dynamic priority scheduler used in the UNIX kernel increases the priority of processes which have not run recently. In data driven scheduling, a task is scheduled when its input data is available. There are many other dynamic scheduling techniques which are not mentioned here. In each, however, overheads are introduced by the run–time mechanism which determines dynamically which task will run next.

Synchronous Communication and Static Scheduling

Real–time image processing applications such as video processing require huge computation and communication rates, and the current generation of parallel computers is just above the necessary performance threshold. Thus, the scheduling and communication overheads in the run–time kernel must be minimized. In addition to high performance, the real–time nature of such systems requires that the computation and communication performance be guaranteed, so the scheduling and communication techniques used must have predictable behavior.

> *To minimize overheads and achieve predictability in the run–time kernel, it is necessary for the kernel to use static scheduling techniques and synchronous communication.*

Development of the RTIP System Synthesis Approach

In order to use the system synthesis techniques provided by MGA to automatically decompose synchronous image processing data flows and allocate them to a network of C40s, the following are required:

- A real–time image processing modeling paradigm for specifying the software data flow, the hardware network, and the performance goals
- A model interpreter to transform the models into a mapping of the data flow to the resources which meets the timing constraints

The overall design for the interpreter component will be developed in this section. During the discussion of the decomposition and mapping schemes, information and concepts which must be present in the models will be mentioned. This approach

of discussing the interpreter before the modeling paradigm was taken so that the motivation behind the elements of the models will be more clear when the final modeling paradigm design is presented in the next chapter.

Interpretation: Mapping Computations to Resources

Somehow, the computations (a synchronous image processing data flow), must be mapped to the underlying resources (a parallel run–time system running on a C40 network). Note that although the hardware architecture chosen for the prototype implementation is a C40 DSP network, the mapping technique should be kept as general as possible to ease the migration of the resulting system to new multi–computer architectures.

Overall Approach

The goal of the interpretation processes is to automatically decompose the image processing synchronous data flow, build performance models of the implementation, scale the parallelism, and allocate the parallelized, scaled computations to the underlying run–time system and processing nodes.

The image processing data flows will be decomposed using the two–level hybrid decomposition technique introduced in a previous section. This approach is to use functional decomposition at the top data flow level, and spatial or temporal decomposition for each sub–computation. Referring to figure 12, note that the data flow has first been partitioned into sub–computation blocks before the second level decomposition and scaling. The reason for this extra step has to do the implementation of the prototype run–time system and mapping algorithm, and will be explained later.

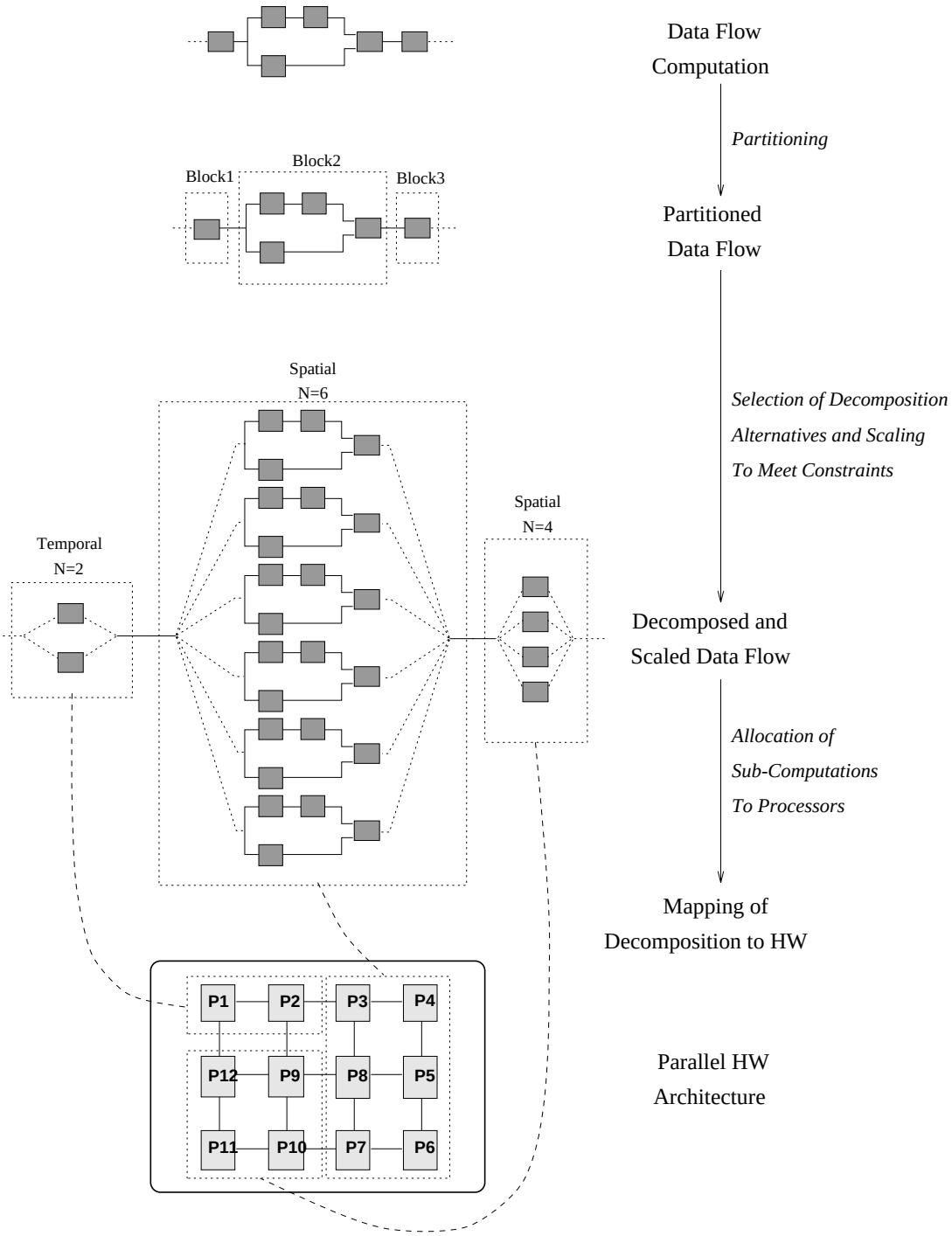

Figure 12: Overall Mapping Approach

The second level decomposition is done by searching for decomposition methods and scaling factors for each of the blocks which will result in the performance constraints being met. In evaluating each combination of block decomposition methods, predictive performance models are used. Building the performance models requires a priori information about (1) how each of the algorithms accesses data in building the partial results (data dependency), (2) the execution time/image size relationships for each algorithm, and (3) how the parallelized sub–computations will be mapped to the underlying run–time system and hardware architecture.

Difficulties

Note that the construction of performance models requires knowledge of the mapping between computations and hardware, but that the mapping is done after the parallelization decisions have been made (see figure 12). A major obstruction to automating the decomposition, scaling, and mapping processes is that these tasks are inherently inter–dependent. Determining acceptable data flow decomposition techniques and scaling the parallelism require performance models, which require the mapping and knowledge of the run–time system.

Because of these inter–dependencies, the general problem of mapping image processing synchronous data flows to arbitrary multi–computer networks while simultaneously guaranteeing that throughput and latency constraints are met has no closed–form solution. Moreover, performing an exhaustive search of all decompositions and allocations until the constraints are met is not a practical alternative because the search space is too large.

The approach toward automatically mapping the computations to the resources must be to reduce the size of the search space by developing simplified decomposition procedures and allocation techniques which exploit the capabilities of the target hardware architecture and favor the properties of the majority of the applications.

The rest of this section discusses the overall assumptions made in this work about the image processing computations to be performed, the decomposition techniques with which they will be parallelized, and the types of hardware architectures they will be mapped to. However, the behavior of the run–time system, the sub–task to processor allocation technique, and the specifics of the performance models are purposely not discussed here. These parts of the system are considered to be less general and more dependent upon the chosen implementation. In the following chapter, a run–time system, an allocation technique, and the related performance models are discussed for an implemented example system.

Assumptions

The following assumptions are made about the image processing computations and the hardware architecture in order to reduce the complexity of the solution.

Well Behaved Algorithms:

It is assumed that each of the image processing algorithms in the data flow is *well behaved*. In addition being an *image processing algorithm* as defined in section II.7, a well behaved image processing algorithm meets the following constraints:

– The execution time of the algorithm either

* is independent of the data content and time invariant

* has an a priori specifiable upper bound

- The algorithm has been benchmarked on the target processing node type (using a single processor) for an adequate range of data sizes such that linear interpolation on these benchmarks will produce an accurate estimate of the execution time for a given data size. (The same can be done with upper bounds if the algorithm is data or time dependent.)

- The *total data dependency set* for each output data element (refer to figure 2 and equation 6) is bounded and known prior to interpretation time. In other words, the manner in which the inputs and past values of the output are accessed in constructing the output data structure is known and specified (in the models).

System With Single Input/Output:

This work has been driven by a particular need for a stand–alone real–time video processing system to be used in both on–line and off–line analysis of a running turbine engine compressor stage. The target system processes video obtained from a single sensor, and displays the results on a monitor or writes them to a real–time disk (see figure 3). For this reason, the scope of this work has been limited to systems in which there is exactly one source of image data and exactly one sink of image data. The image processing synchronous data flow graphs, then, have a single source block (a computation with no inputs) and single sink block (a computation with no outputs). The structure within the system, however, is not constrained to a single computation path. Algorithms can have multiple inputs and outputs, and connections can "fan

out", meaning that output data can be distributed to multiple consumers. However, connecting to an input from multiple outputs ("Fanning in") is not supported. Data can be looped back through delay elements. Figure 13 shows an example data flow with a single input and output, but with complex internal structure.

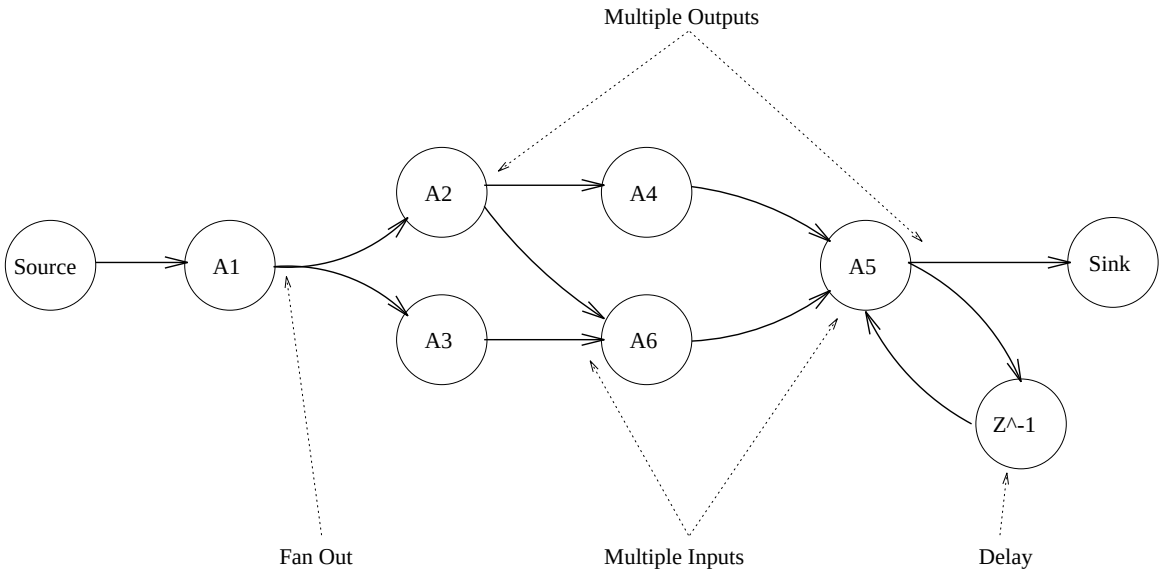

Figure 13: A Complex Data Flow

Although the single image stream limitation fits applications in which a single visual sensor is used to produce a single stream of results, many vision applications require that data from multiple sensors be used to produce multiple results streams. For instance, in robotics, it is common to use stereo vision (two or more sensors with different viewing angles) in constructing a 3-dimensional scene model. Also, military vision systems often use multiple sensors of different types (e.g. radar, visible light, and infrared) and combine the data to gain additional information. In these cases, several independent or inter–dependent analysis may be done on the data streams. In order to support data flows with multiple data sources and data sinks, the approach discussed in this work will have to be extended. One method would be to decompose

the system data flow into communicating sub–systems, each having a single data source and sink. Dealing with this issue should be part of the future direction of the project.

Homogeneous Pipeline–Connected Hardware Architecture:

It is assumed that the underlying hardware architecture is a homogeneous network of C40s connected in a topology containing at least one pipeline–connected sub–graph beginning at the source node (a C40 image digitizer or a PC) and ending at the sink node (a C40 image display or a PC). This sub–graph is a *path* from source to sink. Homogeneity in this case means that each of the nodes in the path has the same performance characteristics and the same basic set of resources. Exceptions are the digitizer and display modules, which have extra capabilities and a different memory configuration. The best case is that the longest path includes each of the C40s (a *Hamiltonian path*[1] between the source and the sink exists) [8]. Note that a Hamiltonian path is not a requirement, but since only the nodes lying on the longest path from source node to sink node will be used by the system, the non–existence of a Hamiltonian path implies that some processors will effectively be wasted.

Why is a Hardware Pipeline a Good Choice?:

It may seem that a pipeline is not an adequately flexible hardware inter–connection topology for executing the types of image processing data flows that have been specified. However, most real–time image processing applications process sequences of images, and in many cases, the latency is not a hard real–time constraint. Thus,

[1] A *Hamiltonian path* is a path in a graph which visits every node in that graph exactly once. A graph containing a Hamiltonian path beginning and ending at the same node is called a *Hamiltonian graph*.

temporal decomposition (sequence splitting) can be used in a large number of applications. Sequence splitting is very easily implemented and mapped to pipeline–connected hardware architectures.

As will be explained in the next chapter, in applications in which latency is more critical, data flows with complex structures can still be functionally and spatially decomposed and mapped to pipeline–connected C40 networks by constructing a run–time system which exploits the advanced features of the C40 DMA co–processors to route all of the communication through the pipeline connections. The specialized routing support adds complexity to the run–time system, and decreases its architecture independence.

Another approach would have been to require the hardware inter–connection topology to mimic the structure of the decomposed data flow. However, this would requiring the hardware topology to be changed whenever the data flow structure changes. Since it has been assumed that the software reconfiguration will occur frequently, and that the implementation should be automatic, requiring no changes to the hardware inter–connection network, this was not considered a viable option.

There is a trade–off between (1) simplicity in the hardware topology and allocation scheme and (2) lower complexity and architecture independence in the run–time system. Using a simple hardware pipeline with a more complex run–time system greatly simplifies the mapping algorithm. Note that the discussion is not about which real–time applications should be supported. It has already been established that both low latency and high throughput applications are necessary, and that complex data flows will be supported. The issue is whether it is more important to have a simple mapping algorithm or an more architecture–independent run–time system.

To decrease the search space of data flow to network mappings, I determined that the decomposition and allocation algorithms should be simplified if the automatic mapping of processes to processors is to be practical. This simplification in the mapping algorithm was made possible by adding complexity to the run–time system. Specifically, a special routing technique was implemented for the C40 which enables all communication to be routed along a hardware pipeline.

The run–time system was kept semi–architecture independent by implementing the communication components as a separate layer which can be re–implemented for new hardware architectures, thus re–using a large part of the implementation. In the next chapter, both a prototype C40 run–time system and the resulting mapping algorithm which were designed and implemented for this work will be discussed. It will be seen that special support in the run–time system reduced the complexity of the mapping algorithm significantly, making the interpretation process more feasible.

CHAPTER V

MIRTIS: A PROTOTYPE SOLUTION

In this chapter, I present a prototype MGA–based environment for real–time image processing. The system, called MIRTIS (Model Integrated Real–Time Image Processing System), is a realization of the ideas which have been developed in the previous chapters. It uses a combination of automatic program translation and meta–level driven software synthesis to automatically parallelize image processing data flows made up of sequentially coded algorithms, and then scale and allocate the decomposed data flows to the underlying parallel resources. The decomposition and scaling decisions are driven by the real–time constraints, which are modeled explicitly.

I will first discuss the overall MIRTIS architecture. Then I give an overview of the modeling paradigm. Next, I describe the prototype run–time system developed and implemented on C40s. After a brief description of the image processing library, I then outline the design of an interactive run–time graphical user interface for the system. In the last section, my discussion centers on the main component of the MIRTIS system, the model interpreter. In that section, I describe the interpretation algorithm.

The MIRTIS Architecture

The MIRTIS system architecture, shown in figure 14, follows the basic model–based system architecture shown in figure 7. The design consists of the following components

- The IPDL–VPE model building environment
- The model database
- The MIRTIS model interpreter
- An image processing application library
- The PCT–C40 run–time system
- The MIRTIS graphical user interface
- A Network of C40s

The IPDL Modeling Paradigm

The MIRTIS modeling paradigm, called IPDL (Image Processing Description Language) was designed specifically for real–time image processing. The concepts were developed by extracting the set of information required to support the automatic decomposition and mapping approach outlined in the previous chapter. This section will briefly describe the paradigm, putting emphasis on the novel concepts. A detailed explanation of the paradigm with examples of each type of model is given in the appendix.

IPDL contains three types of models, *Signal Flow*, *Hardware*, and *Constraints*, which represent the data flow computation to be performed, the hardware resources available for the solution, and the timing constraints required by the solution, respectively (see table 3). The combination of a Signal Flow Application model, a Hardware Network model, and a Constraints RealTimeConstraints model together form a the specifications for a real–time image processing *system*.

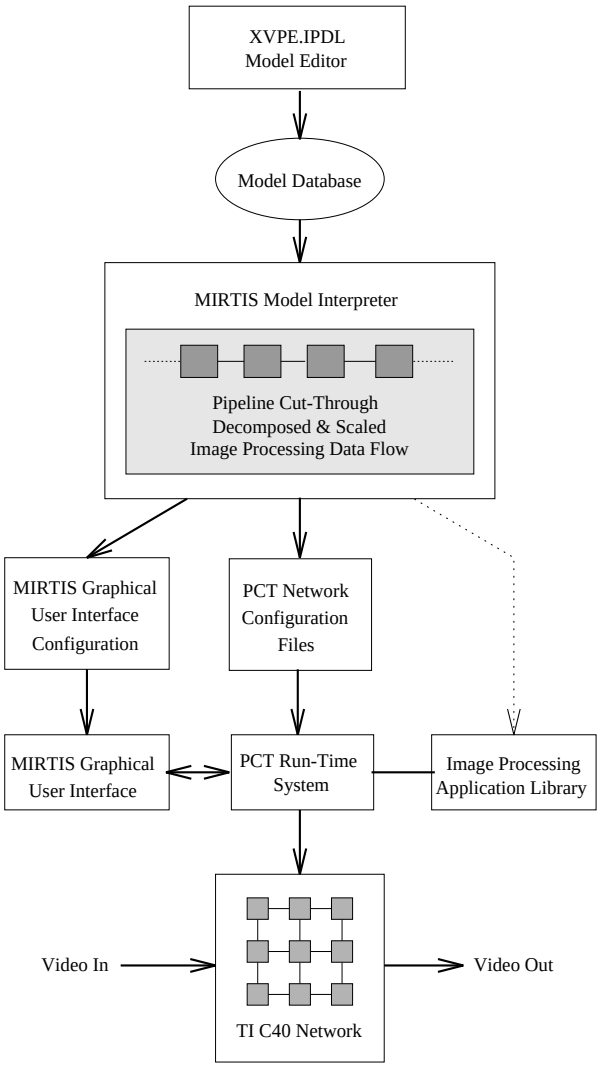

Figure 14: The MIRTIS Architecture

Data Flow Models

Data flow models are used to specify the image processing computations to be performed. The two types of data flow models are *Application models*, and *Algorithm models*.

Application Models

The computations to be performed are represented by an *application model*, which is a data flow graph made up of image processing *algorithm models* (see table 4). The

IPDL Paradigm		
Paradigms	Models	Model Aspects
SignalFlow	Algorithm	Structure Constraints DataDependency ParameterInterface
	Application	Structure
Hardware	Node	Hardware
	HostNode	Hardware
	Network	Hardware
Constraints	RealTimeConstraints	Goals

Table 3: IPDL Modeling Paradigm

idea of specifying an application as a graphical data flow is also used in the Khoros image processing environment, and is widely accepted as an intuitive and easy to learn approach.

Connections between the algorithm models represent the flow of data between the algorithm outputs and inputs. Connections which transfer image data are referred to as *ImageFlow* connections, and those which transfer other types of data, such as integers, floats, and arbitrary arrays are called *SignalFlow* connections. The distinction between image and signal connections is used in the partitioning stage of the interpretation algorithm, which creates a top–level functional decomposition satisfying the *PCT Partition* constraint, which will be discussed later.

Application Models	Object Name	Structure → Object Description	
Attributes			
Parts	Algorithms	algorithm models from the image processing algorithm model database	D
Connections	Signal Flow	connections between algorithm output signals and input signals: flow of non–image data	D
	Image Flow	connections between algorithm output image streams and input image streams: flow of image data	D
Conditionals			

Table 4: Signal Flow Application Models

Algorithm Models

Each algorithm in the image processing library has an associated algorithm model, which provides various types of information about the algorithm. Table 5 shows a breakdown the algorithm model parts, attributes, and aspects. Although only the most important of these components will be discussed in this section, a complete discussion is given in the appendix.

Data Dependency Specification:

When implementing temporal or spatial decomposition, the interpreter must be able to determine how the input data is to be split and allocated to the worker processors so it can (1) configure the communication system correctly, and (2) accurately model the communication overheads.

For a given algorithm following the general image processing algorithm definition shown in figure 2, the *total data dependency set* for a given output image pixel is

Algorithm Models

Parameter Interface
Data Dependency
Constraints
Structure

	Object Name	Object Description				
Attributes	Library Function	name of the ip library function	D			
	Partial Results Combo Method	only concatenation is supported	D			
	Memory Constraint	memory required in kbytes		D		
Parts	Input Images	input image data sequence	D		I	
	Output Images	output image data sequenc	D		I	
	Input Signals	non–image input data	D			
	Output Signals	non–image output data	D			
	Benchmarks	empirically gathered execution times for various data sizes		D		
	Requirements	the algorithm's processor or resource requirements		D		
	NodeRefs	reference to a hardware node		D		
	NodeResourceRefs	reference to a node's resource		D		
	HostNodeRefs	reference to a host node		D		
	HostNodeResourceRefs	reference to a host's resource		D		
	Data Dependencies	mathematical specs of the way the algorithm accesses data			D	
	Flag Parameters	dynamic booleans (switchs)				D
	Continuous Parameters	dynamic numerical parameters (scroll–bars)				D
	Select Parameters	dynamic multiple–choice parameters (pull–down menus)				D
	String Parameters	textual parameters (text boxes)				D
	Parameter If Modes	groupings of parameters				D
Connections						
Conditionals	BenchmarkToNodeAssociations	specifies the benchmark node		D		
	RequirementToResource Associations	specifies a node, hostnode, or resource requirement		D		
	ModeToParameterAssociations	specifiess which parameters are active in each mode				D

Table 5: Signal Flow Algorithm Models

the set of all locations in the input image sequences which will be accessed directly in computing that output pixel. (For simplicity, assume an algorithm with a single image input and output.) If the algorithm is *well behaved*, as has been assumed, then this set of locations will define a 3-dimensional region in the input image sequence.

In order for worker n to compute it's partial result, the communication system must place the total data dependency set for each output pixel to be computed by worker n in it's local memory. The total region of the input seen by that worker will

be the union of the total data dependency sets of each of it's local output pixels, and the total amount of overlap data between worker n and worker m is the intersection of their total local regions. These quantities are used directly in the characterization of the communication overhead in the performance models, the details of which are described later.

For modeling the data dependency behavior of the algorithms, I have devised a mathematical dependency specification format which is unique to this development. The idea is to show a mathematical relationship between an output pixel location and a region in an input image sequence. The data dependency specification is contained in a textual attribute of the algorithm model (see table 5), the format of which is given by:

$$Out[rvar, cvar, tvar] \longleftarrow In[rowrange, colrange, timerange]$$

where Out and In are names of two of the algorithm's image signals, $rvar$, $cvar$, and $tvar$ are dummy row, column, and time index variable names, and $rowrange$, $colrange$, and $timerange$ are *range specifications*. A range specification has the following format:

$$begin()\ [...end()] \quad | \quad ":"$$

where $begin()$ and $end()$ are algebraic formulas specifying the first and last indices of the range, and a ":" specifies the entire range of valid indices.

The data dependency specification defines a dependency relationship between output pixel location $Out(rvar, cvar, cvar)$ and the locations lying within a 3–dimensional box–shaped region in the input sequence space. This box is defined by $row, col, time$

such that each of the following are true:

$$row_{begin}() \leq row \leq row_{end}()$$

$$col_{begin}() \leq col \leq row_{end}()$$

$$time_{begin}() \leq time \leq time_{end}()$$

The simplification to a box–shaped region is not restrictive, since any 3-dimensional region in the input sequence can be enclosed by such a region, and most image processing algorithms access data in box–shaped regions anyway. In the case that the box specification includes input locations which are not actually accessed by the algorithm, the only affect will be extra communication overhead.

Note that any of the algebraic range formulas can depend upon any of the dummy index variables, possibly non–linearly, which implies that this specification supports dependencies for which the relative position and dimensions of the dependency box varies with position in the output image sequence (e.g. the data access patterns could vary from top to bottom in the output image plane).

To further clarify this definition, a few examples are necessary. The data dependency specification for the $5x5$ convolution algorithm is given below, assuming that the convolution model's input and output image signals are named In and Out, respectively.

$$Out[i,j,k] \longleftarrow In[i-2...i+2, j-2...j+2, k]$$

This dependency specification defines a rigid $5x5x1$ dependency box located in the input sequence, centered around the input pixel $In(i,j,k)$.

To demonstrate another feature of the specification format, suppose that the algorithm is a non–neighborhood operator, such as a global histogram operation. The

data dependency specification would be given by:

$$hist[n,k] \longleftarrow In[:,:,k]$$

This specification includes the total range specifier ":", which indicates that all locations along that dimension are required. In other words, calculating the nth element of the kth histogram requires all pixel locations in the kth input image.

The final example given is the application of the specification format to a 2–D decimation algorithm which decimates each image in the row and column dimensions by constant integer factors. In other words, a (R, C) decimation divides the number of rows by R and the number of columns by C. The data dependency specification for this algorithm would be given by:

$$Out[row, col, frame] \longleftarrow In[row/R, col/C, frame]$$

In this case, each range specification is a single formula, which implies that each output pixel depends upon a single input pixel (located at a different position in the input sequence).

A data dependency parser was implemented which parses and interprets the data dependency specifications during interpretation and determines the data access patterns for each algorithm in the data flow. When the decomposition is performed, the data dependency information is used in characterizing the amount of communication and sharing overheads for the performance models.

Benchmarks:

The approach taken in modeling an algorithm's execution time as a function of data size was to rely upon empirically gathered benchmarks. Using measured

execution times instead of other approaches is both more accurate and straight–forward.

Each algorithm model contains a set of *Benchmark* parts (see table 5), each having an attribute specifying the dimensions of the algorithm's input images for that measurement. By providing execution time benchmarks for various data sizes, an execution time versus data size curve can be constructed. The method of estimating the execution time of an algorithm for a particular data size is to use linear interpolation on the benchmarked data sizes. If only a single benchmark has been provided, then linear interpolation is done between that benchmark and an implicit benchmark of zero seconds for zero data size. This unique benchmark–based interpolative approach to approximating execution time has proven to provide very accurate results in testing.

Note that in the models, each benchmark is related to a particular node type by association with a reference to a *Hardware Node* model (see BenchmarkToNodeAssociations in table 5). Thus, separate execution time curves can be formed for different processor types, and the interpolation will be done on the benchmarks related to the node type being used.

Parameter Interface Aspect:

The parameter interface aspect of the algorithm model defines the algorithmic parameters which can be changed dynamically during run–time. The names of the parameters, their possible values, the type of graphical widget which will be used to adjust the parameter in the GUI, and the format of the messages the GUI must send to the run–time system to update the parameter are specified in each parameter

model. The details of this specification are not given here, but some detail is given in the appendix. Similar specifications of adjustable parameters are used in the Khoros environment. However, the mechanism used in updating the parameters of the running real–time computation is unique to this system.

Hardware Models

The types of hardware models are *Node models*, *HostNode models*, and *Network models*.

Network Models

Network models are inter–connected hierarchies containing nodes, hostnodes, and other networks (see table 6).

Network Models	Object Name	Object Description	Hardware →
Attributes			
Parts	Nodes	node models	D
	Host Nodes	models of host nodes: PCs and workstations which can boot the system, provide disk I/O, or may host other I/O devices	D
	Sub Networks	other network models which are contained by this network (there is hierarchy)	D
	Connectors	passive connections between the nodes in this network and the nodes of other networks	D
Connections	HW Connections	connections between the node comm ports, hostnode network interfaces, and sub network connectors which form the communications network	D
Conditionals			

Table 6: Hardware Network Models

Node and HostNode Models

Node Models		Hardware → Object Name / Object Description	
Attributes	CPU Type	C40 or C44 processor	D
	CPU Perf	performance rating of the CPU in MFlops	D
	MemType	type of memory (wait–states)	D
	MemSize	amount of memory on each bus ("Mbytes x Mbytes")	
Parts	CommPorts	I/O ports: have speed rating and port type	D
	Resources	device enabling special capability (e.g. grabber or display hardware)	D
Connections			
Conditionals			

Table 7: Hardware Node Models

Node models represent the network nodes which perform the image processing computations on the data stream. In the case of the prototype, these are C40 DSP modules. Each node model has attributes describing the node type and configuration, the ports, and any special resources (see table 7).

HostNode models represent the PCs or workstations which provide services such as network loading and disk I/O, or run the parameter adjustment GUI. They contain various attributes and parts, notably host interface card parts (see table 8). These represent cards located in PC bus slots that provide communication links to the C40 network through which loading and run–time communication occur.

Constraints Models

The types of timing constraints relevant to real–time image processing are latency and throughput. Thus, constraints models contain latency and throughput parts (see

HostNode Models		Hardware →	
	Object Name	Object Description	
Attributes	Host Speed	CPU clock speed of host computer in MHZ	D
	Host Type	type of host computer: X86 PC, Pentium, or Sun workstation	D
Parts	Resources	relavant resources, such as real–time disks and other host–based video I/O devices	D
	Host Interfaces	cards which provide a C40 port interface, or a memory mapped C40 interface through which the network can be booted and otherwise interacted with: there are several commercially available PC host interface cards	D
Connections			
Conditionals			

Table 8: Hardware HostNode Models

table 9).

Throughput models have a numerical attribute specifying frame rate in $\frac{frames}{sec}$, and latency models have attributes specifying latency in *frames*. Depending on the application, if the interpreter determines that the specified constraints cannot be met using the available resources, it may be acceptable to relax either the latency or throughput constraint, or to reduce the size or frequency of the image data source. These ideas are conveyed with boolean attributes of the throughput and latency constraint models which specify whether that constraint is a *hard*, or *soft* constraint. The constraints model itself also has an attribute specifying whether the application is *hard real–time*, meaning no concessions should be made in performance, or *soft real–time*, meaning that one or more of the timing constraints may be relaxed, or the data source can be down–sampled, if necessary.

Real–Time Constraints Models			
			Goals
	Object Name	Object Description	
Attributes	FailureMode	nature of the real–time constraints: hard real–rime or best effort	D
Parts	Throughput Constraints	the target throughput in frames per second and specification of whether the throughput can be relaxed if necessary	D
	Latency Constraints	the target latency in frames and specification of whether the latency can be relaxed if necessary	D
Connections			
Conditionals			

Table 9: Real–Time Constraints Models

Applying the IPDL Modeling Paradigm

The XVPE graphical model building environment has been configured with the IPDL paradigm, and several example models have been built. The MDF configuration file and more information about the IPDL paradigm, including example models, are given in the appendix.

Of course the models themselves are not of very much use without some form of execution environment and interpreter which can implement the system that has been modeled. The largest part of the work that has been done is building the PCT–C40 parallel run–time system, and the MIRTIS model interpreter. The job of the interpreter is to transform the system models into an implementation which runs on a C40 network under the control of the PCT–C40 kernel.

Before describing the interpreter, I first will explain the details of the PCT–C40

run–time system, which provides the underlying parallel execution support. After that, the reader should have some understanding of the high–level aspects of the system (the models) and the low–level aspects of the system (the run–time system and the C40 network). The final section will explain how the interpreter provides a path between these levels which allows the user to specify applications in the high–level modeling environment and generate the real–time implementations automatically.

The PCT–C40 Run–Time System

A real–time image processing kernel called PCT–C40 has been implemented which provides run–time support for data parallel execution of image processing synchronous data flows on pipeline–connected C40 networks. The kernel runs on each C40 node and performs the scheduling, communication, and synchronization necessary for data parallel computations.

The scheduler on each node runs a Periodic Admissible Sequential Schedule (PASS) which implements the synchronous data flow local to that node. The kernel configures and starts the PCT communication engine, which, in cooperation with the neighboring nodes, distributes the input data appropriately across the processors and combines the local results to form the output data.

Since the computation and communication schedules are static, the scheduler introduces minimal run–time overhead. This also has the effect of simplifying the kernel by pushing the work of generating computation and communication schedules into the model interpretation process.

In the following sections, the PCT–C40 run–time system details are given. First the concepts and C40 implementation details of the PCT communication scheme are

explained. Then the details of the scheduler and how it is synchronized with the communication engine are given. A summary of the PCT–C40 kernel follows.

Pipeline Cut–Through

Pipeline Cut–Through (PCT) is a communication technique which allows synchronous data flows to be parallelized with the spatial or temporal data parallel constructs and mapped to a group of C40s connected in a pipeline. PCT automates the distribution of input data, the collection of partial results, and the coordination between the communication and computation processes, so the data parallelism is absolutely transparent to the programmer.

Related Techniques

The name pipeline cut–through was chosen due that fact that the technique routes all communication along the hardware pipeline by "cutting through" intermediate processors. The technique is similar to two general message routing techniques, *circuit switching* and *wormhole routing*[31], in that messages between non–neighboring nodes are routed *through* intermediate nodes with no local buffering. During transfers, messages pass through intermediate nodes without entering the local memory or interrupting the computation. The largest difference between pipeline cut–through and these techniques is that they are designed for routing asynchronous messages in more general inter–connection topologies. The messages can vary in length, and can be sent at any time. In pipeline cut–through, the message traffic patterns are known prior to run–time, and transfers are routed along the pipeline by *coordinating* the communication engines to execute pre–determined transfer sequences. PCT is not a

general message passing routing algorithm, and is unique to this development.

Before discussing the PCT technique and its implementation details, a few key concepts are defined and some background information about the C40 is given.

PCT–Related Definitions

Partition:

A *partition* of a synchronous data flow graph is a grouping of the computations into *blocks*. Usually the assumption is that each block of a partition will be mapped to a different processing node. Data flow partitioning is usually used for balancing computation and communication loads on the processors to maximize efficiency. The synchronous data flow graph shown in figure 15 has been partitioned into 3 blocks.

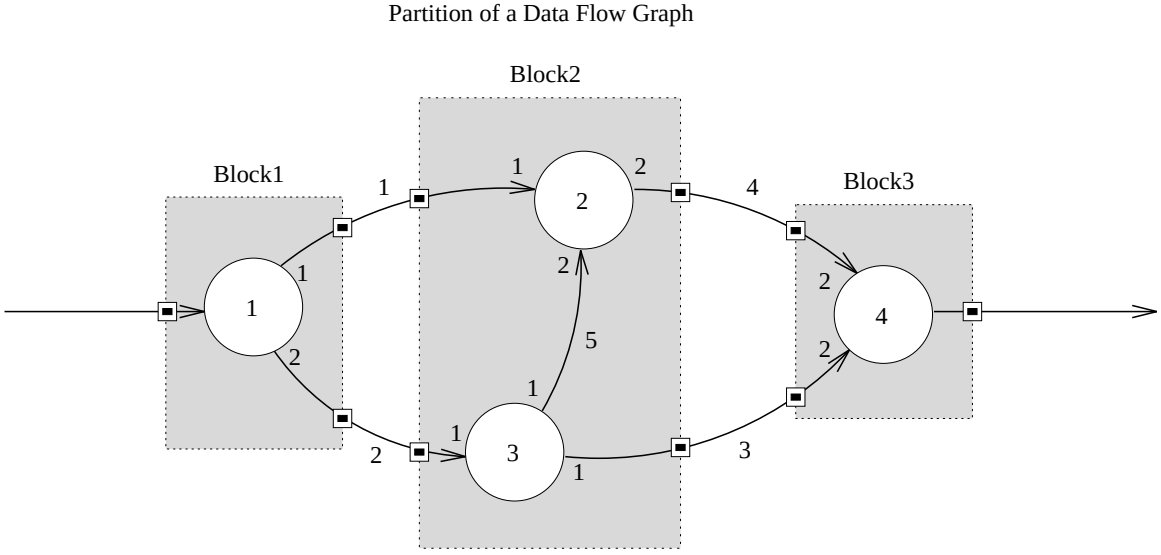

Figure 15: A Partition of a Synchronous Data Flow

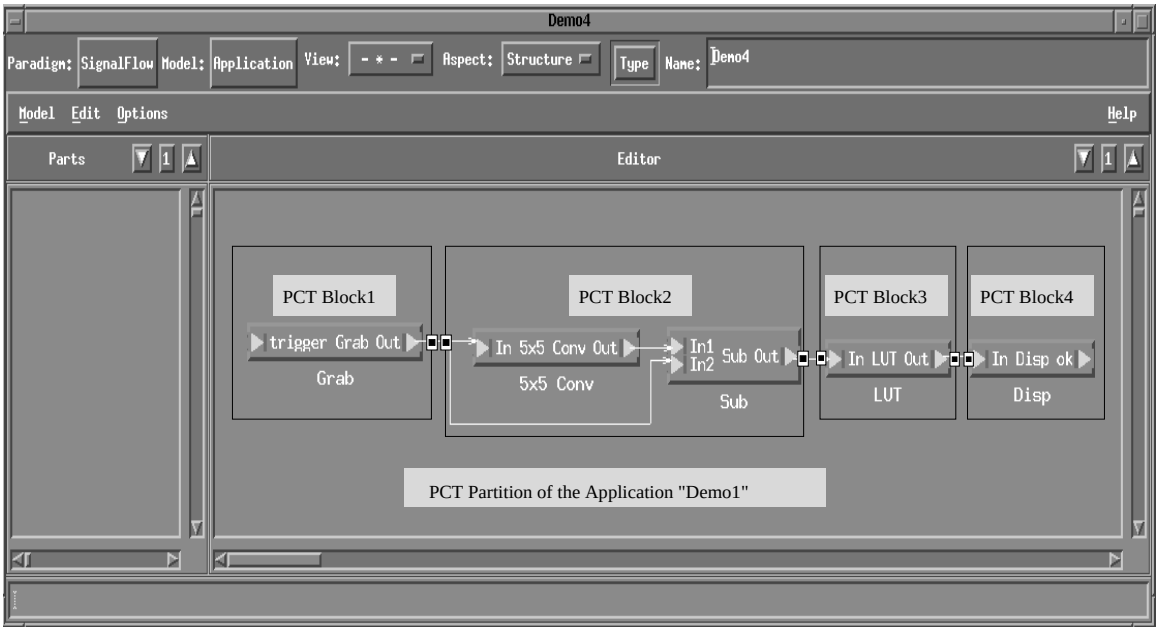

Figure 16: A PCT Partition of an Image Processing Synchronous Data Flow

PCT Computation Block:

A *PCT Computation Block*, or *PCT Block*, is an image processing data flow sub–graph which meets the following constraints:

- The sub–graph has at most one input connection and one output connection.
- The input and output connections transfer image data.

The reasoning behind these constraints is given later in the text.

PCT Partition:

Given an image processing synchronous data flow (with each algorithm following the previously discussed general image processing algorithm definition), a *PCT partition* is a partition for which each computation block is a *PCT Computation Block*. A PCT Partition of the application model *Demo4* into 4 *PCT Computation Blocks* is shown in figure 16.

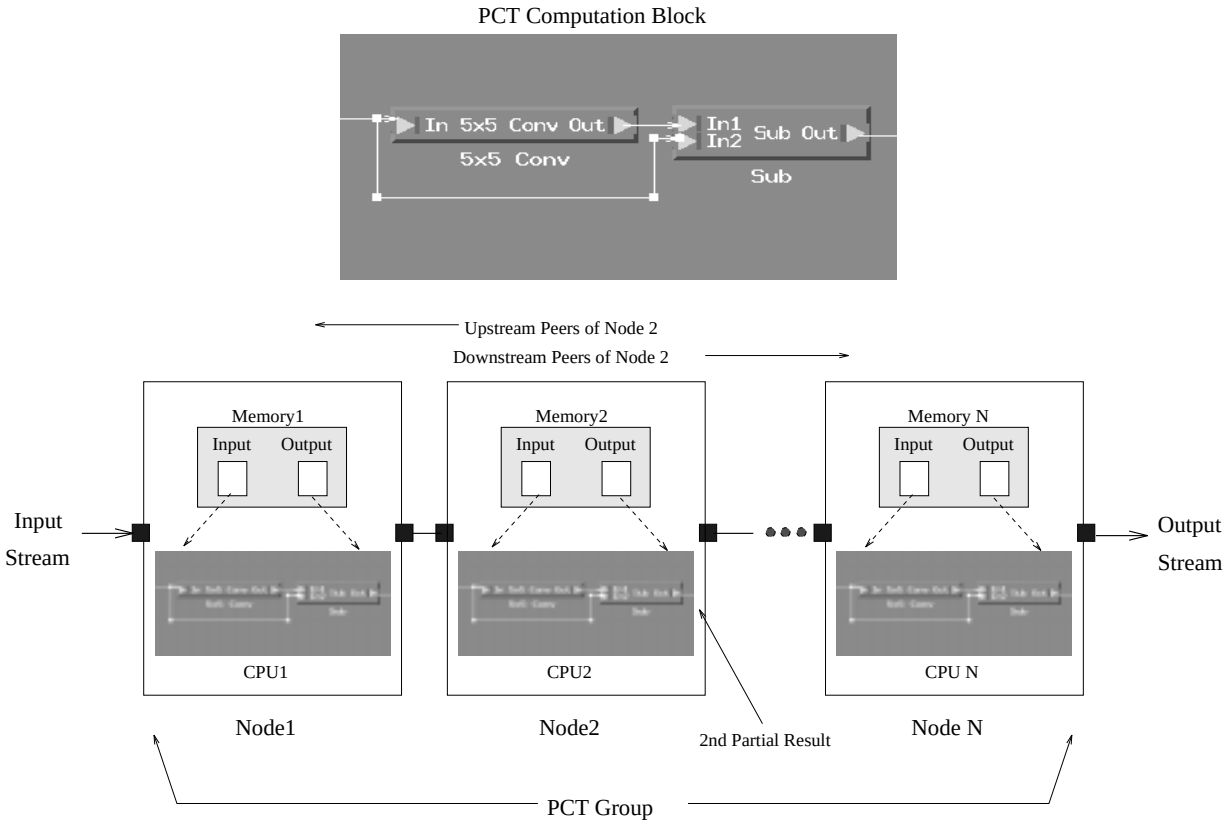

Figure 17: Demonstration of the PCT Group Concept

PCT Group:

N pipeline–connected nodes working together to data parallelize a PCT computation block are referred to collectively as a *PCT group*. A PCT group together with the PCT block it is implementing are shown in figure 17.

Upstream/Downstream Peers:

Given node k of an N–node PCT group, the nodes in the pipeline which are closer to the source data stream are referred to as *upstream peers* of node k, and those farther from the source data stream are referred to as *downstream peers* of node k (see figure 17).

Partial Result:

The work load is being spread across the nodes by decomposing the data stream. For any given cycle, node k will compute a subset of the output data referred to as the *kth partial result* (see figure 17).

Required Input:

Referring to the general image processing algorithm described in figure 2, the computation of the *kth* partial result requires the values of the pixels from a subset of the possible input image pixel locations. These locations together form a *region* within the input image sequence. The set of input data required is referred to as the *required input* of node k.

Shared Input:

A pixel is said to be *shared* if it is part of the required inputs of two or more nodes in a PCT Group. The number and positions of shared pixels depends upon the data dependency properties of the computation, the number of nodes in the PCT group, and which data parallel decomposition method is being used (spatial or temporal).

Intermediate Results:

Connections internal to a PCT Block are referred to as *intermediate results*. Referring to figure 16, the connection between *5x5 Conv* and *Sub* is an intermediate result local to *PCT Block2*.

SIMIOAC (Simultaneous I/O and Computation):

The PCT run–time system supports two forms of process/computation synchronization, one in which the computations are performed simultaneously to the I/O communication, and one in which they are not. This option will be referred to as SIMIOAC.

Decomposition Method:

The way in which the computation is being decomposed across the PCT Group. The supported methods are *Sequential*, *Temporal*, and *Spatial (rows)*, which will be discussed in the text.

Decomposition Alternative:

The combination of a Decomposition Method and a SIMIOAC setting. There are six supported decomposition alternatives (3 Decomposition Methods * 2 SIMIOAC Settings).

C40 Background

The C40 DSP was designed to be used as a parallel system building block. It has 6 communication ports and 6 autonomous DMA co–processors on the chip which can transfer data either (1) from memory to memory, (2) from a communication port to memory, (3) from memory to a communication port, or (4) directly from a communication port to another communication port[1]. The DMAs can operate independently of the CPU, which enables the communication and computation processes to occur

[1]This is possible since the C40 communication ports are memory mapped devices, and the DMA can be programmed to transfer directly to or from the port address. In this mode, however, the port synchronization features do not work, so care must be taken not to create a deadlock.

simultaneously.

A powerful feature of the C40 DMA engines is that each DMA can be set up to automatically "re–program itself" when a transfer terminates (see *Fun With Link Pointers* in [50]). This feature, called link pointer auto initialization, allows the DMAs to be programmed with link pointer tables and interrupt routines so that they operate as autonomous communication engines.

PCT Overview

With these definitions and background information out of the way, the next sections will explain how the PCT–C40 technique works and how it was implemented.

Each node of a PCT group (a pipeline–connected group of C40s) will perform the same computations on a different section of the image data. The incoming stream must be split and spread across the memory banks of the group nodes, and after the local data flow computation has produced the partial results, they must be combined (merged) to form the output data stream. The type of decomposition that is being performed, and the data dependency requirements of the algorithms in the data flow determine exactly how the data stream will be split and merged. As well as distributing the decomposed data across the memory banks, the communication engine must also support the sharing of pixels from the input sequences between two or more nodes in a PCT group.

PCT Communication State Machines:

The distribution of the required inputs (including replication of shared pixels) and collection of partial results in a PCT group is achieved by programming two

of the DMAs on each node to operate together as a *synchronous communication state machine*. A PCT communication state machine has four possible states (see figure 18):

1. *receive & send:* read input data from input port to memory and simultaneously send partial results from memory to output port

2. *insert:* send copies of local input pixels which are to be shared with downstream peers to the output port

3. *forward:* send data from input port to output port (cut–through).

4. *idle:* cycle has terminated and communication engine is idle, waiting to be restarted

The implementation is achieved by programming the DMAs with link tables so that they execute the correctly timed sequences of *receive & send*, *forward*, and *insert* that cause the input data stream to be distributed appropriately across the memories of the group, and the output stream to be assembled.

Each PCT Group node's communication engines are synchronized not only with the local computation, but also with its upstream and downstream peers so that together they work as a larger state machine. The operation of a PCT group communication state machine varies depending upon the decomposition alternative being used. The operation of the PCT group communication will be examined for each decomposition alternative in the following sections.

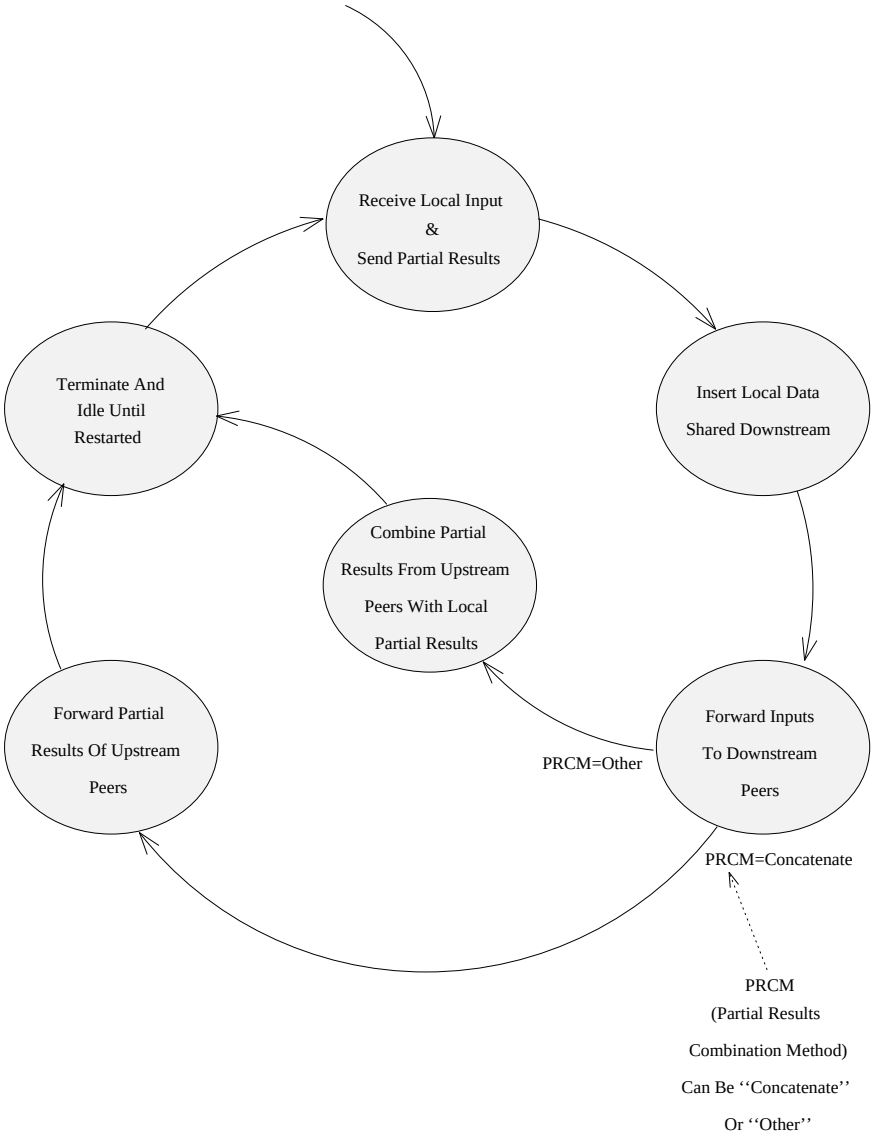

Figure 18: PCT Communication State Machine

Shared Input Data:

The most difficult part of implementing data parallel computations is dealing with the shared data. One approach could have been to support dynamic asynchronous message passing in the communication system so that shared data could be transferred between the PCT group nodes during the computation. However, this approach would add complexity to the communication system, and eventually to the mapping algorithm, since the mapping would have to take into account the strong relationship

between the routing of the data sharing messages and the performance of the resulting implementation. To avoid this problem, the approach used by PCT is to place a copy of each shared input pixel in the local memory of every node requiring it during the splitting phase, before the computation begins. The determination of which nodes require which pixels is done by constructing and analyzing the local requirements sets. This technique eliminates the need for run–time inter–peer communication and simplifies the run–time communication system and the mapping algorithm. The down side is that the replication adds communication overhead.

Shared Intermediate Results:

Similarly to the sharing of input data, sharing of the intermediate results may also be required. This occurs when there is overlap between the regions of an intermediate result required by two or more processors. The overlap data must somehow be shared by the processors.

For example, suppose a PCT block contains two algorithms in series, the second being a $5x5$ convolution, and suppose that spatial decomposition is being used. Each copy of the convolution algorithm will compute a sub–image of each output frame, and will require a region of each input image which is slightly larger than the output region it computes. To be exact, two extra rows must be added on the top and bottom of any horizontal cut, and two extra columns must be added to the left and right of any vertical cut. Thus, there are regions in each frame of the convolution's input image sequence which must be known on more than one of the group nodes. In other words, regions of the intermediate results sequence must be *shared* between peers.

The run–time system could have been designed to support shared intermediate results regions by exchanging the data during the computation of the group schedule via dynamic asynchronous message passing. However, this would cause unpredictable performance behavior and would add complexity to the system. The mapping algorithm would have to take into account the placement of processes and the routing of these run–time messages.

The way that PCT implements sharing of intermediate results is by replicating the computations. In other words, the values of shared intermediate results are calculated locally to each node on which they are required. This obviously introduces computation overhead, but in contrast to the alternative, it was seen to be the best method of keeping the generality without jeopardizing the performance predictability of the PCT kernel and the plausibility of the mapping algorithm. The overhead of replicating computations will show up in the performance models, which will tend to *discourage* decompositions which have large amounts of shared intermediate data.

PCT Details

The communication state machines are implemented by taking advantage of DMA link pointer auto initialization and DMA interrupt service routines. The communication engine can be synchronized with the local computation in one of two ways: the I/O communication and computation can occur at different times, or can occur simultaneously. Whether or not the I/O and computation will be done simultaneously is an option called *SIMIOAC* in the PCT configuration. Obviously, if I/O and computation are simultaneous, the input and output images will have to be buffered locally, since the computation should not work on an input image until it has been

fully received.

The following examples demonstrate the operation of the PCT communication engine and how it works in conjunction with the local computation, both with and without simultaneous I/O and computation. In each case, a pipeline–connected PCT Group of N C40 nodes is shown implementing a PCT Computation Block called *Comp(in,out)* on the image data stream with spatial data parallelism.

Operation Without Simultaneous I/O and Computation:

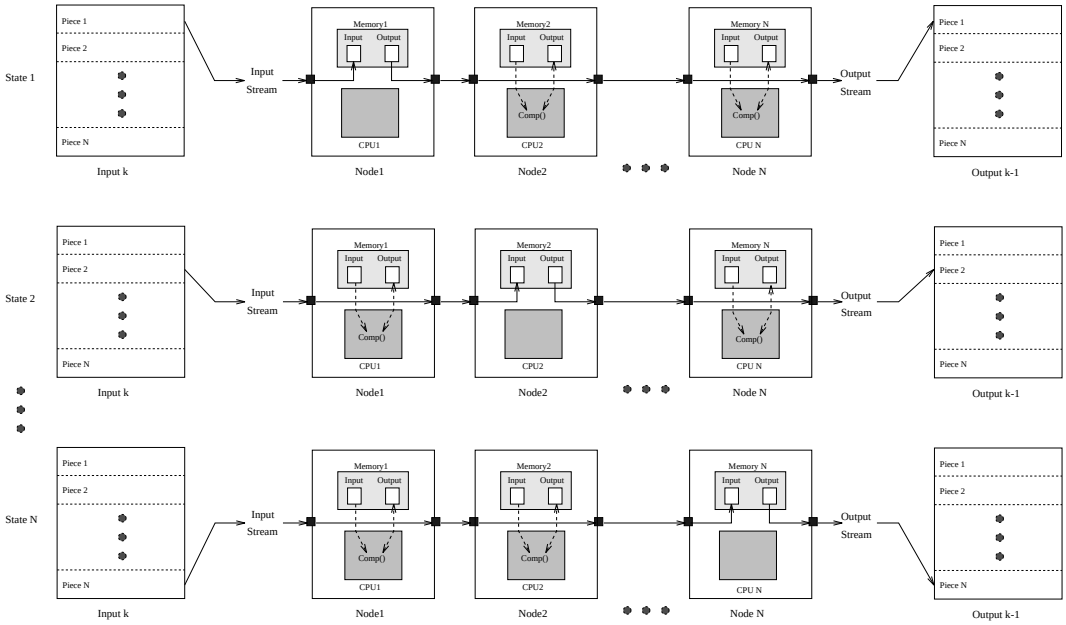

Figure 19: Pipeline Cut–Through Without Simultaneous I/O and Computation

The communication and computation states of the PCT mechanism shown in figure 19 represent the case where the communication and I/O computation processes are not simultaneous. Any particular node's state cycle begins with receiving a local input image from the input port and sending the partial results of the previous computation to the output port. After this I/O state has terminated, the computation is begun with the newly received input, and the communication engine begins

forwarding data from the input port to the output port. The computation and communication processes are synchronized before the next I/O state can begin. At any given time, one of the nodes is in its I/O state. The rest are in the forwarding state. Each of the upstream peers is forwarding the input data stream from its input port to its output port, and each of the downstream peers is forwarding the output data stream from its input port to its output port. Also, during this time, each upstream and downstream peer is simultaneously performing *Comp(in,out)* on its local input data.

Effectively, the node in its I/O state is receiving data from the input stream *through* its upstream peers, and sending data to the output stream *through* its downstream peers. Through the group cycle, the effect is that the input stream is split across the memories of the N processing elements, and the N partial results are computed and concatenated to form the output stream.

Operation With Simultaneous I/O and Computation:

The operation of the PCT Group in the case with simultaneous I/O and computation is largely the same. The only differences are that each node is performing the computation at all times, and the local input and output images are double buffered. The image being split during cycle k will not be used in the computation until cycle $k+1$. For this reason, the PCT group cycle has twice as many states. Notice that a side–effect of the double buffering is that additional latency is introduced. Figure 20 shows the case in which communication and computation are simultaneous. Since the communication is concurrent to the computation, each node is very nearly always computing.

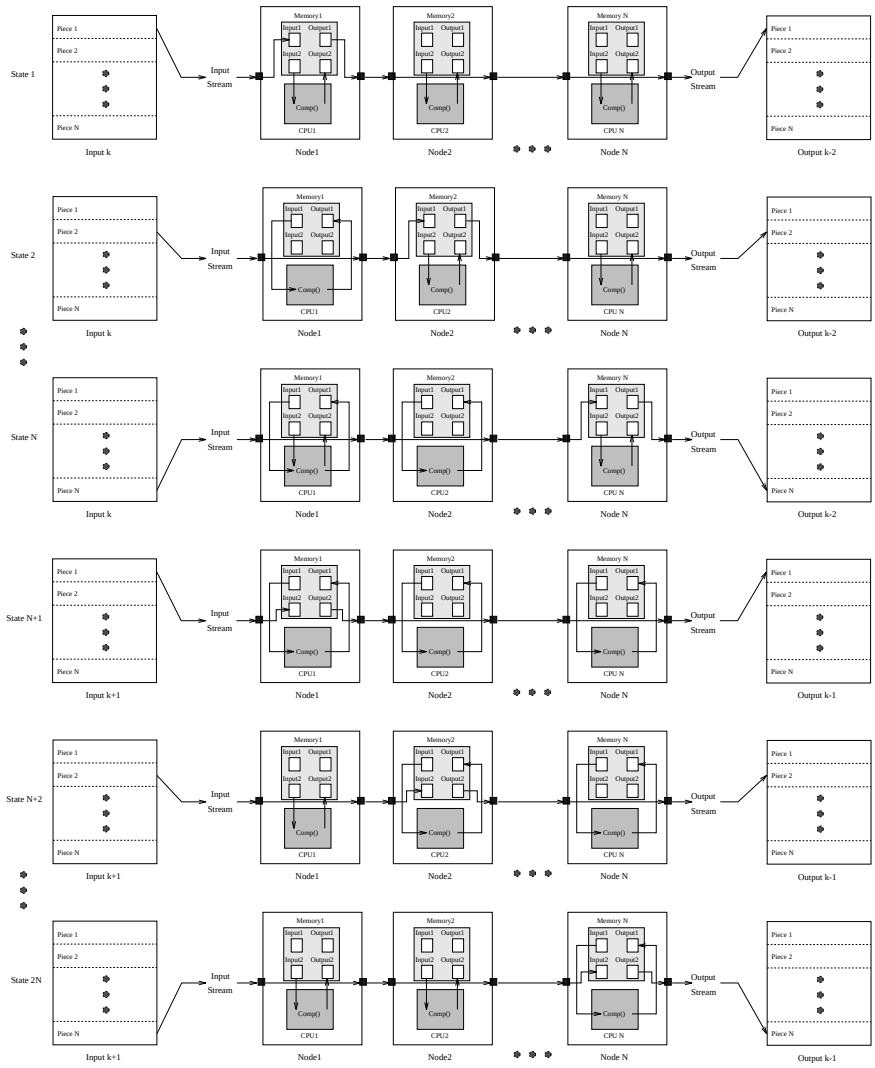

Figure 20: Pipeline Cut–Through With Simultaneous I/O and Computation

PCT Group Operation Overview:

The reasoning for supporting two methods of synchronization between the communication engine and the scheduler can be explained easily by referring to figures 19 and 20. Note that in the case that I/O and computation are not simultaneous, there are only three states, and that a particular image spends one full computation cycle in the local memory of the group nodes. However, if I/O and computation are simultaneous, the double buffering causes any particular image to spend two full computation cycles in the local memory of the group nodes. The difference is latency.

The double buffering doubles the latency of the computation. However, also note that in the case with SIMIOAC, every node is always computing, but in the case without SIMIOAC, only $N-1$ are always computing. This reduces the efficiency, and causes more processors to be used to achieve the same throughput. A more in–depth discussion of the trade–offs involved in choosing a decomposition alternative will be given later.

Note that in the example in figures 19 and 20 a simplification has been made so that the figures would be understandable. The implementation of data sharing between processors requires additional *insert* communication states in which upstream processors insert copies of local input data into the stream at the appropriate time during the splitting process, which effectively replicates sections of the data stream. There are no insertion states shown, so no input data is being shared by the processors in the examples.

Programming the DMA co–processors to achieve the appropriate PCT communication patterns is tedious. The DMA state table is programmed based on several variables which are set at run–time, including the image dimensions, the data dependency characteristics of the PCT Block being executed, the number of processors in the PCT group, the type of data decomposition being implemented, and the processor's location in the PCT group. These variable are used to determine the timing of the communication state table, which implements (1) I/O of local inputs and partial results (2) forwarding of non local inputs and partial results, and (3) duplication of shared inputs and their insertion into the data stream at the appropriate time. Although doing the configuration of the PCT DMA engines through hand coding is not feasible, the configuration can be automated, as will be seen.

The PCT run–time system implements the local PCT Block data flow computation with simple round robin static scheduler. The interpreter generates a Periodic Admissible Sequential Schedule (PASS) for each computation block, which is down–loaded to the run–time system when the system is started. This static scheduling scheme introduces almost no run–time scheduling overhead. The generation of the block schedules will be discussed in a later section.

As the PASS is run, the algorithm currently being executed receives its inputs and outputs through calls to the PCT run–time support library, and computes its outputs as if there were no parallelism. The scheduler takes care of the propagation of local intermediate results between the local producer and consumer algorithms, and the correctness of the static schedule guarantees that each time an algorithm runs, its input data will be available.

Scheduler Without Simultaneous I/O and Computation:

In that case that the I/O and computations are not simultaneous, the end of the local I/O is synchronized with the beginning of the local schedule. The communication engine is run and allowed to terminate before the scheduler is begun (see figure 21). A pseudo code version of this synchronization loop is given in figure 22.

Scheduler With Simultaneous I/O and Computation:

If the I/O and computations are simultaneous, the scheduler and the communication engine are started at the same time. Synchronization between the two must occur before the next cycle (see figure 23). Whichever one finishes first must wait on

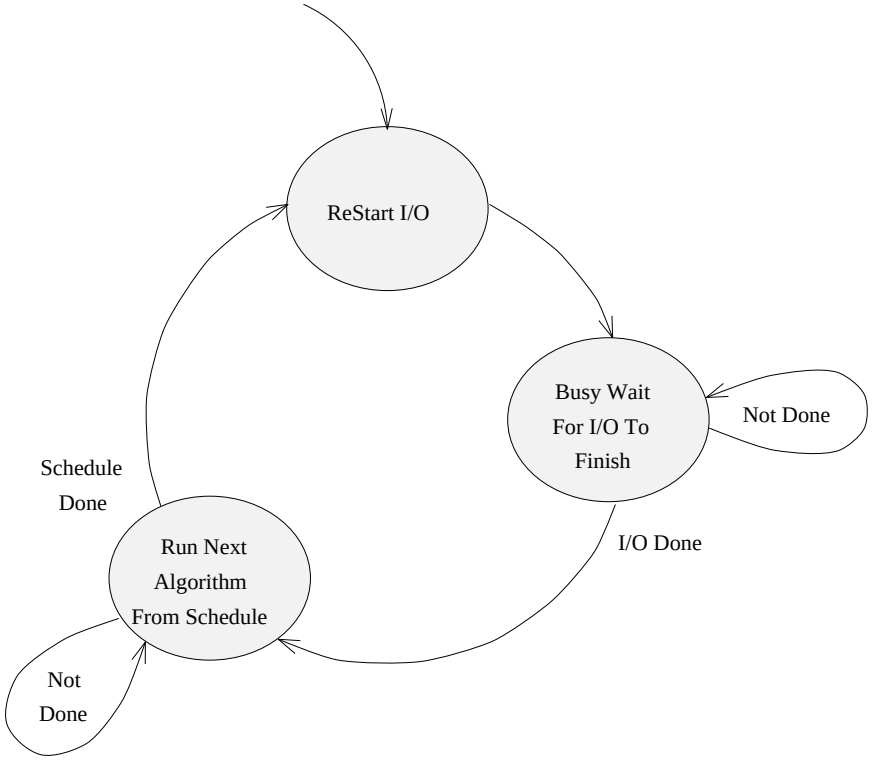

Figure 21: PCT Scheduler Without Simultaneous I/O and Computation

the other. The management of the synchronization is done through DMA interrupts from the communication side, and through busy waiting from the computation side. For further clarification, a pseudo code version of the main program loop is given in figure 24.

Summary of the PCT–C40 Run–Time System

The PCT run–time system provides the communication, scheduling, and synchronization support necessary to implement a PCT Computation Block data flow on a pipeline–connected PCT Group of C40 nodes. It is made up of a communication engine which exploits the powerful DMA engines of the C40, and a scheduler which implements a PASS data flow schedule provided to it at run–time.

The following decomposition techniques are supported (each can be used with or

```
//pct main loop without SIMIOAC

while 1 {
// receive next input and send last output
   start I/O engine
   while receiving {
     busy wait
   }
// insert shared local data
   start insert engine
   while inserting {
     busy wait
   }
//run schedule
   for each algorithm (A) in schedule {
     run A
   }
}
```

Figure 22: The PCT Synchronization Loop Without SIMIOAC

without simultaneous I/O and computation):

- Sequential: no parallelism
- Temporal data parallelism
- Spatial data parallelism (splitting into blocks of rows)
- Spatial+Temporal data parallelism (splitting into 3–D cubes)

The system can implement PCT Partitioned synchronous image processing data flows having exactly one image data source and one image data sink. The use of PCT Partitioning, which implies that each computation block has exactly one image input and one image output (the top level partition data flow is a pipeline of PCT Computation Blocks), allows the processes of decomposing and scaling the computation blocks to be decoupled from the allocation of the decomposed data flow to the

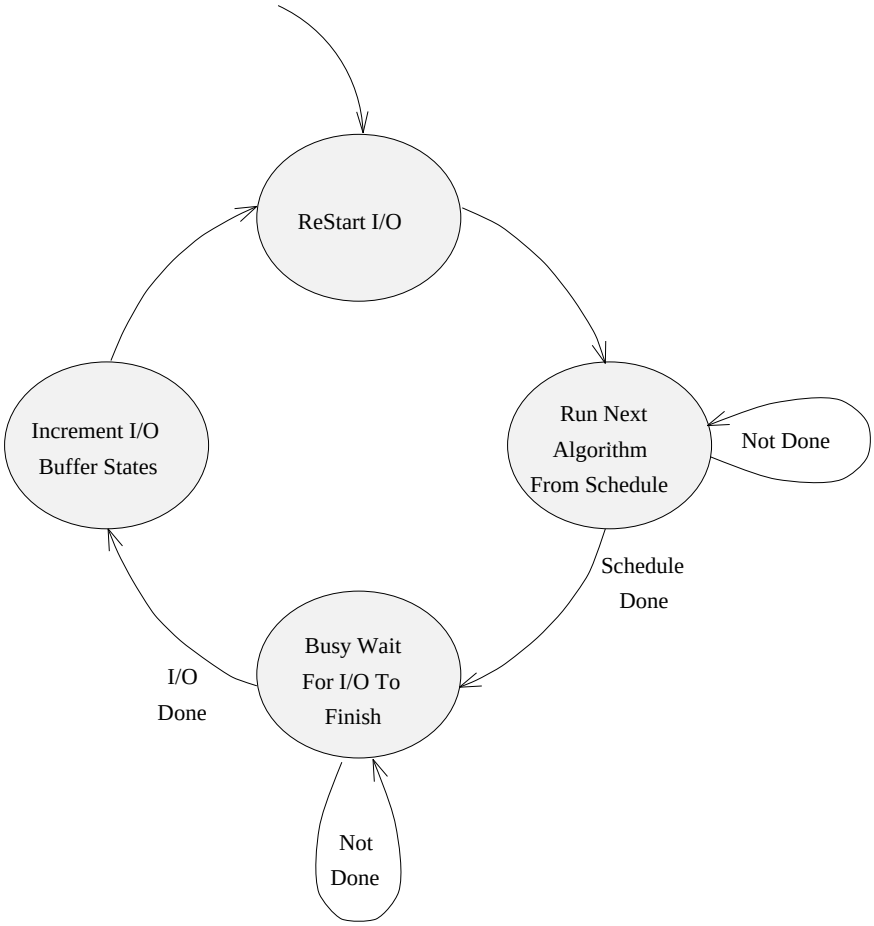

Figure 23: PCT Scheduler With Simultaneous I/O and Computation

hardware (this will be clarified later). This decoupling makes the automatic mapping process feasible.

The PCT Partitioning is not unnecessarily restrictive, since each PCT Computation Block can have a complex internal structure (several algorithms, local connections with fan outs, delay loops, etc.). The worst case is that the entire data flow, with the exception of the source and sink processes, will be in a single PCT Block. Since many image processing computations are pipeline in nature anyway, and ones that are not can always be partitioned into this form, this solution is quite flexible.

```
//pct main loop with SIMIOAC

while 1 {
// receive next input and send last output
  start I/O engine
//insertion of shared data will occur automatically
//run schedule
  for each algorithm (A) in schedule {
    run A
  }
  while (receiving or sending or sharing) {
    busy wait
  }
}
```

Figure 24: The PCT Synchronization Loop With SIMIOAC

Caveats

With the initial PCT implementation, only images can be transferred between blocks, but other data structures can be passed between algorithms internal to a block. Also, only concatenation of partial results is supported in the initial prototype.

The Prototype Image Processing Library

The actual image processing functionality is provided by a library of image processing algorithms written in C and compiled with the standard Texas Instruments C40 compiler. This library is the simplest component of the system, since the image processing functions can be written as if they were to be used in a normal uni–processor system, with the exceptions that the programs should not use global variables, and each function provides two entry points: (1) A COMPUTE function: the normal function which implements the computation, and (2) a SETUP function: an optional dynamic parameter setup function. The global variable constraint is not a hard rule,

but a strong recommendation. The COMPUTE and SETUP entry points will be discussed next.

The Compute Function

The actual image processing algorithm is implemented with a call to a normal sequential C–coded function. An application programming interface (API) to the PCT support library provides an interface through which the compute function gains access to the current and past states of input and output data buffers.

There is no mention of parallelism in the code whatsoever, so any programmer with experience with the C programming language and an understanding of an image processing algorithm can add capabilities to the system. A major strong point of the approach is that it is possible to exploit commercial image processing libraries which are optimized for the C40. This can dramatically reduce programming cost and increases performance. Note that in the convolution algorithm code given in the appendix, the COMPUTE function is merely a wrapper around a call one of the assembly coded convolution functions *aconv3x3* or *aconv5x5*, which were taken from an image processing library optimized for C40s.

The Setup Function

The parameter setup function is used in dynamically managing the context of the running algorithm. It is passed an array of tokenized arguments in the standard argc,argv format. The functionality of the setup function is actually already part of any C–coded image processing algorithm which parses the command line arguments in setting program options. The differences are (1) the command line parsing is

moved into a separate function (2) the setup function will be called multiple times. The setup function is registered with the PCT run–time system so that it can be called whenever a parameter update message is received for that function instance. The run–time system manages the dynamic setup by delivering setup messages to all nodes running the targeted function instance, parsing the formatted string into the argc,argv format, and calling the setup function with argc,argv. An example of a SETUP function (the one used for with the *conv* image processing library function) is included in the appendix.

Implemented Algorithms

The algorithms which have been implemented in the prototype image processing library are shown in table 10. For an example of the coding style and API calls, refer to the example given in the appendix.

The MIRTIS Model Interpreter

The model interpreter is the heart of any MGA system, and requires the largest implementation effort. The job of the MIRTIS model interpreter is to translate the IPDL models into a scaled decomposition of the data flow, map the decomposition to the underlying hardware architecture, and construct network communication and computation schedules which will be implemented by the real–time image processing kernel to realize the parallel real–time data flow. Referring to figure 14, the products of the interpretation are (1) PCT network configuration files, and (2) a GUI configuration file. These files are used in (1) booting the network, (2) configuring the network communication engines and schedulers, and (3) configuring the dynamic

ALGORITHM NAME	ALGORITHM DESCRIPTION
absdiff	Absolute value of the difference between frames
conv	Convolution with 3x3, 5x5, 7x7, or 9x9 kernel
disp	Display frames (runs on the NEL Display TIM)
ds	Down–sample (in time or in the image plane)
grab	Grab frames (runs on the NEL Grabber TIM)
lap3x3	Convolution with a 3x3 laplacian kernel
lut	Table lookup
makefake	Either generates test patterns or buffers frames and repeatedly sends them in an infinite sequence (for test purposes)
plothist	Plot histogram (runs on the NEL Grabber TIM)
screech	Specialized algorithm for detecting short–lived "screech" phenomena evident in exit nozzle turbine engines during altitude simulation tests.
tavg	Time average (average last N frames)
track	Track an object (runs on the NEL Display TIM)

Table 10: Image Processing Algorithms Implemented

parameter graphical user interface.

Relationship Between Performance Models and Allocation

Performance models are needed for determining (1) if a particular computation can meet the specified performance goals using the available hardware, (2) a decomposition method and granularity of parallelism (scale) for each block, and (3) a mapping of the decomposed computations to the hardware that will meet the constraints.

In general, performance models are dependent upon the properties of the particular computations, the parallelization technique, and the allocation to the hardware network. This forces the processes of decomposing the data flow and allocating it to the hardware to somehow occur simultaneously. It is preferable to decouple these

processes to make the mapping more practical to automate.

Due to the properties of the PCT communication technique, the support provided by the PCT run–time system, and the restriction of the top–level PCT Partition structure to a pipeline, the allocation scheme and hardware architecture can be simplified enough that the throughput and latency models can be built in a separate step before allocation. This effectively decouples the decomposition and allocation processes, making the automation of mapping data flows to hardware tractable. The emphasis in developing performance models can thus be placed on the properties of the computations. In a later section, I show that building performance models requires the following a priori knowledge of each algorithm in the data flow: (1) the relationship between each algorithm's execution time and data size, and (2) the way that each algorithm accesses the input data structures in computing each output data element.

The Interpretation Procedure

Because the decomposition and allocation processes have been effectively decoupled by the assumption that the PCT run–time system will be used, the search for an appropriate mapping between the image processing data flow and the hardware pipeline can be reduced to finding an appropriate data flow decomposition. Decomposing the data flow is a two level process: (1) find a PCT Partition, and (2) find a *decomposition alternative* (a supported type of parallel decomposition) and scaling factor for each PCT Block. Judging the success of a particular decomposition involves (1) building throughput and latency models and comparing them to the throughput and latency goals specified in the system's RealTimeConstraints model (2) making sure that the hardware architecture can support the decomposition (it

must have enough of the right kind of processors, and they must be connected in a an appropriate topology).

The procedure followed by the interpreter is to first partition the SDF into a PCT compliant partition, determine the valid decomposition alternatives for each block, and then search for a combination of valid block decomposition alternatives and scaling factors that will meet the performance constraints and that the hardware architecture can support.

```
build a PCT Partition (P) of the application data flow
//initialize partition
for every PCT block (B) in P {
   build a PASS for B
   construct block data dependencies
   initialize alternatives
   for every decomposition alternative DA {
     build throughput & latency models
     scale throughput {
       if goal throughput was not reached
         use the max throughput
       }
  }
//search for decomposition alternative set
for every combination of block decomposition alternatives (C) {
   //load balance
   for every PCT block (B) in P {
     scale throughput(B) to throughput(P)
   }
   if latency(c) > goal latency {
     if latency constraint is not hard {
       store C in possibilities set
     }
     continue
   }
   if throughput(c) < throughput goal {
     if throughput constraint is not hard {
       store C in possibilities set
     }
     continue
   }
//constraints are met with P & C
   allocate P to network
   write configuration files
}
// no solution was found → choose from possibilities set
for every C in possibilities set {
   if constraints can be relaxed to C {
         allocate P to network
         write configuration files
   }
// no solution exists
FAIL
```

Figure 25: Interpretation Procedure

A pseudo code version of the interpretation procedure is shown in figure 25. Note that the decomposition procedure contains two layers, (1) partitioning of the data flow into PCT–compatible blocks, and (2) data parallel decomposition of the blocks. The allocation occurs only after a scaled decomposition has been chosen for the solution. This decoupling of the decomposition and allocation was made possible by partitioning the top level data flow into a pipeline of PCT Blocks, which can be implemented on a hardware pipeline using the PCT run–time facilities. Without this simplification, the performance models would be inextricably dependent upon the allocation, and thus a much more complicated procedure would be required.

An exhaustive search is made of all decomposition alternative combinations until either the constraints have been met, or the valid alternative sets have been exhausted. Alternative combinations which meet the hard real–time constraint(s), but may not meet the other(s), are stored in the *possibilities set* during the search. The end result of a successful search is a partition, and a set of decomposition alternatives and scale factors for the partition blocks which can be allocated to the hardware pipeline to achieve the target throughput and latency. If no solution was found during the search, an attempt is made to relax the throughput and/or latency constraints. If both constraints are *hard* constraints, then no concessions are made. Otherwise, the alternative sets which were stored in the *possibilities set* are examined, and the one which most nearly matches the constraints is chosen. The decision of which of these most nearly matches the constraints is made by putting priority on throughput by choosing the set which produces the highest frame rate. This decision was made primarily because the system was designed with real–time video in mind, and in video applications throughput is most often the more important constraint.

In the future, this approach favoring throughput should be replaced with a more flexible one to reflect the needs of other domains. However, note that this problem only occurs if no mapping could be found which would meet the constraints.

The initial version of the algorithm is also flawed in that, since the latency (in frames) depends strongly upon the number of blocks and the decomposition alternative set, the graph should immediately be repartitioned if the latency apparently cannot be met with the current partition and the goal is latency is a hard constraint. This would avoid unnecessarily long searches. This more intelligent approach will be implemented in any future work on this project.

Partitioning The Data Flow Graph

The image processing synchronous data flow graph must be partitioned into a PCT Partition to meet the requirement of the PCT run–time system. For a PCT Partition, each computation block must meet the following set of rules:

- only image sequences can be inputs and outputs to the block
- at most one block input
- at most one block output
- no other block can share this block's input connection

A simple algorithm is used to find the *first* PCT Partition of the synchronous data flow, which is the PCT Partition with the most blocks. A pseudo code version of the algorithm is shown in figure 26.

```
//build initial partition

build a partition (P) with exactly one algorithm per block

//combine blocks until each is a PCT Block

while P is not a valid PCT Partition {
   for each block (B) in P {
      if B is source or sink
         continue
      if B has ( > 1 input) or ( non–image inputs) {
         combine B with block producing of each input
         continue
      }
      if B has ( > 1 output) or ( non–image outputs) {
         combine B with each block consuming each input
         continue
      }
   }
}
```

Figure 26: PCT Partitioning Algorithm

Generating Block Schedules

Of the several available static scheduling algorithms for synchronous data flows, I chose the Lee–Messerschmidt algorithm for synchronous data flows to generate a static schedule for each partition block. The algorithm, developed in [32], is based on a linear algebraic formulation of the internal buffer levels for the local synchronous data flow computation. The existence of a PASS is determined by examining the null space of the next buffer state transformation matrix, and the schedule is determined through a simple procedure which repetitively updates the buffer state vector by application of an algorithm selection vector to the transformation matrix. The implementation of the algorithm in the interpreter required a matrix manipulation package, which I wrote in C++. The details of the Lee–Messerschmidt algorithm can be found in [32].

Analyzing the Decomposition Alternatives

Each partition block can be assigned one of six supported decomposition alternatives, each consisting of a data decomposition method, and whether or not the I/O and computation processes will be simultaneous. The supported decomposition alternatives are shown in table 11. For each alternative, the block's net I/O data dependency is determined, and local input, output, and intermediate data sizes are computed. These are used in building performance models for the block.

DA#	Decomposition Method	Simultaneous I/O and Computation
1	sequential	no
2	sequential	yes
3	temporal	no
4	temporal	yes
5	spatial (rows)	no
6	spatial (rows)	yes

Table 11: Supported Decomposition Alternatives

Building Performance Models

For each supported decomposition alternative, models for throughput and latency of a PCT Block have been developed, and are discussed in detail in this section. First, the overall requirements and assumptions of the performance modeling technique are given, and then the PCT Block and PCT Partition performance models for each decomposition alternative are developed.

Requirements and Assumptions

Execution Time Benchmarks:

The approach assumes that for each algorithm, the dependence of that algorithm's execution time on data size has been determined via empirical benchmarks which have been declared in the function's model. The performance of an algorithm for a particular data size is estimated by linear interpolation on these benchmarks, so each algorithm must be benchmarked with a sufficient set of data sizes such that the relationship inferred by linear interpolation on the benchmark set is an acceptable estimate of the execution time for a particular data size. Algorithms with non–linear execution time/data size relationships require a larger set of benchmarks to be specified than algorithms with more linear execution time.

Data Dependency Specification:

It is also necessary to assume that the data access patterns for the algorithms are known. Some sort of *data dependency* specification will convey the relationship between elements of the output data sequences and required regions in the input sequences. From these specifications, *total data dependency set* will be constructed for each element of the output data sequence, resulting in direct knowledge of what regions of the input sequences are required to be local to the computation of each output data element.

Specification of Worst Case Behavior:

Although the execution time and data access patterns of the algorithm must be known a priori to the performance modeling, neither must be fixed. Both can vary

with changes in the nature of the data, as long as worst case specification of the execution time and required regions are available prior to run–time.

For algorithms with data dependent performance, the benchmarks given are assumed to be upper bounds, so the products of the performance analysis are worst case throughput and latency models. However, note that this practice may lead to very inefficient solutions in the cases in which the actual time taken by the algorithm is far less than the upper bound.

For algorithms with data dependent data access patterns, the worst case behavior is specified as a bounding region for each of the input images, where a bounding region is a set of pixel locations which is guaranteed to be sufficient to calculate all local output data for all cases. Again, note that in cases where the actual data required locally to each processor is much smaller than the worst case bounding region, the implementation may be inefficient due to unnecessary communication overheads.

In the next sections, the latency and throughput models for each of the decomposition methods will be derived, using the assumptions above and also supposing that the PCT Block data flow will be implemented by the PCT–C40 run–time system.

General Execution Time Model

Given a single node from a PCT Group of processors implementing a PCT Block data flow using any of the decomposition methods, the execution time required for that node to compute it's local partial result can be characterized by general execution time models, depending upon whether the I/O and computations are performed simultaneously.

Without SIMIOAC:

In the case in which the I/O and computation are not simultaneous, the local input must be received, the local output sent, and the local inputs shared with the downstream processors before the scheduler is allowed to proceed). For this reason, the I/O and sharing times for this case must be added to the actual computation time. The resulting model is shown below.

$$T_{exec} = T_{I/O} + T_{share} + T_{schedule}$$

With SIMIOAC:

In the case in which the I/O and computation are simultaneous, the local input is received, the local output is sent, and the local inputs are shared with the downstream processors as the computation is being performed on the previous input buffer.

With this synchronization method, the time taken to complete one computation cycle will depend upon which is slower, the computation, or the communication. The resulting execution time model is shown below.

$$T_{exec} = MAX(T_{I/O} + T_{share}, T_{schedule})$$

Calculation of $T_{I/O}$, T_{share}, and $T_{schedule}$:

The values of $T_{I/O}$, T_{share}, and $T_{schedule}$ in the general models obviously depend upon the block input, output, and intermediate local data sizes, the amount of the local data which is to be replicated and shared with the downstream peers, and the time taken to compute the local output. These quantities will vary depending upon the decomposition method and the scaling factor (number of nodes).

The I/O time, $T_{I/O}$, scales linearly with the local input and output data sizes, and is modeled simply by

$$T_{I/O} = \frac{MAX(size_{input}, size_{output})}{Bandwidth_{link}}$$

Similarly, the input sharing overhead is

$$T_{share} = \frac{size_{shared}}{Bandwidth_{link}}$$

where $size_{shared}$ is determined by analysis of the data dependency specification. The shared size is the number of pixels lying in the intersection between the region in the input sequence the node requires locally and the regions in the input sequence the node's downstream peers require (the total number of pixels which are local to this node which must be replicated and sent to downstream peers). This quantity depends upon the decomposition method, and will be characterized for each case in the following discussion.

The block computation time, $T_{schedule}$, is achieved by summing the execution times of the individual algorithms. Each algorithm's execution time is calculated by linear interpolation on it's benchmarks using the data size seen on that algorithm's output connection. The data sizes of the image sequence connections are obtained by first determining the size of the local partial result image based on the total image size, the decomposition method, and the number of nodes in the PCT group. Then the data dependency specifications of the algorithms are used to propagate back from the block output through the block data flow to determine the required regions of each input and intermediate results sequence.

For example, consider the PCT block introduced previously containing two algorithms in series, the second being a $5x5$ convolution. The schedule execution time

for each group node will be derived from the size of the partial results computed on each node, and the size of the intermediate results which are necessary to compute the partial results. Since the second computation is a $5x5$ convolution, the first algorithm must compute two extra rows at the top and the bottom of it's output data so that the convolution will have all necessary input data. The result is that the first algorithm's output data size is larger that the output algorithm's. Considering two group nodes, the extra rows of the intermediate result which must be computed on each are shared between the two, and this sharing introduces computation overhead.

The procedure for calculating $T_{schedule}$ is to first determine the local required region of each input image and intermediate results sequence based on the local partial results size and the data dependency specifications. The execution time of each algorithm is then calculated by linear interpolation on the algorithm's execution time benchmarks using the size of the local required region of it's output image connection. The individual algorithm execution times are then summed to arrive at $T_{schedule}$.

$$T_{schedule}(size_{local\ output}) = \sum_{i=0}^{M} time_{algorithm_i}(size_{connection_i})$$

where M is the number of local algorithms, $size_{local\ output}$ is the size of the local partial result, and $size_{connection_i}$ is the size of required region of $algorithm_i$'s output image. Note that if spatial decomposition is used, the block output data size will be

$$size_{local\ output} = \frac{size_{output}}{N}$$

and if temporal decomposition is used, it will be

$$size_{local\ output} = size_{output}$$

Relating Execution Time, Latency, and Throughput:

How the execution cycle time of the individual nodes relates to the throughput and latency of the PCT Group varies with the decomposition alternative being implemented. If the data is being decomposed spatially, the I/O and execution times for each group node decreases, which increases throughput and decreases latency (latency in seconds, but not latency in frames). In the case of temporal decomposition, the I/O and execution times for each group node do not decrease, but several images will be computed simultaneously, so the number of output per unit time (throughput) increases. However, because several images will exist in the pipeline all the time, the latency increases,

In the following sections, throughput and latency models specifically for each decomposition alternative will be developed by examining the timing characteristics of PCT–C40 implemented data flows. Note that the ordering of the discussion does not go linearly through the decomposition alternatives shown in table 11, but instead begins with the spatial decomposition alternatives.

Performance Models for Spatial Decomposition

DA #5 – Spatial Decomposition Without SIMIOAC:

Figure 27 shows the timing of a spatially decomposed PCT Block without simultaneous I/O and computation. Notice that the splitting of $input(k)$ and the merging of $output(k-1)$ are started simultaneously, and that the scheduler is not started until the I/O processes finish.

The schedule execution time can be tied directly to the size of the local partial

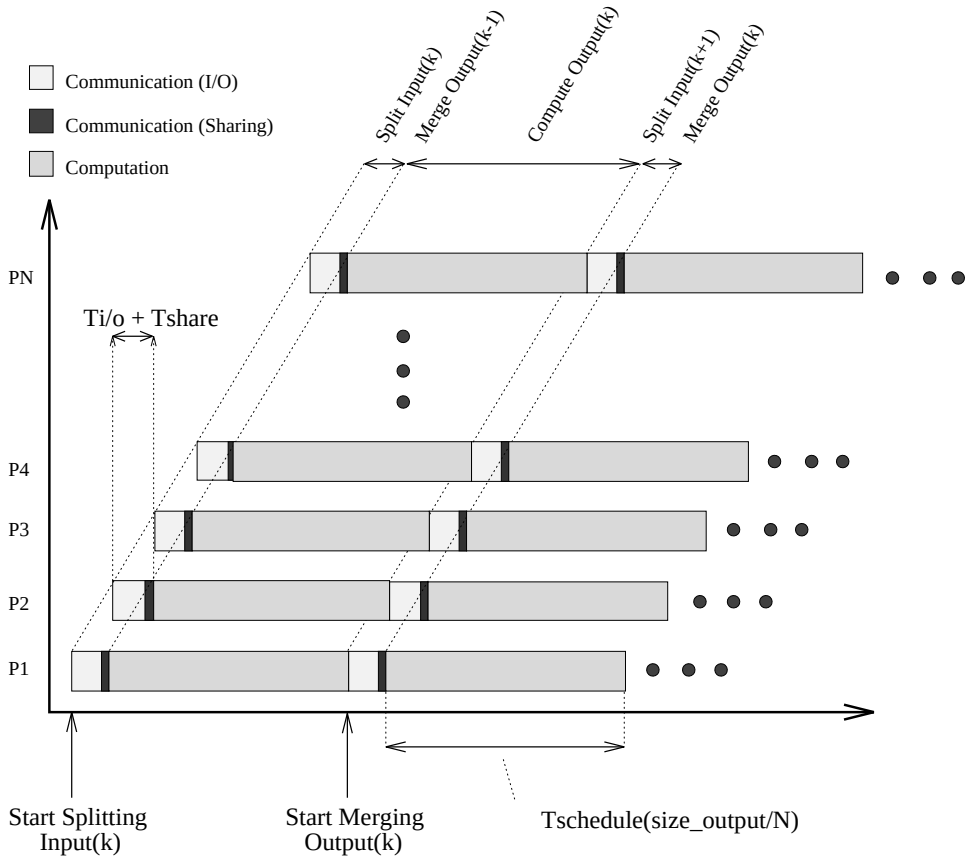

Figure 27: Split–and–Merge Timing (Splitting and Computation Not Simultaneous)

result, which is the block output image. In the case of spatial decomposition, the block output size is $\frac{size_{output}}{N}$, so the schedule execution time is

$$T_{schedule}(\frac{size_{output}}{N}) = \sum_{i=0}^{M} time_{algorithm_i}(local\ size_i)$$

To form the throughput model, notice that N image pieces enter, and N partial results (one image) exit the system every cycle time, which is given by

$$cycle\ time(N) = T_{I/O} + T_{share} + T_{schedule}(\frac{size_{output}}{N})$$

The throughput in $\frac{frames}{sec}$, then, is merely the inverse of the cycle time.

$$throughput = \frac{1}{cycle\ time(N)}\ \frac{frames}{sec}$$

Thus, the system throughput is given by

$$Throughput(N) = \frac{1}{T_{I/O} + T_{share} + T_{schedule}(\frac{size_{output}}{N})}$$

$$Throughput(N) = \frac{1}{T_{I/O} + T_{share} + \sum_{i=0}^{M} time_{algorithm_i}(size_{connection_i})}$$

To determine the latency model, refer to figure 27, and note that $output(k)$ starts exiting the block exactly one *cycle time* after $input(k)$ begins to enter the system. Since $output(k)$ is the result of running the schedule on $input(k)$, the latency is

$$latency(N) = 1 cycle\ time = 1\ frame$$

Note that the latency in seconds will decrease as the cycle time decreases, but the latency in frames will remain constant at 1 $frame$.

DA #6 – Spatial Decomposition With SIMIOAC:

Figure 28 shows the timing of a spatially decomposed PCT Block with simultaneous I/O and computation.

PCT–C40 does not allow a particular input image to be received and computed simultaneously. When simultaneous I/O and computation is used, the input and output images are double buffered. When $input(k)$ is being split (received locally), $output(k-1)$ is being computed from $input(k-1)$, and $output(k-2)$ is being merged (see figure 28).

During any processor's schedule cycle, which lasts $MAX(T_{I/O}+T_{share}, T_{schedule}(\frac{size_{output}}{N}))$ time units, exactly N pieces, or an entire image, enters and exits the system. Thus, The throughput is given by

$$Throughput(N) = \frac{1}{MAX\left(T_{I/O} + T_{share}, T_{schedule}(\frac{size_{output}}{N})\right)}$$

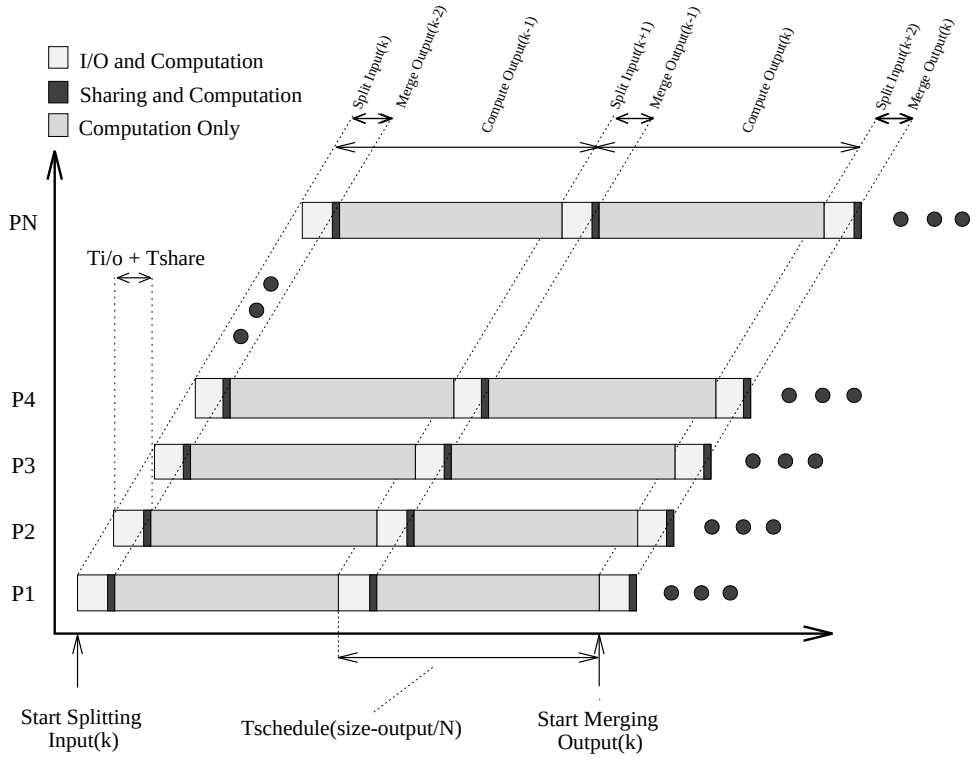

Figure 28: Split–and–Merge Timing (With Simultaneous Splitting and Computation)

$$Throughput(N) = \frac{1}{MAX\left(T_{I/O} + T_{share}, \sum_{i=0}^{M} time_{algorithm_i}(local\ size_i)\right)}$$

To determine the latency model, refer to figure 28, and note that the $output(k)$ begins to exit the block exactly two cycles after $input(k)$ begins to enter the block. Thus, the latency is given by

$$Latency(N) = 2 cycle\ times = 2\ frames$$

Discussion:

Note that in either of the models shown, the achievable throughput will of course have an upper bound. It will either be limited by the hardware communication bandwidth, or by the nature of the problem.

In the first case, the amount of communication occurring on one of the communication links reaches it's maximum data rate, and the throughput abruptly reaches a maximum. At that point, additional processors will begin to decrease the throughput.

In the second case, the schedule time will decrease to the point that it is balanced with the local I/O time (they are equal). Referring to figures 27 and 28, note that if the problem is not *computation bound* (the $T_{schedule}(size_{output})$ is not very large with respect to $T_{I/O} + T_{share}$), it is probable that after N increases even slightly, the throughput will reach a maximum. This case is certainly not disastrous. It is actually the preferable case, since this means that only a few nodes will be required to scale the problem to the I/O bandwidth, which is usually the target throughput.

When using either DM#5 or DM#6 (see table 11), the maximum throughput is reduced to below the hardware link bandwidth as the amount of input data required to be shared between the group nodes becomes large. In either case, the following holds true:

$$Throughput(N) * image\ size \leq bandwidth_{link} - bandwidth_{share}$$

In other words, the maximum achievable communication data rate will be bound by the hardware bandwidth reduced by the bandwidth required to share input data using the links internal to the PCT Group.

Although the throughput is scalable to the same level with either decomposition alternative, there are advantages to using one over the other in certain situations. The obvious advantage of DA #6 is the factor of two latency reduction. However, examining figure 27, note that in the case with simultaneous I/O and computation, at any given time N processors are computing. In the other case, shown in figure 28, only

$N-1$ are computing at any one time. By allowing all processors to compute all the time, higher throughput is achieved with fewer processors, but the double buffering that is required increases the latency. The trade–off is evident. When latency is important, then the simultaneous I/O and compute option should not be used, but when latency is not important, it should.

Performance Models for Temporal Decomposition

The images are not split into pieces with temporal decomposition, so the input/output communication times are easily characterizable. However, there are still two cases to be considered: with and without simultaneous communication and computation. These cases are represented by the timing diagrams in figures 29 and 30. Notice that the timing diagram shows no time for sharing images between the group processors. The PCT–C40 run–time system does not support sharing entire images between the group peers because the resulting overheads would be too costly. However, future versions of the system using higher bandwidth connections may provide such support. $T_{I/O}$ can be characterized simply by

$$T_{I/O} = \frac{MAX(size_{input}, size_{output})}{bandwidth_{link}}$$

DA #3 – Temporal Decomposition Without SIMIOAC:

Figure 29 shows the timing of a temporally decomposed block when the nodes do not compute while receiving data. Notice that $input(k)$ enters the system while $output(k-1)$ exits, and that the computation of $output(k)$ from $input(k)$ begins after $input(k)$ has been received.

The throughput model is formed by noting that N entire images enter/leave the

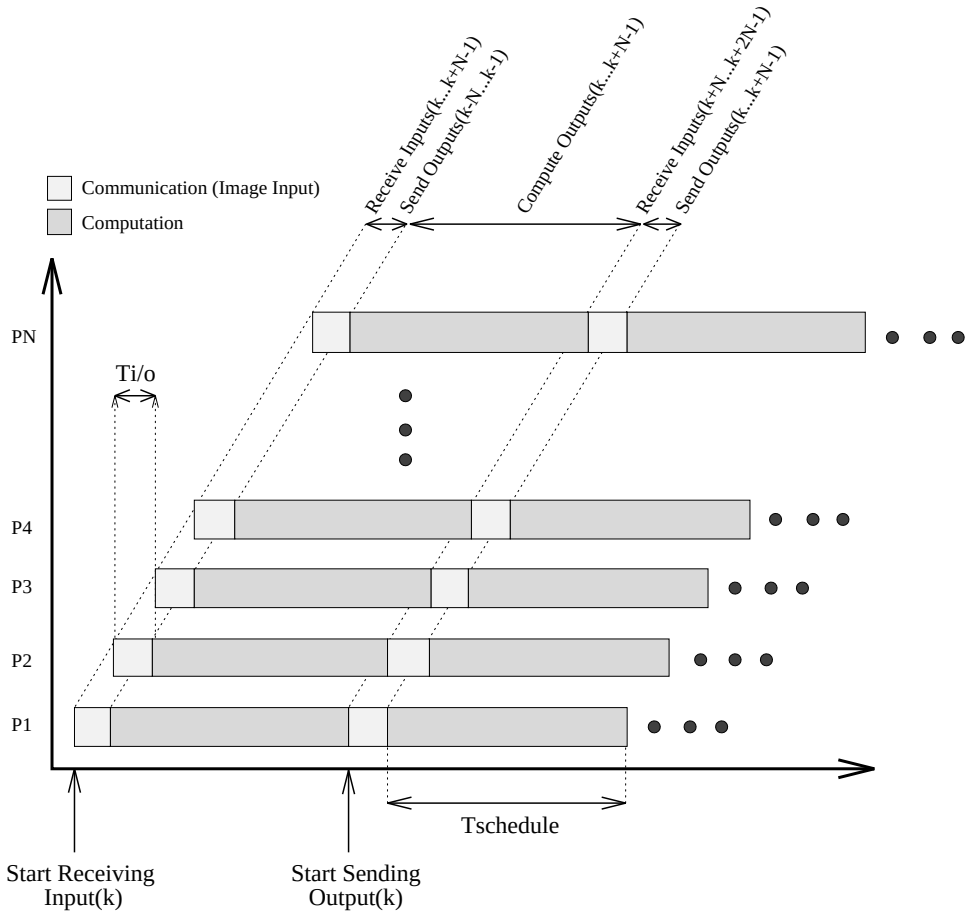

Figure 29: Temporal Decomposition Timing (Communication and Computation Not Simultaneous)

system every $T_{schedule} + T_{I/O}$ time units. The throughput is given by

$$Throughput(N) = \frac{N}{T_{schedule} + T_{I/O}}$$

The latency model in frames is obtained by noting that $output(k - N)$ begins leaving the block exactly when $input(k)$ begins entering the block.

$$Latency(N) = N\ frames$$

The latency in seconds is obviously

$$Latency(N) = T_{I/O} + T_{schedule}$$

Note that the latency in frames increases with N, but the latency in time remains constant.

DM #4 – Temporal Decomposition With SIMIOAC:

Figure 30 shows the timing of a temporally decomposed block when the nodes do compute while receiving data.

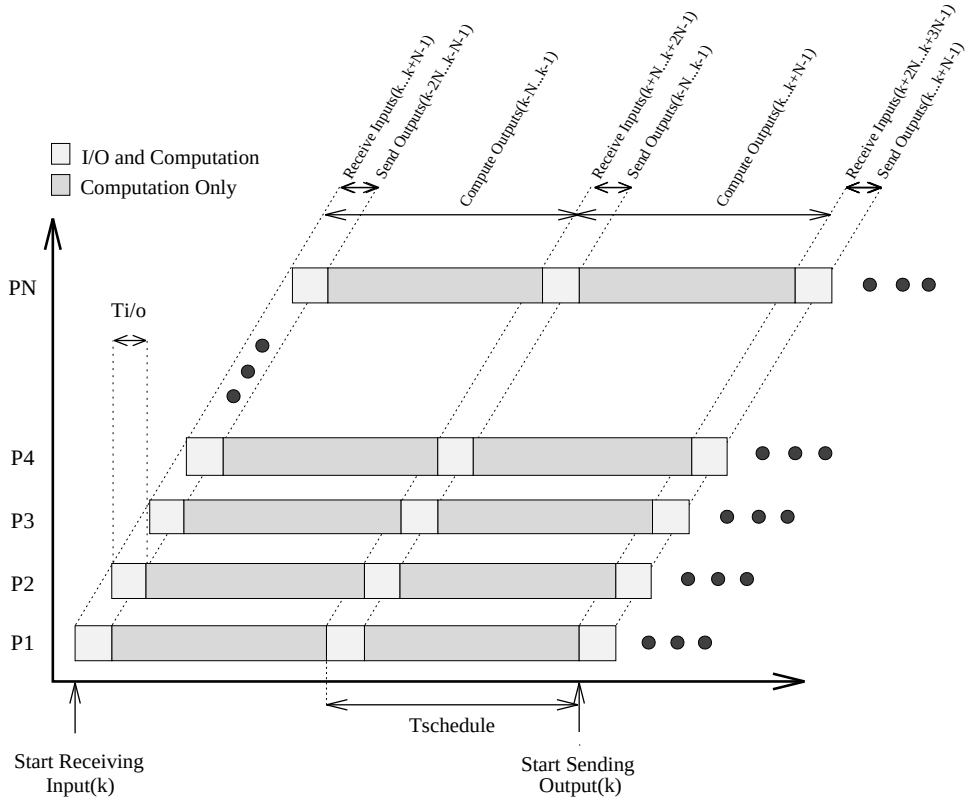

Figure 30: Temporal Decomposition Timing (With Simultaneous Splitting and Computation)

The throughput model is given by (see figure 30)

$$Throughput(N) = \frac{N}{T_{schedule}}$$

Noting that $output(k - 2 * N)$ exits the block at the same time that $input(k)$ enters

it, the latency is given by

$$Latency(N) = 2 * N \ frames$$

The Latency in seconds is

$$Latency(N) = 2 * T_{schedule}$$

Discussion:

An obvious drawback of temporal decomposition is that the computational latency increases with the number of processors. However, the throughput scales with N, and the problem of throughput being limited because of the nodes sharing input pixels is removed. Temporal decomposition is useful for non latency critical applications, algorithms which have large required regions, or when the algorithms are non–splittable.

Performance Models for Sequential Execution

These cases, DM#1 and DM#2, are not of interest, since the throughput and latency models are the same as either the spatial or the temporal models with $N = 1$.

The Dynamic Parameter Graphical User Interface

A GUI for adjusting the dynamic parameters of the algorithms in the data flow has been specified, but only partially implemented. The GUI is being implemented as a set of reconfigurable Java applets which are used in conjunction with HTML pages and the Netscape web–browser. In lieu of the GUI implementation, a textual interface is currently used for interacting with the running network. It runs on the

PC host, and allows the user to dynamically configure the running algorithms by broadcasting messages across the C40 network via the PCT communication system. The messages are tokenized and passed to the appropriate image processing algorithm setup functions, which interpret the message and update the parameter values of the algorithms as they run.

The interpreter generates a GUI configuration file from (1) the information contained in the parameter interface aspect of the individual image processing function models, and (2) the data flow decomposition. This configuration file will be used by the GUI in determining what parameters can be adjusted, the types of widgets to be used, the initial values, the function to which each parameter belongs, and how the messages are to be constructed.

As an example, consider the $5x5$ convolution algorithm model seen in figure 37 of the appendix.

This model defines parameters for dynamically adjusting the convolution kernel, the scale, and the offset. The convolution function model then contains descriptions of the kernel, scale, and offset parameters which are put into the configuration file to (1) inform the user interface to include widgets for adjusting the kernel (an array of floats), the scale, and the offset (floats), and (2) describe to the user interface what messages to send and how to send them in order to update those parameters when the widgets are adjusted.

CHAPTER VI

CONCLUSIONS

This dissertation has presented the problem posed by the need for support of the high performance image processing and vision applications which will become prevalent in the future. Such vision applications may require on the order of tens of billions of operations per second.

One approach which shows promise in supporting the applications is to somehow decompose the image processing computations and map them to parallel architecture. However, parallel architectures are prohibitively difficult to program. For the parallel approach be become feasible, high–level parallel programming development environments and tools for generating the parallel applications are essential.

This work has taken a novel approach toward parallel programming by generating parallel real–time applications of image processing data flows from high–level specifications. Both (1) an environment with which the user can build graphical models of a data flow computation, the hardware resources available to solve the problem, and real–time specifications, and (2) an interpreter for automatically transforming these models into a real–time implementation have been developed.

The interpreter performs the data flow decomposition, performance modeling, scaling, load balancing, and scheduling simultaneously and automatically, then allocates the decomposed, scaled computation to a network of DSPs. A parallel image processing run–time kernel provides communication, routing, scheduling, and synchronization for the implementation.

Contributions

Parallel Program Generation

A unique approach toward parallel program generation was developed which combines automatic translation of sequential programs and meta–level driven software synthesis. Existing sequential programs are automatically parallelized with the aid of information about the algorithms stored in graphical models.

Explicitly Specified Real–Time Constraints

Both throughput and latency constraints are modeled explicitly and drive decisions about the types of decomposition and granularity of the parallelism to be used.

Performance Modeling

Predictive performance models have been developed using a multiple benchmarking technique which enables the parameterization of algorithms which have a non–linear relationship between execution time and data size. The performance models also take into account the communication overheads due to sharing of data between processors.

Data Dependency Specification Language

A mathematical specification language for modeling the data dependency patterns of image processing algorithms has been developed, along with a method of combining these data dependency specifications to determine the dependency between any two image signals in a data flow. This capability was necessary to the implementation of

the multi–level decomposition technique for complex data flows.

PCT–C40 Run–Time Kernel

A run–time kernel has been implemented for C40s which uses a unique communication routing technique to allow data parallel implementations of complex image processing data flows to be mapped to a C40 pipeline. The communication engines are programmed to operate as state machines which route all communication along the pipeline, cutting through intermediate nodes without affecting their on–going computations.

Automatic Data Flow Decomposition and Mapping

A method of automatically decomposing image processing data flows and mapping them to parallel hardware resources has been devised. The decomposition and scaling decisions are driven by the real–time constraints specifications and success of a particular scheme is gauged with predictive performance models.

Future Work

Some aspects of this model–based approach toward generating real–time parallel image processing systems are beyond the scope of this dissertation and should be investigated further in the future.

Support of Applications with Multiple I/O Streams. The MIRTIS system supports only data flows with single source and sink image streams. Applications such as robotics routinely use multiple cameras for stereo vision, and military target tracking systems often use multiple sensors types (visible light and infrared) simultaneously

in tracking objects. These applications should be supported.

Support of Other Decomposition Methods. Currently, the run–time system supports temporal and spatial decomposition by blocks of rows. Other methods which should be supported are spatial decomposition by blocks of columns and windows.

More Efficient Mapping Algorithm. The mapping algorithm used by the MIR-TIS model interpreter performs an exhaustive search of all combinations of block decomposition alternatives. For N blocks, this generates 6^N possible combinations. A much more intelligent approach is possible by noting that the latency is dependent only upon the data flow partitioning and the decomposition alternative of each block. If the latency constraint were guaranteed first, during the partitioning stage, the search space would be greatly reduced. The throughput constraint could then be satisfied quickly by treating each of the blocks separately during the scaling stage. The need for an exhaustive search of all decomposition alternatives would then be eliminated. Since the blocks can be treated separately, and each has at most six possible alternatives, the previous search through 6^N combinations could be reduced to at most $6 * N$ combinations.

Appendix A

GLOSSARY OF ACRONYMS AND TERMS

- *C40:* The Texas Instruments TMS320C40 DSP. A DSP specially designed to be used as a parallel processing building block. Each C40 has a 40 MFlop floating point processor and 6 communication ports capable of transferring $20\frac{Mbyte}{sec}$. C40s can be bought in COTS packages called TIMs (TI Modules) which usually contain local memory banks, and are plugged onto various types of mother boards. The inter–processor connections are made with ribbon–cables. Virtually any topology can be constructed from C40s.

- *COTS:* Commercial Off–The Shelf. A standard part purchased from a commercial vendor from stock is said to be COTS.

- *DMA:* Direct Memory Access Co–Processor. An autonomous co–processor which can be programmed to perform memory transfers independent of the CPU. (DMA transfers can occur simultaneously to CPU execution.) The C40 DMAs can transfer data from memory to memory, to/from communication ports from/to memory, or directly from communication port to communication port.

- *DSP:* Digital Signal Processor: A class of micro–processors specialized for signal processing computations.

- *HPF:* High Performance Fortran. A version of Fortran 90 which is augmented with parallel directives with which the programmer gives hints to the complier

about how an array should be distributed across the memory elements of a multi–computer.

- *MGA:* Multigraph Architecture. A MIPS architecture developed at Vanderbilt University which provides a frame–work and tools for (1) building graphical domain specific models and (2) transforming the graphical models into executable applications [25]. Unlike other MIPS architectures, MGA use domain specific modeling paradigms, which (1) allows the domain engineers to specify a system in familiar terms, and (2) reduces the search space when trying to transform the models into a system.

- *MIMD:* Multiple Instruction–Multiple Data. A class of parallel computers defined by Flynn's classification of parallel architectures [18]. In an MIMD architecture, each processing unit has a separate control unit, and executes its unique instruction stream on its separate data stream independently of the other processors. See figure 11.

- *MIRTIS:* Model–Integrated Real–Time Image Processing System. A real–time image processing system which employs the MGA to automatically generate parallel real–time implementations of image processing data flows. The implementations run on a parallel C40 network under the control of the PCT parallel run–time system.

- *MIPS:* Model–Integrated Program Synthesis. A method of managing complexity in large scale engineering systems which involves synthesizing software based on models describing the target system.

- *Multigraph Architecture:* A MIPS architecture developed at Vanderbilt University which provides a frame–work and tools for (1) building graphical domain specific models and (2) transforming the graphical models into executable applications [25]. Unlike other MIPS architectures, MGA use domain specific modeling paradigms, which (1) allows the domain engineers to specify a system in familiar terms, and (2) reduces the search space when trying to transform the models into a system.

- *OCCAM:* A parallel language based on the Communicating Sequential Processes (CSP) formalism. OCCAM was designed primarily for programming the Inmos transputer, a parallel processing building block which was the predecessor to the modern generation of parallel DSPs, such as the C40 and the C44.

- *PASS:* Periodic Admissible Sequential Schedule. A type of schedule for synchronous data flows. See [32].

- *PCT:* Pipeline Cut–Through. A communication scheme which automatically implements the communication necessary for data parallel image processing algorithms on a C40 hardware pipeline.

- *PC++:* Parallel C++. A language specification which includes support for data parallelism at the data object level. Data parallel objects are derived from special template primitives which are explicitly distributed across a network. The implementation of the parallelism is facilitated by the primitive classes and is semi–transparent.

- *RTIP:* Real–Time Image Processing.

- *SDF:* Synchronous Data Flow. A data flow consisting of synchronous computations. A synchronous computation is one which, each time it is invoked, consumes a fixed number of data tokens on its inputs and produces a fixed number of data tokens on its outputs.

- *SIMD:* Single Instruction–Multiple Data. A class of parallel computers defined by Flynn's classification of parallel architectures [18]. In an SIMD architecture a single control unit provides an instruction stream to many processing units. Each processor executes the same instruction of different data synchronously. See figure 11.

- *TIM/TIM40:* TI Module. TIM40 is an industry standard C40 package. The TIM40 board contains a C40, memory interface circuitry, and RAM chips, and fits to a TIM40 motherboard via two connectors.

- *XVPE:* The graphical model editor available with the current generation of the MGA tool set. It is built on top of X Windows/Motif, and can store models in either the *Obst* or the *Objectivity* object–oriented databases (OODB). The next generation of MGA tools will also provide a native Windows editor and will be able to store models in any OODB which is compliant with the ODMG93 standard (a standard programming interface for OODBSs).

Appendix B

THE IPDL MODELING PARADIGM AND EXAMPLE MODELS

The MIRTIS system uses a modeling paradigm which has been designed specifically for real–time image processing. The paradigm is called Image Processing Description Language, or IPDL, and contains separate paradigms for describing the computations (*Signal Flow Models*), the hardware resources (*Hardware Models*), and the real–time constraints (*Constraints Models*). As each of these paradigms is discussed below, refer to the configuration file "ipdl.mdf" also included in the appendix.

Signal Flow Models

The signal flow paradigm contains models of the individual algorithms, and of applications. Applications are synchronous data flow graphs made up of algorithm models. The contents of the algorithm and application models are discussed below.

Algorithm Models

Each algorithm in the image processing library has an associated algorithm model. An algorithm model has several *attributes*, *parts*, and *conditional relationships* between its parts which are viewed from four *aspects* (see previous description in *MGA Models* section). The algorithm model aspects, and the information conveyed in each, are:

– Structure: the function's inputs and outputs

- Data Dependency : how the input and output data are accessed in constructing the output data structure
- Constraints : performance benchmarks and hardware requirements
- Parameter Interface : the dynamically adjustable parameters and adjustment modes

Structure Aspect

The algorithm model's structure aspect relays information about the algorithm's input and output data and the relevant image processing library function. Referring to the example shown in figure 31, it contains the following attributes and parts:

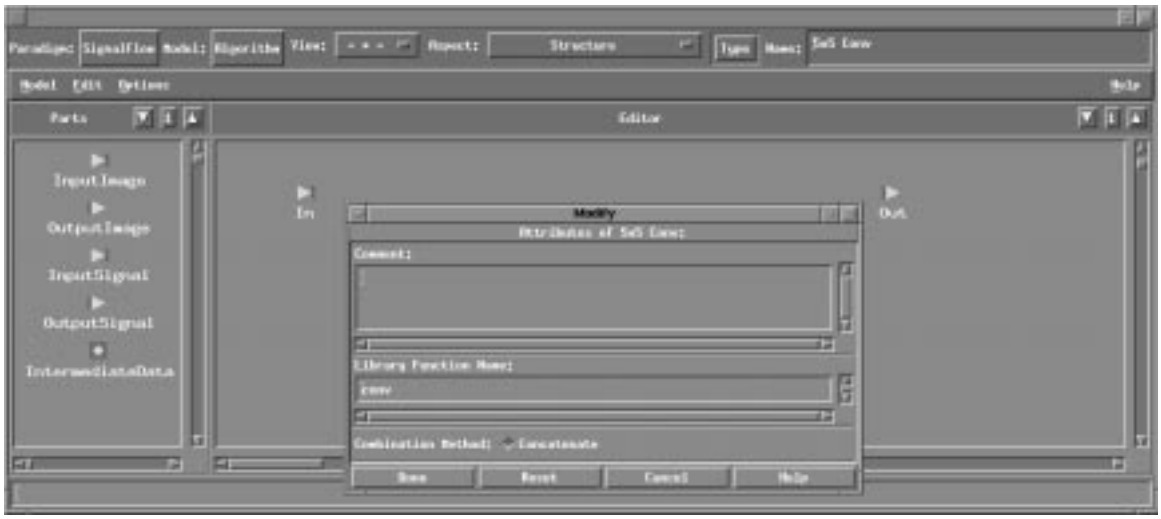

Figure 31: 5x5 Convolution Model Structure Aspect

- Attributes
  - Library function name: name of C function which implements the algorithm and the code library in which it resides.

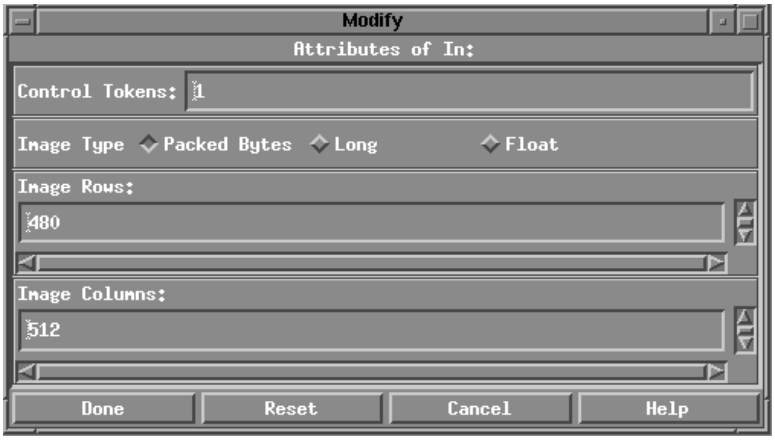

Figure 32: Attributes of 5x5 Convolution Image Signal "In"

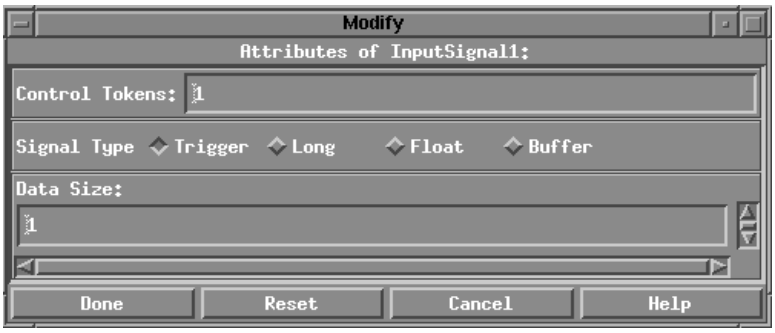

Figure 33: Attributes of a Non-Image Signal

* Partial results combination method: can be "concatenate" or nothing. Concatenate means that the partial results from a spatial decomposition of this function can be merely concatenated. If concatenate is not selected, the implication is that the function cannot be spatially decomposed (it is not splittable). No other combinations methods are supported by the system as of yet.

– Parts

* Input and Output Images: these parts specify image data inputs and outputs of the algorithm. The attributes of an example Input Image part are shown in figure 32.

* Input and Output Signals: these specify inputs and outputs of the algorithm which are not images. For instance, algorithms may produce and pass integers, floats, or non–image buffers. The attribute panel for an input signal is shown in figure 33.

Data Dependency Aspect

The algorithm model's data dependency aspect relays information about how the algorithm accesses input image data and past values of the output data structure in constructing each output data element. Referring to figure 34, it contains the following parts:

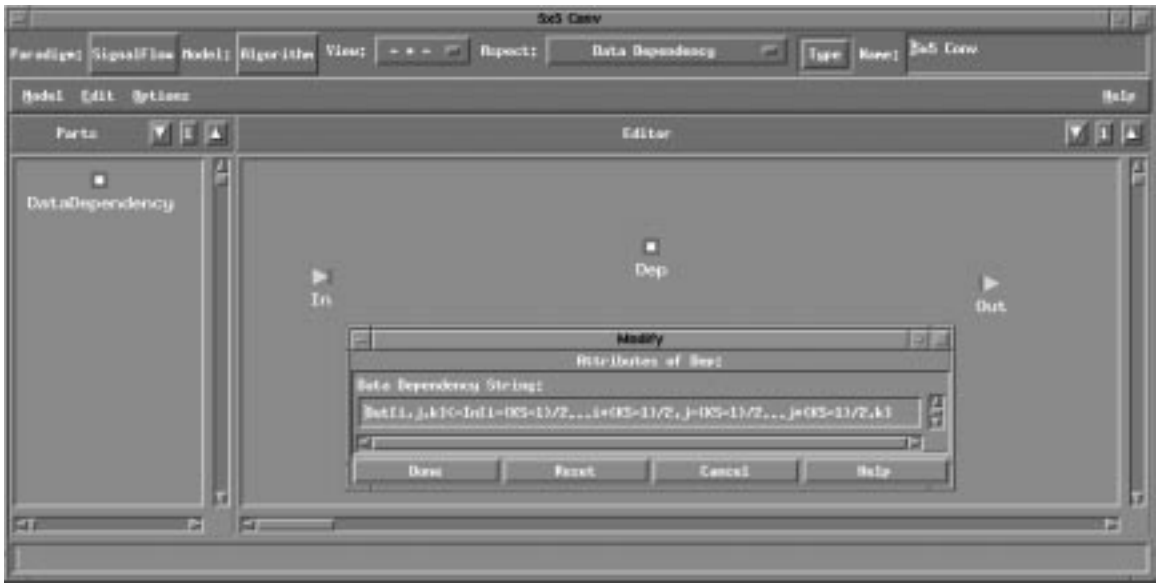

Figure 34: 5x5 Convolution Model Data Dependency Aspect

– Parts

* Input and Output Images: inherited from the structure aspect

* DataDependencies: these are parts with a single textual attribute which is a *data dependency specification*. This specification describes what region of an image signal is required in calculating a data element of another output or intermediate image signal. The syntax and meaning of the dependency specification is described in a later section. A model can contain as many data dependency parts as necessary to describe all direct dependencies between the image signals. A useful feature of the interpreter allows the formulas in the data dependency specification to refer to the global variables or the values of scalar parameters defined in the parameter interface aspect.

Constraints Aspect

The algorithm model's constraints aspect relays information about the algorithm's performance characteristics and its hardware resource requirements. Referring to figures 35and 36, the constraints aspect contains the following parts:

Figure 35: 5x5 Convolution Model Constraints Aspect

Figure 36: Grab Model Constraints Aspect

- References to Node and HostNode models from the Hardware paradigm (the Hardware paradigm will be explained later):

- References to Node and HostNode model resources from the Hardware paradigm:

- Requirements: these parts have no attributes, but can be associated with a reference to a node, a hostnode, a node resource, or a hostnode resource to indicate hardware resource requirements of the algorithm. During interpretation, these requirement specifications are used in determining whether a particular node satisfies the needs of this algorithm Requirements associated with nodes or hostnodes are combined with a logical "OR" and the requirements associated with node or hostnode resources are combined with a logical "AND". If a node type is either in the hardware reference list, or has all resources specified in the resource references, then it satisfies the requirements.

– Benchmarks: Each benchmark part describes the execution time of the algorithm for a particular data size, and is associated with a node reference which specifies on which type of node the benchmark was taken. By adding several benchmarks for various data sizes, all associated with the same node, a piece–wise linear version of the execution time versus data size curve can be constructed. For functions with near linear time/data size relationships, a single benchmark is adequate. However, for functions for which the time/data size relationship is non–linear may require several benchmarks. For instance, suppose a 2-D FFT (Fast Fourier Transform) algorithm was being modeled which was accelerated by zero padding the data into a square image with width the next highest power of 2. The execution time curve would be piece–wise constant, with discontinuities at widths of 256 , 512, and 1024. This curve would be modeled by providing 6 benchmarks, one for just smaller and one for just larger than each of the discontinuity points: benchmark(255x255), benchmark(256x256)=benchmark(511x511), benchmark(512x512)=benchmark(1023x1023), and benchmark(1024x1024). Linear interpolation on this set would then produce accurate estimates of the execution time for a particular data size.

Parameter Interface Aspect

The run–time system was designed in a way that as the real–time computation is being performed on the parallel hardware, special messages can be sent from a the host node which will make adjustments to so called "dynamic parameters" of the algorithms. Each of the library functions has an associated dynamic setup function which interprets the messages and adjusts that function's context. The result is that a

graphical user interface running on the host node can be configured to allow the user to adjust these dynamic parameters via graphical widgets, such as slider bars, pull–down menus, and select boxes. The GUI configuration must include enough information that the values of the adjustment widgets can be turned into the correctly formatted messages and sent to the executing system.

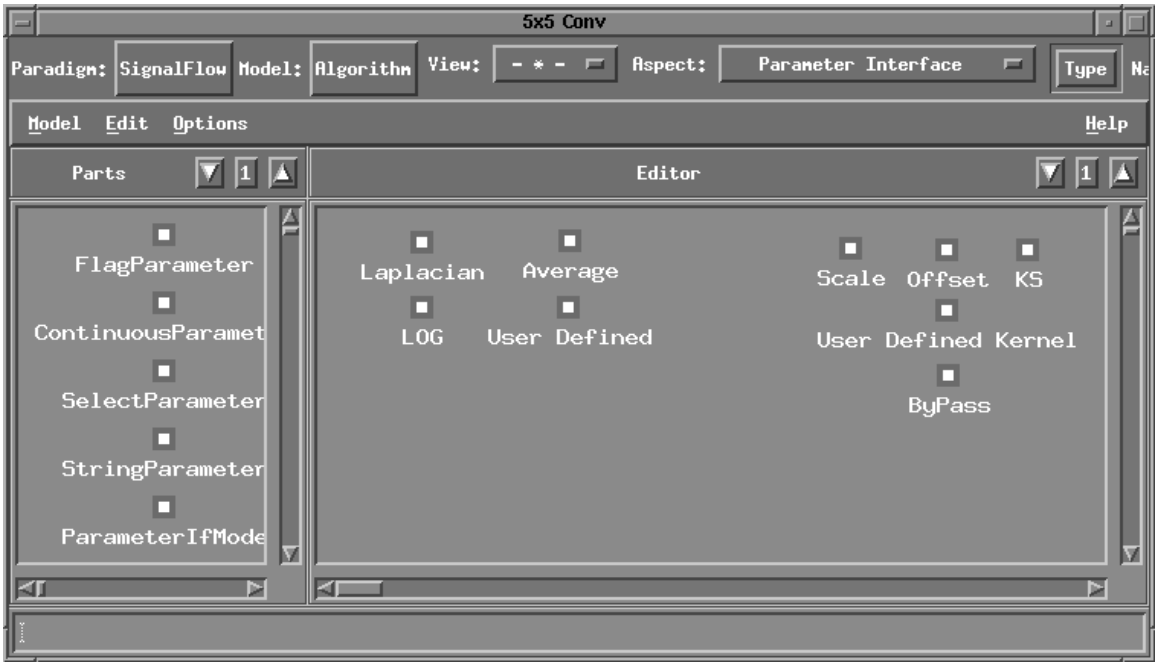

Figure 37: 5x5 Convolution Model Parameter Interface Aspect

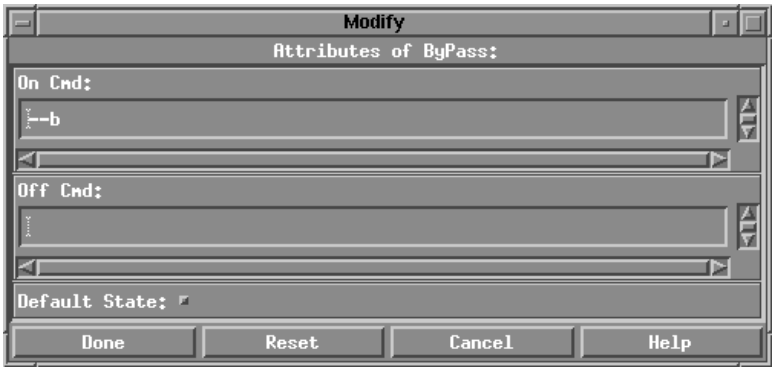

Figure 38: Attributes of the 5x5 Convolution FlagParameter "ByPass"

The parameter interface aspect of the algorithm model contains, for each algorithm

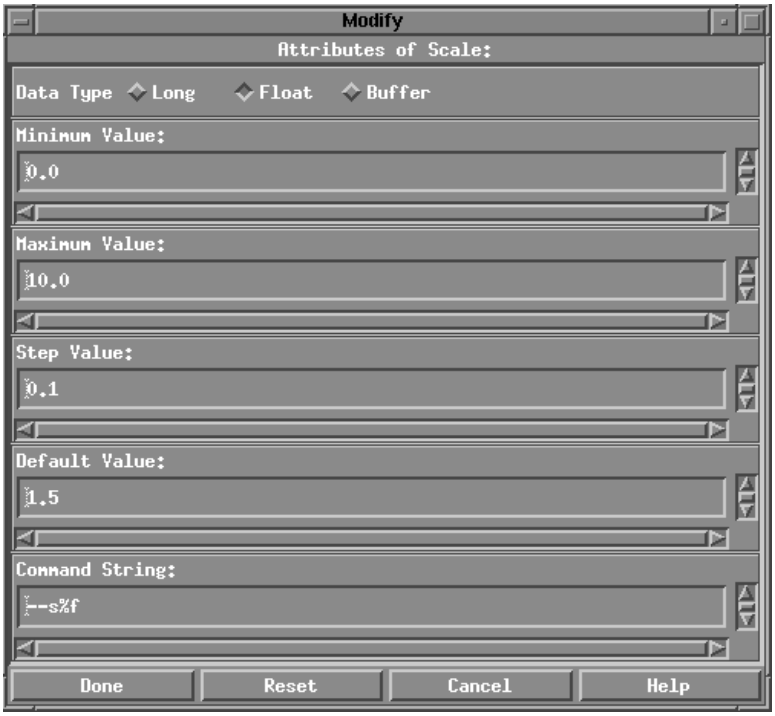

Figure 39: Attributes of the 5x5 Convolution ContinuousParameter "Scale"

type, the information needed to configure the GUI and construct the adjustment messages. Referring to figure 37 the parameter interface aspect contains the following parts:

- Flag Parameters: A Flag Parameter is a Boolean entity which can be set to *true* or *false*. The parameter *ByPass* shown in figure 38, is a parameter of the *5x5Convolution* model. If its state is false, which it is by default, it has no effect on the computation. However, if it is set to true, the convolution algorithm will bypass the function and copy the input image into the output image field.

- Continuous Parameters: A Continuous Parameter is a scalar which can be adjusted between a minimum and maximum value by the specified increments. The parameter *scale* shown in figure 39, is a parameter of the *5x5Convolution* model. Each pixel in the output image is multiplied by the value of *scale* after

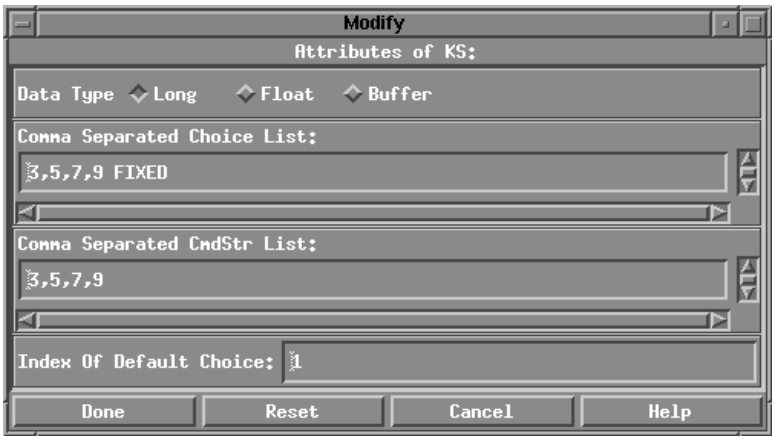

Figure 40: Attributes of the 5x5 Convolution SelectParameter "Kernel Size"

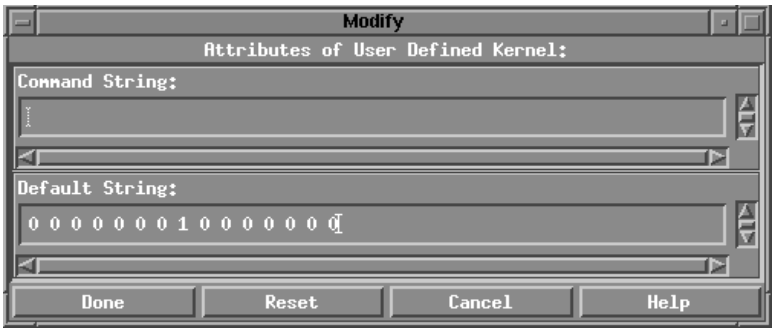

Figure 41: Attributes of the 5x5 Convolution StringParameter "User Defined Kernel"

the mask has been applied to the input data, and the scalar *offset* has been subtracted (see figure 37).

– Select Parameters: A Select Parameter is an entity whose value is chosen from a fixed list of states. The parameter *KS* (Kernel Size) shown in figure 40, is a parameter of the *5x5Convolution* model. The attainable values are 3,5,7,and 9.

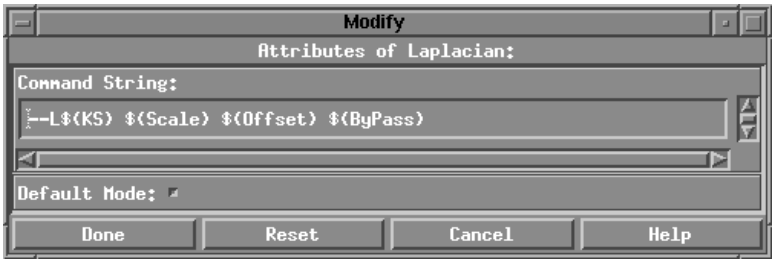

Figure 42: Attributes of the 5x5 Convolution ParameterIfMode "Laplacian"

- String Parameters: A String Parameter is text entered into a field to be parsed by the algorithm's parameter setup function. Some knowledge is required about the expected text format. For instance, the parameter *User Defined Kernel* shown in figure 41 is a textual version of a KS *by* KS convolution kernel. The expected format is a list of $KS * KS$ numbers. The first KS numbers will be interpreted as the first row of the kernel, the second KS the second row, and so on. Note that the string in the *Default* field specifies the 5x5 identity kernel which will have no effect on the data.

- Parameter Interface Modes: The parameter interface mode models are used to provide some level of flexibility in the dynamic parameter user interface. Conditional associations between the parameter interface modes and subsets of the parameters are used to group relevant parameters together to represent *operation modes* of the algorithm. For instance, the parameter interface mode *Laplacian* shown in figure 42 is taken from the *5x5Convolution* model. It cannot be shown well in the figures, but the *Laplacian* mode is conditionally associated with the *Scale*, *Offset*, *KS*, and *ByPass* parameters, which implies that, when the parameter interface is in *Laplacian* mode, the values of each of these parameters will used in constructing the message passed to the *5x5Convolution* algorithm's setup function whenever one of the parameters changes. The current operation mode will be selected from a pull–down menu of the GUI.

Fixed Parameters

Note that the string *FIXED* appears in the *Comma Separated Choice List* field of the *5x5Convolution* model's select parameter *KS* shown in figure 40. This specifies that this parameter may only be set when the system is first started. This parameter cannot be dynamically adjusted. The explanation is simple. The performance characteristics for a convolution with a 5x5 kernel is far different from that with a 3x3 kernel. When the models are interpreted, performance models are built assuming the performance characteristics for a 5x5 convolution. It the kernel size is changed during run–time, the performance characteristics will change. Such parameters must be fixed at interpretation time. Changing a fixed parameter requires a re–interpretation so the real–time constraints will continue to be met. Any of the parameter types can specified as fixed parameters by placing the string FIXED in any of the string fields, and the parameter will will disabled in the GUI.

Application Models

Computational data flows are represented as *application models*. Applications are made up of algorithm models and their inter–connections. They have no relevant attributes, and only a single aspect which contains algorithm parts and connections between the algorithm inputs and outputs. In the XVPE.IPDL model editor an application is built up by merely dragging algorithm models from the model browser into the application model pane and then connecting them. Figures 43 and 44 show the model browser and the application model *Demo4*, respectively.

One notable fact about application models is that they are not *hierarchical*. Applications cannot contain other applications. This simplification was made merely to

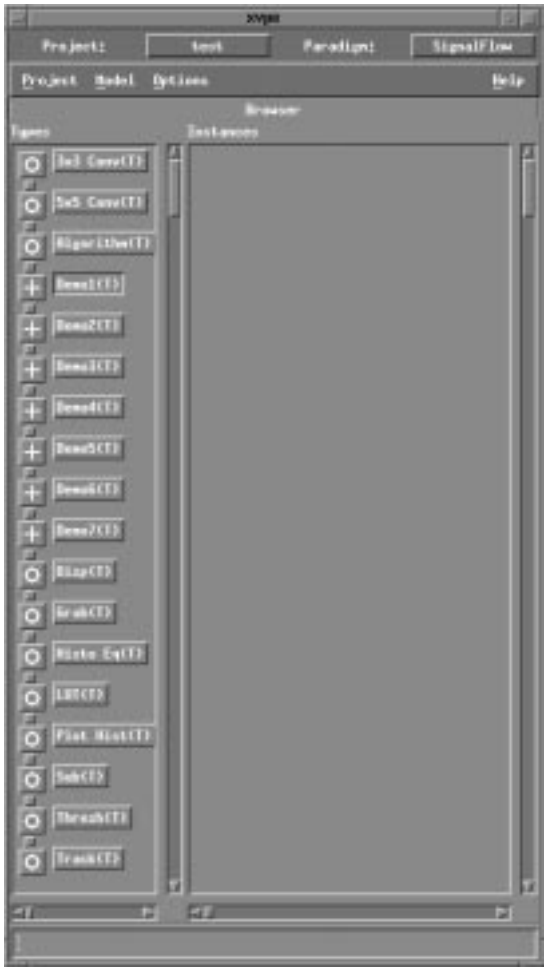

Figure 43: The XVPE.IPDL Model Browser

save time in building the prototype model interpreter. Hierarchy should certainly be added in the future to save time in building application models.

Hardware Models

The hardware paradigm contains models describing the available hardware resources. The types of hardware models are Nodes, HostNodes, and Networks. The meaning and the contents of each of these models are discussed below.

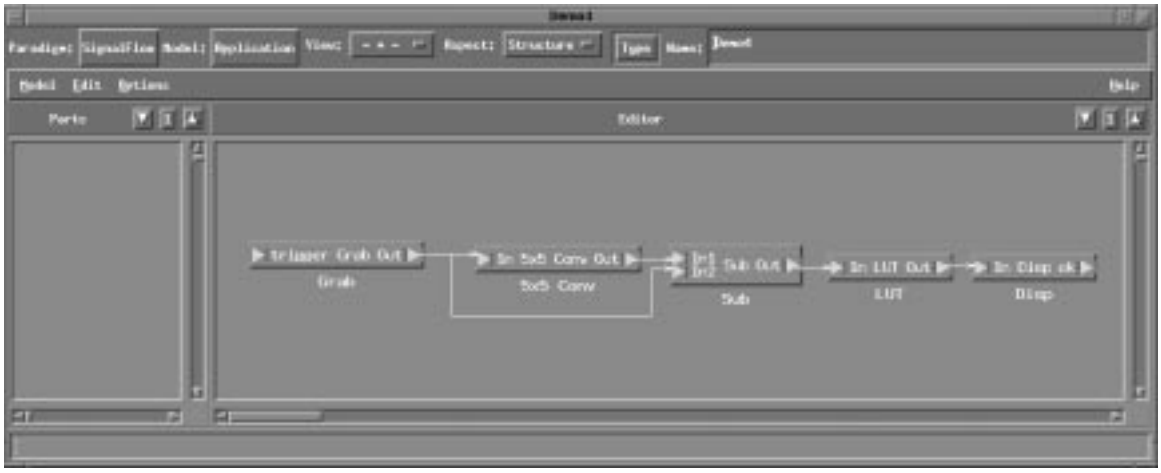

Figure 44: An Application Model

Node Models

A Node model represents one of the network processors (e.g. a C40) which will be performing the image processing computations. The "TIM40"[1] model is shown in figure 45. The model shown is a 40MHz C40 with 2Mbytes of dynamic RAM on each the local and the global bus. Node models have the following attributes and parts:

Node Model Attributes

- CPU Type: This specifies whether the CPU is a C40, or a C44
- Clock Speed (in MHz):
- Performance: This is the MegaFlop rating.
- Memory Type: This can either dynamic ram or static ram (static ram is much faster than dynamic).

[1] A TIM40 is an industry standard C40 package. The board contains a C40, memory interface circuitry, and RAM chips, and fits to a TIM40 motherboard via two connectors.

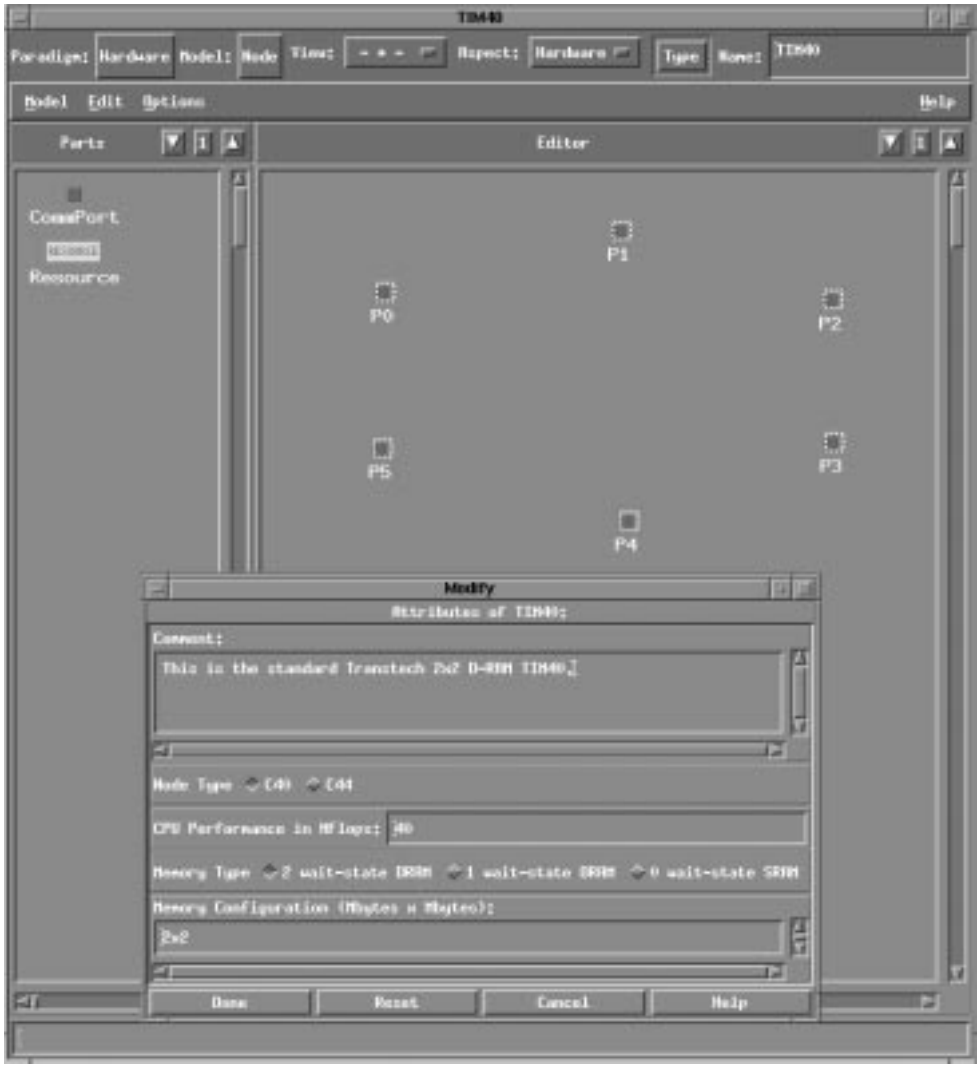

Figure 45: TIM40 Node Model

– Memory Configuration: a formatted string describing how many MBytes of memory is on each of the C40's Global and Local busses[2]. The format is "LocalMemSize x GlobalMemSize" (see figure 45).

[2]The C40 has two memory busses, one called the *Local Bus*, and the other called the *Global Bus*.

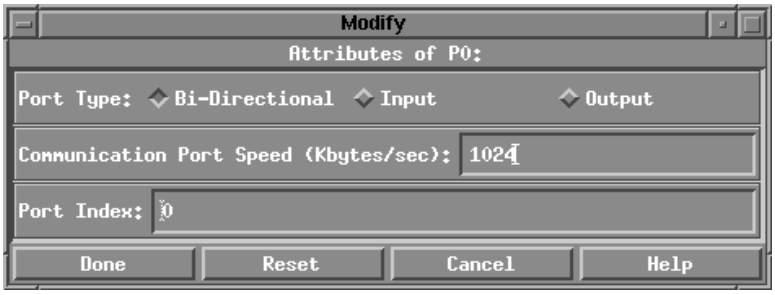

Figure 46: Commport Model Attributes

Node Model Parts

- CommPorts: The ports themselves are atomic models, and have attributes for the port's direction, speed, and port index. Figure 46 shows the attributes of the TIM40 model's port "P0". Note that the "Communication Port Speed" attribute contains $10\frac{MBytes}{sec}$, but a C40 port is rated at $20\frac{MBytes}{sec}$. Practice has shown that this is a more realistic estimate of the real data rate that can be achieved if the ports are connected via ribbon cables of reasonable length.

- Resources: Resource parts are used to indicate that this node has some non–standard capability, such as the ability to digitize video. The types of resources which can be specified are *A/D (Grabber Module)*, *D/A (Display Module)*, or *Other*. The *NEL Grabber* node model is shown in figure 47. It represents a C40 frame grabber TIM. Note the inclusion of the *Grabber HW* resource, which specifies that this node type is capable of digitizing frames of video. The attribute panel of *Grabber HW* are shown in figure 48.

HostNode Models

A hostnode model represents a PC or workstation which may be used to boot and load the C40 network, act as a data source or data sink by providing disk I/O,

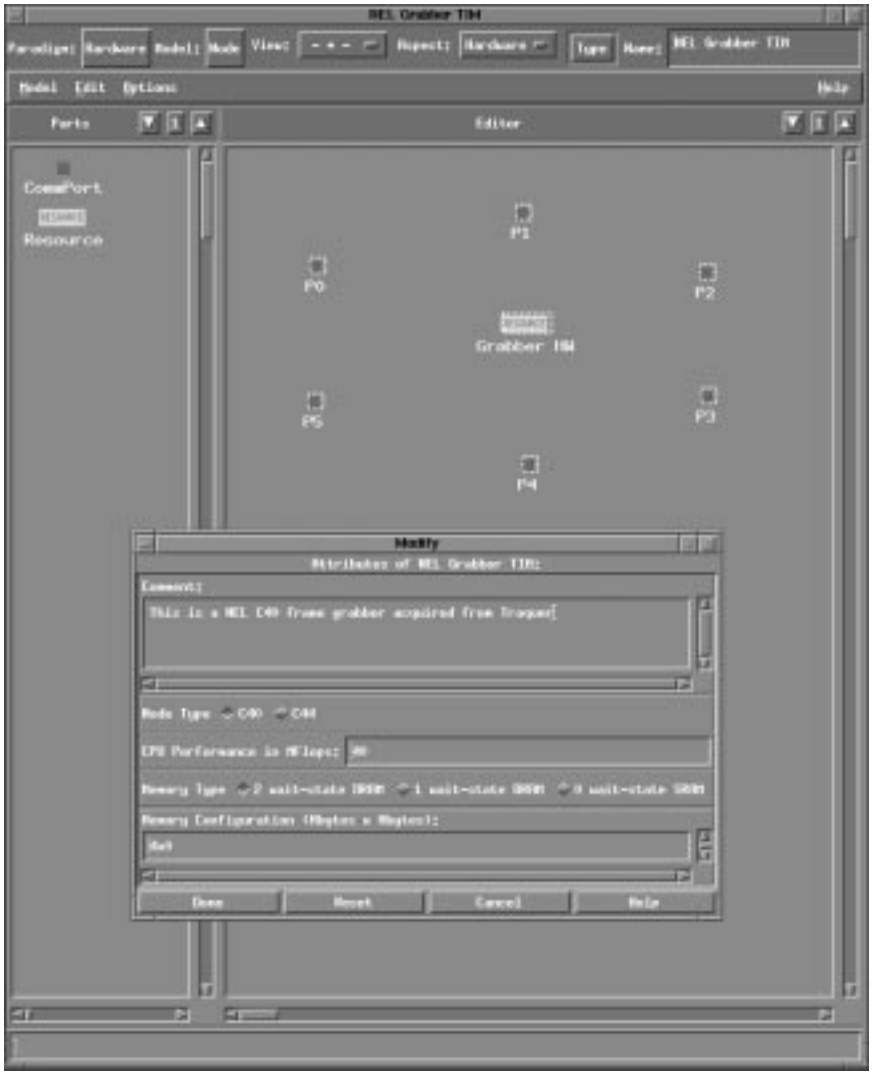

Figure 47: NEL Grabber Node Model

or run the dynamic parameter adjustment GUI. Figure 49 shows a hostnode model from the MIRTIS demo network. The machine is a 66MHz Intel 486 named "venus". The attributes and parts of HostNode models are described below:

HostNode Model Attributes

- CPU Clock Speed (In MHz):

Figure 48: NEL Grabber Node Model

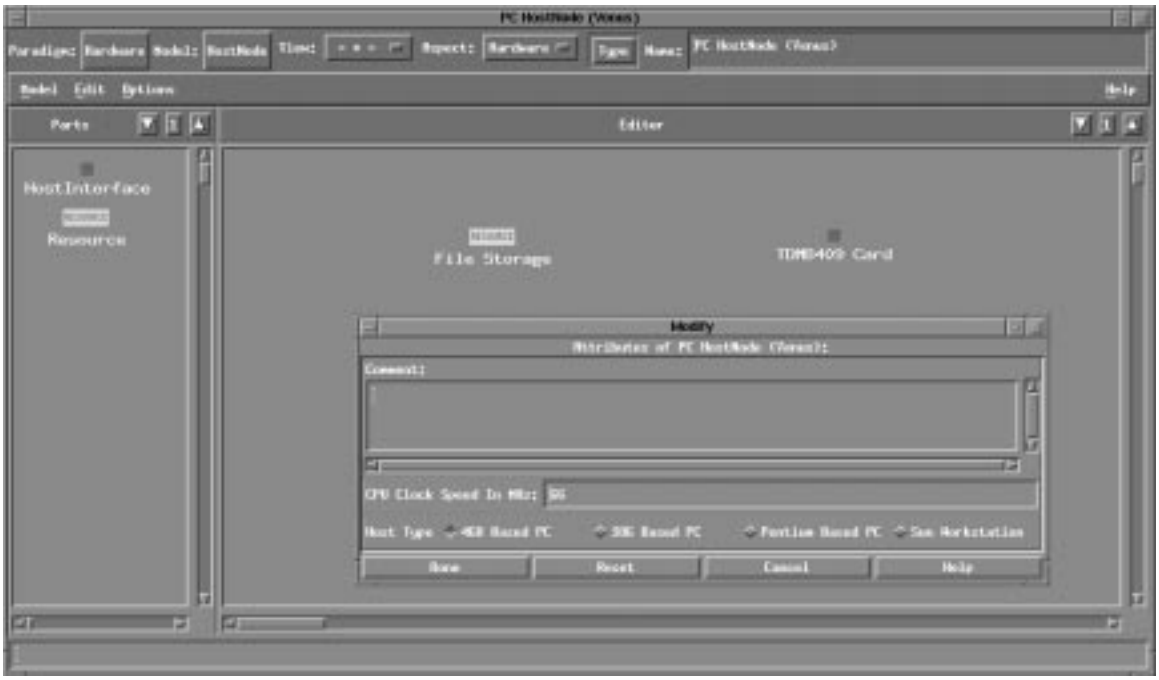

Figure 49: A HostNode Model

– Host Type: This attribute specifies the host node architecture, which can be *386 Based PC*, *486 Based PC*, *Pentium Based PC*, or *Sun Workstation*.

HostNode Model Parts

– Host Interface Cards: These parts represent the network interface cards used to communicate with the C40 network. The two supported interface cards are the *EISA C40 Host Interface Card*, which is a Transtech TDMB409 C40 motherboard, and the *Coreco F64 C40 Card*, which contains lots of image processing support hardware in addition to a C40. Other attributes are the cards's hardware I/O address and the measured port speed (in $\frac{Kbytes}{sec}$). Figure 50 shows the

attributes of the Host Interface Card in *venus*.

– Resources: HostNode model resources are the same as Node model resources. An example of a HostNode with a resource part is the PC HostNode model shown in figure 49. Note that it has a resource part named *File Storage*, the attributes of which are shown in figure 51.

Figure 50: Host Interface Card Attributes

Figure 51: HostNode "File Storge" Resource Attributes

Network Models

Node and Hostnode models are connected together to form *Network* models. Network models have only one aspect, *HardwareAspect*, which contains no attributes. Network models contain Node parts, HostNode parts, Network parts, and Connector parts. Note that the inclusion of Network parts means that Network models are hierarchical. Hierarchy reduces the work required to model large systems. The connector parts are basically the same as CommPorts, but they have no attributes, since they are merely passive connectors. The various assembly levels can be modeled once and replicated to form large network models. For example, the model in figure 52 is of

a VME–based 4–C40 motherboard. Note that there are several connections internal to the card, and 12 ports for external connections. The model shown in figure 53 is a VME chassis made up of 3 VME cards and a C40 display module. The hierarchy allows the many connections internal the VME card to be modeled once and replicated, which saves a vast amount of time. Still another level of hierarchy can be seen in the Network model in figure 54. This model represents a 22 node network made up of the PC Host *Venus*, an *NEL Grabber TIM* attached to the host interface card, the 13 nodes of the *VME Chassis*, and the 8 nodes contained on the EISA motherboards *Card1* and *Card2*

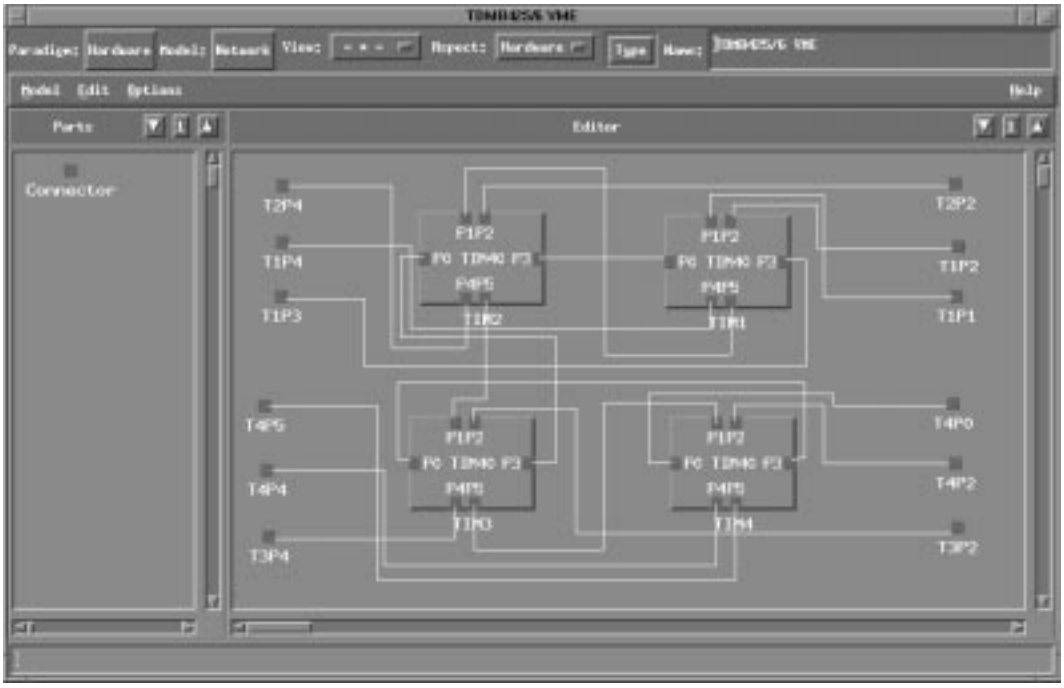

Figure 52: 4–Processor VME Card Network Model

Constraints Models

The Constraints paradigm contains models which specify the timing constraints of a particular application. These are RealTimeConstraints models, and are viewed from

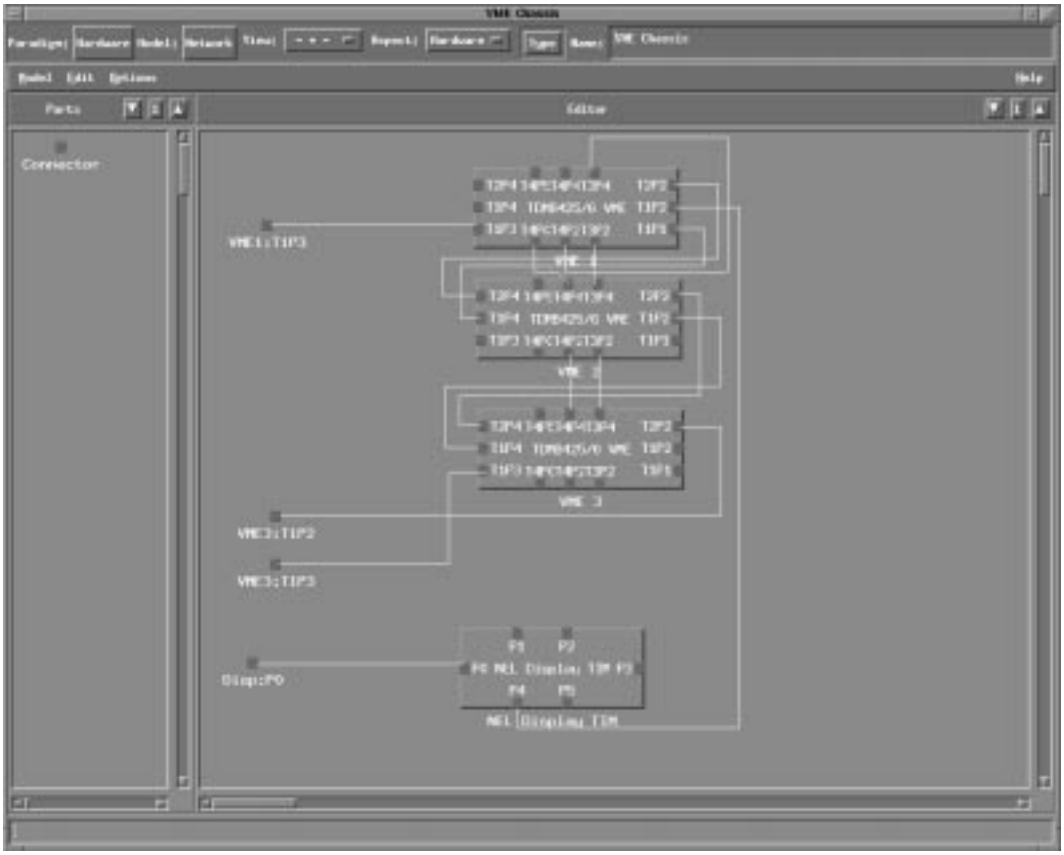

Figure 53: 3–Card VME Chassis Network Model

a single aspect, the *GoalsAspect*. The contents of the RealTimeConstraints models are discussed below.

RealTimeConstraints Model Attributes

These models have a single attribute, *FailureMode*, which can be set to *Best Effort*, or *Hard Real-Time (Fail)*. The *Failure Mode* is used by the interpreter to determine what is to be done in the event that the constraints cannot be achieved with the current hardware architecture. The best effort approach is to sacrifice either throughput or latency (or both) and find a mapping for which the performance will approach the goals. A RealTimeConstraints model called *RTVideo* (Real–Time Video) is shown in

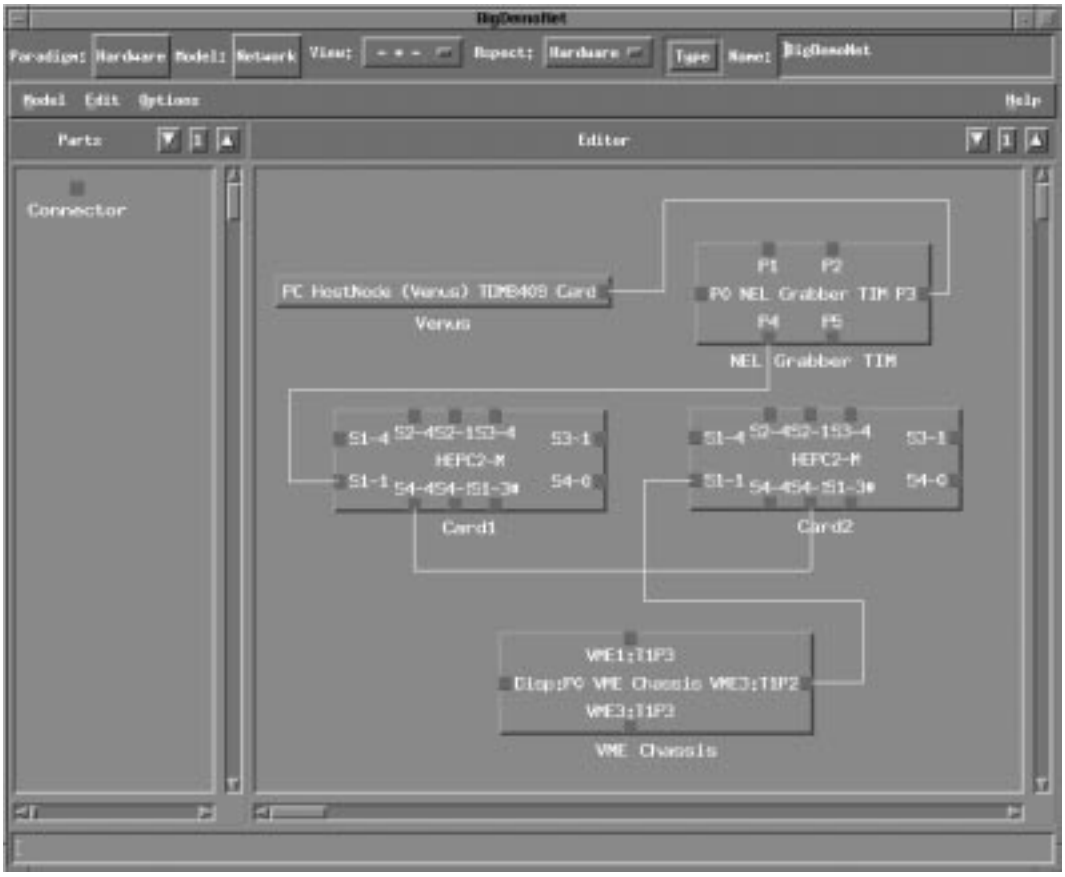

Figure 54: A Large Network Model

figure 55.

RealTimeConstraints Model Parts

RealTimeConstraints models have a single throughput part and a single latency part. These parts have attributes describing the numerical goals in units of $\frac{frames}{second}$ for throughput and $frames$ for latency. Note that latency would normally be specified in $seconds$, but the specification in $frames$ simplified the formation of the latency model. The latency goal in $seconds$ can be obtained by simply dividing the latency goal specified in $frames$ by the $throughput$ specified in $\frac{frames}{second}$. Both throughput and latency parts have a *Nature of Constraint* attribute (see figures 56 and 57), which

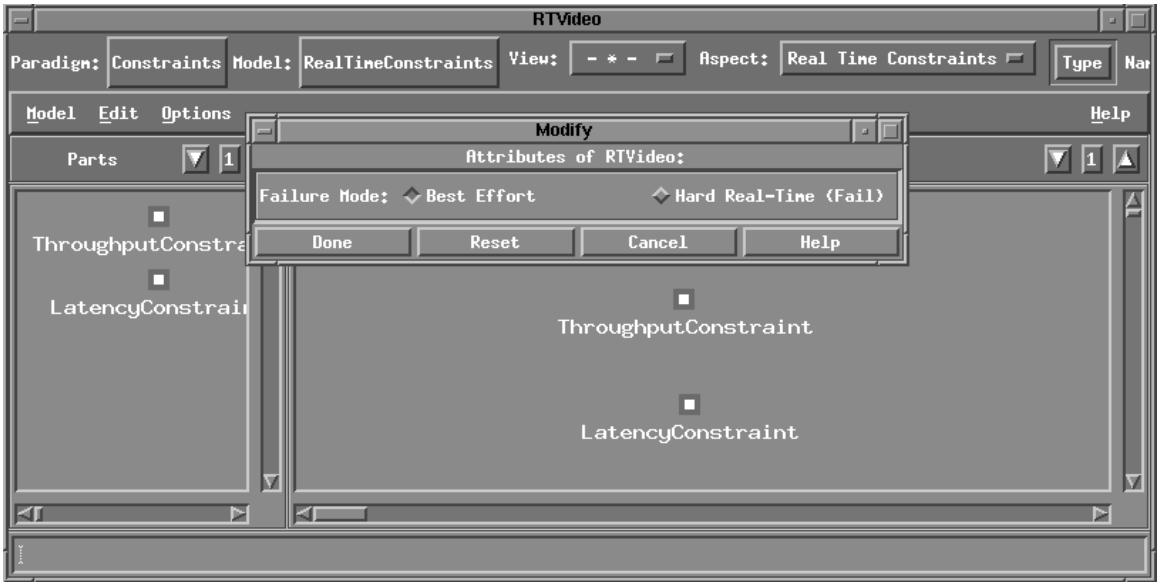

Figure 55: Real–Time Video Timing Constraints

specify whether the constraint is *critical* or *non–critical*, and are analogous to hard and soft real–time.

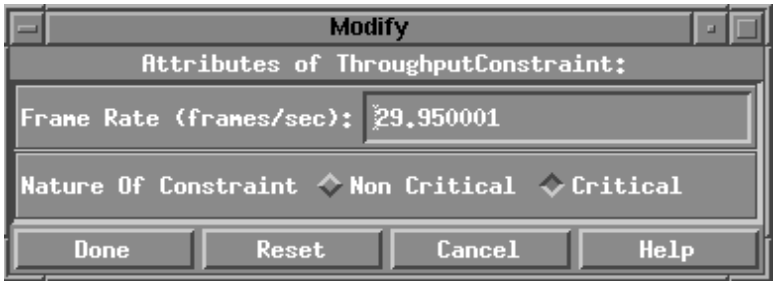

Figure 56: Attributes of Real–Time Video Throughput

In the case of soft real–time, the model indicates that it is acceptable to make concessions in that constraint when making a best–effort approach to meeting the performance goals. In the case of hard real–time, the model indicates that it is if the constraint cannot be met, then the interpreter should indicate a failure to find a viable mapping.

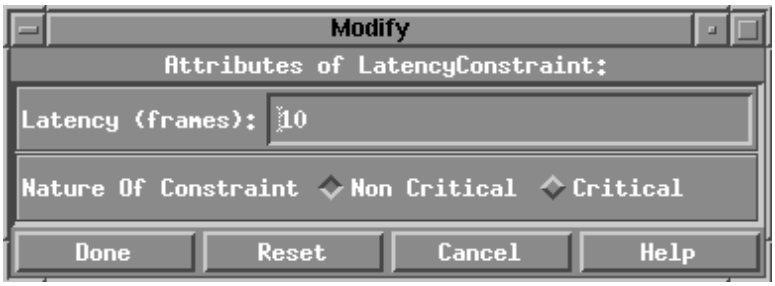

Figure 57: Attributes of Real–Time Video Latency

Appendix C

THE IPDL–VPE CONFIGURATION FILE

```
// vpe configuration file for Image Processing Description Language (IPDL)
// ipdl.mdf

attribute NumControlTokens : field int "Control Tokens:" "1";
attribute SignalType : menu "Signal Type"
                       {
                         "Trigger" TriggerSignalType ;
                         "Long"    LongSignalType ;
                         "Float"   FloatSignalType ;
                         "Buffer"  BufferSignalType ;
                       };
attribute DataType : menu "Data Type"
                       {
                         "Long"    LongDataType ;
                         "Float"   FloatDataType ;
                         "Buffer"  BufferDataType ;
                       };
attribute DataSize : page "Data Size:" (1 64) "1";
attribute ImageDataType : menu "Image Type"
                       {
                         "Packed Bytes"  PbyteImageDataType ;
                         "Long"          LongImageDataType ;
                         "Float"         FloatImageDataType ;
                       };
attribute ImageRows : page "Image Rows:" (1 64) "480";
attribute ImageCols : page "Image Columns:" (1 64) "512";
attribute FrameRate : field float "Frame Rate (frames/sec):" "30.0";
attribute Latency : field int "Latency (frames):" "30";
attribute Hardness : menu "Nature Of Constraint"
                      {
                        "Non Critical" SoftConstraint ;
                        "Critical"     HardConstraint ;
                      };
attribute Comment : page "Comment:" (4 64) "";
attribute PortSpeed : field int "Communication Port Speed (Kbytes/sec):" "8096";

paradigm SignalFlow
{
  classes
    model Process
    {
```

```
  StructureAspect
  {
    attrs
    {
      attr Comment;
    }
    parts { }
  }
  DataDependencyAspect
  {
    attrs
    {
      attr Comment;
    }
    parts { }
  }
  ConstraintsAspect
  {
    attrs
    {
      attr Comment;
    }
    parts { }
  }
  ParameterIfAspect
  {
    attrs
    {
      attr Comment;
    }
    parts { }
  }
}
atom NonImageSignal
{
  attr NumControlTokens;
  attr SignalType;
  attr DataSize;
};
atom ImageSignal
{
  attr NumControlTokens;
  attr ImageDataType;
  attr ImageRows;
  attr ImageCols;
};
atom Parameter { };
```

```
atoms
  InputSignal      "insig.icon"      is_a ( NonImageSignal );
  OutputSignal     "outsig.icon"     is_a ( NonImageSignal );
  InputImage       "inimseq.icon"    is_a ( ImageSignal );
  OutputImage      "outimseq.icon"   is_a ( ImageSignal );
  IntermediateData "imdata.icon"
  {
    attr DataType;
    attr DataSize;
  };
  FlagParameter    "flagpar.icon" is_a ( Parameter )
  {
    OnCmd   : page "On Cmd:" (1 64)  "";
    OffCmd  : page "Off Cmd:" (1 64) "";
    Def     : toggle "Default State:" false;
  };
  ContinuousParameter "contpar.icon" is_a ( Parameter )
  {
    attr DataType;
    Min       : page "Minimum Value:"  (1 64) "";
    Max       : page "Maximum Value:"  (1 64) "";
    Step      : page "Step Value:"     (1 64) "";
    Def       : page "Default Value:"  (1 64) "";
    CmdString : page "Command String:" (1 64) "";
  };
  SelectParameter "selpar.icon" is_a ( Parameter )
  {
    attr DataType;
    Choices    : page "Comma Separated Choice List:" (1 64) "";
    CmdStrings : page "Comma Separated CmdStr List:" (1 64) "";
    Def        : field int "Index Of Default Choice:" "0";
  };
  StringParameter "strpar.icon" is_a ( Parameter )
  {
    CmdString : page "Command String:" (1 64) "";
    Def       : page "Default String:" (1 64) "";
  };
  ParameterIfMode "pifmode.icon"
  {
    CmdString : page   "Command String:" (1 64) "";
    Def       : toggle "Default Mode:"  false;
  };
  DataDependency "datadep.icon"
  {
    DependencySpec : page "Data Dependency String:" (1 64) "";
  };
  Benchmark "benchmrk.icon"
```

```
  {
    FrameRate   : field float "Measured Rate (frames/second):" "5.0";
    FrameRows   : field int   "Frame Height (Rows):"           "480";
    FrameCols   : field int   "Frame Width  (Cols):"           "512";
    attr ImageDataType;
    NProcs      : field int   "Number Of Processors:"          "1";
  };
  Requirement "require.icon" { };
models
  Algorithm is_a ( Process ) primitive
  {
    StructureAspect "Structure"
    {
      icon rect
      {
        left :  InputImages  InputSignals;
        right:  OutputImages OutputSignals;
      };
      font 3;
      color foreground;
      attrs {
        LibFunctionName           : page "Library Function Name:"
                                (1 64) "functionname : pctnode.dwn";
        PartialResultsComboMethod : menu "Combination Method:"
        {
          "Concatenate" ConcatenatePartialResults ;
        };
      }
      parts
      {
        InputImages      : InputImage   link ;
        OutputImages     : OutputImage  link ;
        InputSignals     : InputSignal  link ;
        OutputSignals    : OutputSignal link ;
        Intermediates    : IntermediateData  ;
      }
    }
    DataDependencyAspect "Data Dependency"
    {
      font 3;
      color foreground;
      attrs { }
      conns { }
      conds { }
      parts
      {
        InputImages      : InputImage inherited;
```

```
    OutputImages     : OutputImage inherited;
    Intermediates    : IntermediateData inherited;
    DataDependencies : DataDependency ;
  }
}
ConstraintsAspect "Computational Constraints" {
  font 3;
  color foreground;
  attrs
  {
    MemoryConstraint : page "Memory Requirement (kbytes):"
                                              (1 64) "256";
  }
  conds
  {
    BenchmarkToNodeAssociation Benchmarks :
      { } { NodeRefs };
    AlgorithmRequirementToResourceAssociation Requirements :
      { } { NodeRefs NodeResourceRefs
            HostNodeRefs HostNodeResourceRefs };
  }
  parts
  {
    Benchmarks   : Benchmark ;
    Requirements : Requirement;
    NodeRefs             -> { Hardware :
                              Node     : HardwareAspect } ;
    NodeResourceRefs     -> { Hardware : Node
                                       : HardwareAspect
                                       : Resources } ;
    HostNodeRefs         -> { Hardware : HostNode
                                       : HardwareAspect } ;
    HostNodeResourceRefs -> { Hardware : HostNode
                                       : HardwareAspect
                                       : Resources } ;
  }
}
ParameterIfAspect "Parameter Interface" {
  font 3;
  color foreground;
  attrs { }
  conds
  {
    ModeToParameterAssociation ParameterIfModes :
      { } { FlagParameters ContinuousParameters
            SelectParameters StringParameters };
  }
```

```
        parts
        {
          FlagParameters       : FlagParameter;
          ContinuousParameters : ContinuousParameter;
          SelectParameters     : SelectParameter;
          StringParameters     : StringParameter;
          ParameterIfModes     : ParameterIfMode ;
        }
      }
    }
    Application is_a ( Process ) compound
    {
      StructureAspect "Structure"
      {
        icon rect { };
        font 3;
        color foreground;
        attrs { }
        conns
        {
          SignalFlow { 1 solid line arrow } :
            { Algorithms OutputSignals -> Algorithms  InputSignals };
          ImageFlow { 1 solid line arrow } :
            { Algorithms  OutputImages -> Algorithms InputImages };
        }
        parts
        {
          Algorithms      : Algorithm hierarchy;
        }
      }
    }
}

paradigm Hardware
{
  classes
    model HW
    {
      HardwareAspect
      {
        attrs
        {
          attr Comment;
        }
      }
    }
    atom HWConnector { };
```

```
atoms
  CommPort "port.icon" is_a ( HWConnector )
  {
    PortType : menu "Port Type:"
    {
      "Bi-Directional" BidirectionalType ;
      "Input"          InputType;
      "Output"         OutputType;
    };
    attr PortSpeed;
    PortIndex : field int "Port Index:" "0";
  };
  Connector "connect.icon" is_a ( HWConnector );
  Resource "resource.icon"
  {
    ResourceType : menu "Resource Type:"
    {
      "A/D (Grabber Module)" GrabberHW ;
      "D/A (Display Module)" DisplayHW ;
      "Other"                OtherHW   ;
    };
  };
  HostInterface "hostif.icon" is_a ( HWConnector )
  {
    InterfaceType : menu "Network Interface Type"
    {
      "EISA C40 Host Interface Card" TDMB409HostInterfaceCard ;
      "Coreco F64 C40 Card"          CorecoF64C40Card ;
    };
    HardwareAddress : field int "HW Address Of Host IF Card:"
                                                        "0x200";
    attr PortSpeed;
  };
models
  Node is_a ( HW ) primitive
  {
    HardwareAspect "Hardware"
    {
      icon rect;
      font 3;
      color foreground;
      attrs
      {
        CPUType    : menu "Node Type"
        {
          "C40" C40NodeType ;
          "C44" C44NodeType ;
```

```
      };
      CPUPerf : field int "CPU Performance in MFlops:" "40";
      MemType  : menu "Memory Type"
      {
        "2 wait-state DRAM" TWS_DRAM;
        "1 wait-state DRAM" OWS_DRAM;
        "0 wait-state SRAM" ZWS_SRAM;
      };
      MemSize : page "Memory Configuration (Mbytes x Mbytes):"
                                                    (1 64) "2x2";
    }
    parts
    {
      CommPorts : CommPort link;
      Resources : Resource;
    }
  }
}
HostNode is_a ( HW ) primitive
{
  HardwareAspect "Hardware"
  {
    icon rect
    {
      right:  HostInterfaces;
    };
    font 3;
    color foreground;
    attrs
    {
      HostSpeed: field int "CPU Clock Speed In MHz:" "66";
      HostType : menu "Host Type"
      {
        "468 Based PC"     PC486Host;
        "386 Based PC"     PC386Host;
        "Pentium Based PC" PCPentiumHost;
        "Sun Workstation"  SunPWHost;
      };
    }
    parts
    {
      HostInterfaces : HostInterface link;
      Resources      : Resource ;
    }
  }
}
Network is_a ( HW ) compound
```

```
    {
      HardwareAspect "Hardware"
      {
        icon rect;
        font 3;
        color foreground;
        conns
        {
          HWConn { 1 solid line butt } :
                 { Connectors               -> SubNetworks Connectors }
                 { Connectors               -> Nodes CommPorts }
                 { SubNetworks Connectors   -> SubNetworks Connectors }
                 { SubNetworks Connectors   -> Connectors }
                 { SubNetworks Connectors   -> Nodes CommPorts }
                 { SubNetworks Connectors   -> HostNodes HostInterfaces }
                 { Nodes CommPorts          -> SubNetworks Connectors }
                 { Nodes CommPorts          -> Nodes CommPorts }
                 { Nodes CommPorts          -> Connectors }
                 { HostNodes HostInterfaces -> Nodes CommPorts }
                 { HostNodes HostInterfaces -> SubNetworks Connectors };
        }
        parts
        {
          Nodes       : Node hierarchy ;
          SubNetworks : Network hierarchy ;
          HostNodes   : HostNode hierarchy ;
          Connectors  : Connector link;
        }
      }
    }
}

paradigm Constraints {
  atoms
    ThroughputConstraint "thruput.icon"
    {
      attr FrameRate;
      attr Hardness;
    };
    LatencyConstraint    "latency.icon"
    {
      attr Latency;
      attr Hardness;
    };
  models
    RealTimeConstraints primitive {
      GoalsAspect "Real Time Constraints" {
```

```
        icon oval;
        font 3;
        color foreground;
        attrs {
          FailureMode : menu "Failure Mode:" {
            "Best Effort"           BestEffortMode;
            "Hard Real-Time (Fail)" HardRealTimeMode;
          };
        }
        parts
        {
          ThroughputConstraints : ThroughputConstraint ;
          LatencyConstraints    : LatencyConstraint ;
        }
      }
    }
}
```

Appendix D

THE IMAGE PROCESSING LIBRARY FUNCTION "CONV.C"

```
/*
 * conv.h
 */

#include <pct.h>

#define FNMBASE conv

typedef struct {
  int inited;
  int type;
  int kernel_size;
  float scale;
  float offset;
  float *float_input;
  float *float_output;
  float *kern;
} CNTX_STRUCT;

#define inited ((*cntx)->inited)
#define type ((*cntx)->type)
#define kernel_size ((*cntx)->kernel_size)
#define scale ((*cntx)->scale)
#define offset ((*cntx)->offset)
#define float_input ((*cntx)->float_input)
#define float_output ((*cntx)->float_output)
#define kern ((*cntx)->kern)

#define LAPLACIAN3x3 0
#define LOG 1
#define AVE 2
#define TEST 3
#define USER_DEFINED 4

#define CNTX_DEFAULTS {0,LAPLACIAN3x3,3,1.0,0.0,NULL,NULL,NULL}

typedef struct {
  unsigned long *input_img;
  unsigned long input_img_cid;
} INPUT_STRUCT;
```

```
typedef struct {
  unsigned long *output_img;
  unsigned long output_img_cid;
} OUTPUT_STRUCT;

#define input_img  (inputs->input_img)
#define output_img (outputs->output_img)
#define input_img_cid  (inputs->input_img_cid)
#define output_img_cid (outputs->output_img_cid)

extern void aconv3x3(float *,float *,float *,float,unsigned long,unsigned long);
extern void aconv5x5(float *,float *,float *,float,unsigned long,unsigned long);
extern void pb2f(unsigned long *,float *,unsigned long);
extern void f2pb(float *,unsigned long *,unsigned long);
extern void trunf_and_pack2pb(float *,unsigned long *,unsigned long);
extern void trunabsf_and_pack2pb(float *,unsigned long *,unsigned long);

/*
 * conv.c
 */

#include "pct.h"
#include "conv.h"
#include "pctiplib.h"

static CNTX_STRUCT default_cntx=CNTX_DEFAULTS;

float lap_kernel[3*3] = {  0.0, 1.0, 0.0 ,
                           1.0,-4.0, 1.0 ,
                           0.0, 1.0, 0.0 };

float user_defined_kernel[25] = {0,0,0,0,0,
                                 0,0,0,0,0,
                                 0,0,1,0,0,
                                 0,0,0,0,0,
                                 0,0,0,0,0};

#define LOG3x3_EPSILON 1.85
#define LOG5x5_EPSILON 0.60

static void make_log(int width,int height, float *buff,CNTX_STRUCT **cntx) {
  int i,j,iorg,jorg;
  float sigma,thing,epsilon= (kernel_size==3 ? LOG3x3_EPSILON :
                              ((kernel_size==5) ? LOG5x5_EPSILON : -1));
  float ee=2.718281828,pi=3.141592654;
   iorg = (height-1)/2;
```

```
   jorg = (width-1)/2;
   sigma = ((float)width+(float)height)/12.0; /* average of w,h/6 */
   for(i=0;i<height;i++) {
      for(j=0;j<width;j++) {
         thing = ((i-iorg)*(i-iorg) + (j-jorg)*(j-jorg))/(sigma*sigma);
         buff[i*width+j] = (1.0/epsilon)*(thing-2.0)*
                           pow(ee,-thing/2.0)/(2.0*pi*sigma*sigma*sigma*sigma);
      }
   }
}

static void make_ave(int width,int height, float *buff) {
  int i,j;
   for(i=0;i<height;i++) {
      for(j=0;j<width;j++) {
         buff[i*width+j] = 1.0/((float)width*height);
      }
   }
}

void SETUP(int argc, char* argv[])
{
 int i,j,v;
   TRACE();
   CNTX_INIT();
   type=LAPLACIAN3x3;
   kernel_size=3;
   scale=1.0;
   offset=0.0;
   for(i = 0; i < argc; i++) {
     if((argv[i][0] == '-')&&(argv[i][1] == '-'))
       switch(argv[i][2]) {
         case 'L':
           type=LOG;
           if(argv[i][3]!='d') {
             v=atoi(&(argv[i][3]));
             if((v==3)||(v==5)) kernel_size=v;
             else kernel_size=3;
           }
           break;
         case 'l':
           type=LAPLACIAN3x3;
           kernel_size=3;
           break;
         case 's':
           scale=atof(&(argv[i][3]));
           break;
```

```
      case 'o':
        offset=atof(&(argv[i][3]));
        break;
      case 'a':
        type=AVE;
        if(argv[i][3]!='d') {
          v=atoi(&(argv[i][3]));
          if((v==3)||(v==5)) kernel_size=v;
          else kernel_size=3;
        }
        break;
      case 'K':
        type=USER_DEFINED;
        v=atoi(&(argv[i][3]));
        if((v==3)||(v==5)) kernel_size=v;
        else kernel_size=3;
        j=0;
        while(j<kernel_size*kernel_size) {
          user_defined_kernel[j++] = atof(argv[++i]);
        }
        break;
      case 't':
      default:
        type=TEST;
        kernel_size=3;
        break;
    }
}
_pct_set_overlap((kernel_size-1)/2,(kernel_size-1)/2);

TRACE();
if(kern) _pct_free(kern);
kern=(float *)_pct_ocbuf_alloc(kernel_size*kernel_size*sizeof(float));
if(!kern) _pct_error(__LINE__);
if(type==LAPLACIAN3x3) {
  TRACE();
  memcpy(kern,lap_kernel,9*sizeof(float));
}
if(type==USER_DEFINED) {
  TRACE();
  memcpy(kern,user_defined_kernel,kernel_size*kernel_size*sizeof(float));
}
if(type==LOG) {
  TRACE();
  make_log(kernel_size,kernel_size,kern,cntx);
}
if(type==AVE) {
```

```
    TRACE();
    make_ave(kernel_size,kernel_size,kern);
  }
  for(i=0;i<kernel_size*kernel_size;i++) {
    kern[i]*=scale;
  }
  TRACE();
}

void COMPUTE()
{
 long rows=_pct_get_equal_imrows();
 long cols=_pct_get_imcols();
  CNTX_INIT();
  if(!inited) {
    float_input=(float *)_pct_buf_alloc(rows*cols,GLOBAL_BUS,LOCAL_BUS,ONCHIP_BUS);
    if(!float_input) _pct_error(__LINE__);
    float_output=(float *)_pct_buf_alloc(rows*cols,LOCAL_BUS,GLOBAL_BUS,ONCHIP_BUS);
    if(!float_output) _pct_error(__LINE__);
    inited=1;
  }
  if(output_img==NULL) {
    output_img=(image)_pct_alloc_connection(output_img_cid,_pct_get_local_ims());
    if(output_img==NULL) _pct_error(__LINE__);
  }
  TRACE();
  pb2f(input_img,float_input,rows*cols);
  TRACE();
  switch(kernel_size) {
    case(3):
      TRACE();
/*
 * call the assembly 3x3 convolution routine
 */
      aconv3x3(float_input,float_output,kern,offset,rows,cols);
      TRACE();
      break;
    case(5):
      TRACE();
/*
 * call the assembly 5x5 convolution routine
 */
      aconv5x5(float_input,float_output,kern,offset,rows,cols);
      TRACE();
      break;
    case(5):
      TRACE();
```

```
/*
 * call the assembly 7x7 convolution routine
 */
       aconv7x7(float_input,float_output,kern,offset,rows,cols);
       TRACE();
       break;
     case(5):
       TRACE();
/*
 * call the assembly 9x9 convolution routine
 */
       aconv9x9(float_input,float_output,kern,offset,rows,cols);
       TRACE();
       break;
     default:
       _pct_error(__LINE__);
   }
   TRACE();
/*
 * truncate the float buffer at 0 and 256 and pack bytes
 */
   trunf_and_pack2pb(float_output,output_img,rows*cols);
   TRACE();
}
```

Appendix E

EXAMPLE PCT CONFIGURATION FILES

The following shows the configuration files generated by running the interpreter using the application *Demo1*, the nine processor network *DemoNet*, and the constraints *LowLatencyVideo*. These files are used to boot the network and configure the PCT scheduler and communication engines.

An Example Boot File

```
;
; <<< PCT boot file for application model "Demo1" >>>
;
; <<< Warning: This was Generated by the MIRTIS model interpreter >>>
; <<< Pipeline Cut-Through C40 boot file:                          >>>
; <<< Application model = "Demo1"                                  >>>
; <<< Network model     = "DemoNet"                                >>>
; <<< System Throughput =  4.69 frames/sec                         >>>
; <<< System Latency    =     3 frames                             >>>
; <<< Warning: This was Generated by the MIRTIS model interpreter >>>

 0  1 0 \mirtis\iplib\grab.dwn -n0 -N1 -g0 -M -i3 -o4 -G0 -L4096
 1  2 4 \mirtis\iplib\pctnode.dwn -n0 -N1 -g2 -S -M -i1 -o0 -G2048 -L2048
 2  3 0 \mirtis\iplib\pctnode.dwn -n0 -N6 -g1 -S -M -i3 -o0 -G2048 -L2048
 3  4 0 \mirtis\iplib\pctnode.dwn -n1 -N6 -g1 -S -M -i3 -o0 -G2048 -L2048
 4  5 0 \mirtis\iplib\pctnode.dwn -n2 -N6 -g1 -S -M -i3 -o4 -G2048 -L2048
 5  6 4 \mirtis\iplib\pctnode.dwn -n3 -N6 -g1 -S -M -i1 -o2 -G2048 -L2048
 6  7 2 \mirtis\iplib\pctnode.dwn -n4 -N6 -g1 -S -M -i5 -o2 -G2048 -L2048
 7  8 2 \mirtis\iplib\pctnode.dwn -n5 -N6 -g1 -S -M -i5 -o2 -G2048 -L2048
 8  9 2 \mirtis\iplib\pctnode.dwn -n0 -N1 -g3 -S -M -i5 -o1 -G2048 -L2048
 9 10 1 \mirtis\iplib\disp.dwn -n0 -N1 -g4 -M -i4 -G0 -L4096
```

An Example Schedule File

```
; PCT scedule file for application model "Demo1"
; <<< This File Was Automatically Generated >>>
```

```
SGRP 0 ; block id #0
-1 ; no input connection
0 ; output connection id #0
0  ; 0 local connections
1 ; number of functions = 1
SFUN 2 ; function id #2 ("Grab")
grab
1 ; 0 connected input[s]
1 0 ; 1 connected output[s]
  ; initialization string
EFUN
EGRP
SGRP 2 ; block id #2
0 ; input connection id #0
1 ; output connection id #1
0  ; 0 local connections
1 ; number of functions = 1
SFUN 3 ; function id #3 ("LUT")
lut
1 0 ; 1 connected input[s]
1 1 ; 1 connected output[s]
--g0.590000  ; initialization string
EFUN
EGRP
SGRP 1 ; block id #1
1 ; input connection id #1
2 ; output connection id #2
0  ; 0 local connections
1 ; number of functions = 1
SFUN 0 ; function id #0 ("5x5 Conv")
conv
1 1 ; 1 connected input[s]
1 2 ; 1 connected output[s]
--L5 --s1.500000 --o128  ; initialization string
EFUN
EGRP
SGRP 3 ; block id #3
2 ; input connection id #2
3 ; output connection id #3
0  ; 0 local connections
1 ; number of functions = 1
SFUN 4 ; function id #4 ("Threshold")
lut
1 2 ; 1 connected input[s]
1 3 ; 1 connected output[s]
--tl100 --tu256  ; initialization string
EFUN
```

```
EGRP
SGRP 4 ; block id #4
3 ; input connection id #3
-1 ; no output connection
0  ; 0 local connections
1 ; number of functions = 1
SFUN 1 ; function id #1 ("Disp")
disp
1 3 ; 1 connected input[s]
1 ; 0 connected output[s]
  ; initialization string
EFUN
EGRP
```

REFERENCES

[1] B. Abbott, T. Bapty, C. Biegl, G. Karsai, and J. Sztipanovits, "Model–Based Software Synthesis," IEEE Software, May, 1993: pp. 42–52.

[2] B. Abbott, A. Ledeczi: "TICK: a TMS320C40 Utility Program," Proceedings of the International Conference on Signal Processing Applications and Technology (ICSPAT94), Dallas, Texas, 1994.

[3] D. Fernandez–Baca, "Allocating Modules to Processors in a Distributed System," IEEE Transactions on Software Engineering, Vol 15, No 11., November 1989.

[4] Bapty, T., et al: "Parallel Turbine Engine Instrumentation System," Proceedings of Computing in Aerospace 9, San Diego, Ca, 1993, pp. 423–433.

[5] Bapty, T., Ledeczi, A, Sztipanovits, J., Davis, J., "Synthesis of Large-Scale Real-Time Instrumentation Systems using Model-Based Techniques", Proceedings of the Software Engineering Research Forum, Boca Raton, FL, 1995.

[6] D.H. Ballard, "Animate Vision," Artificial Intelligence, 1991. 48(1): p. 57-86.

[7] S.H. Bokhari, "Partitioning Problems in Parallel, Pipelined and Distributed Computing," IEEE Transactions on Computers, Vol. 37, No1, Jan. 1988, pp. 48–57.

[8] J.A. Bondy and U.S.R. Murty, Graph Theory With Applications, North–Holland, New York, 1976.

[9] S. Borkar, R. Cohn, G. Cox, S. Gleason, T. Gross, H.T. Kung, M. Lam, B. Moore, C. Peterson, J. Pieper, L. Rankin, P.S. Tseng, J. Sutton, J. Urbanski, and J. Webb, "iWarp: An integrated Solutions to High–Speed Parallel Computing," Supercomputing '88, Kissimmee, Florida, November, 1988.

[10] Brown, C.M., "Parallel Vision with the Butterfly Computer," Third International Conference on Supercomputing, 1988, Boston, MA.

[11] V. Cantoni and S. Levialdi, "Languages and Environments for Vision Applications," IEEE Software, November 1991: p. 34–36.

[12] Carnes J. R., Davis, W. S., Biegl, C., Karsai, G.: "Integrated Modeling for Planning, Simulation and Diagnosis", Proc. of the IEEE Conference on AI Simulation & Planning in High Autonomy Systems, Cocoa Beach, FL, April 1991.

[13] A. Choudhary and S. Ranka, "Parallel Processing for Computer Vision and Image Understanding," IEEE Computer, Feb 1992: p. 7–10.

[14] Crisman, J.D. and J.A. Webb, "The Warp Machine on Navlab," IEEE Transactions on Pattern Analysis and Machine Intelligence, 1991. 13(5): p. 451-65.

[15] Davidson, David B., "Parallel Computing: Computing for the Impatient and Indomitable," IEEE Potentials, April/May 1995, pp. 6–10.

[16] Deguchi, K., K. Tago, and I. Morishita, "Integrated Parallel Image Processings on a Pipelined MIMD Multi-Processor System PSM," 10th International Conference on Pattern Recognition, 1990, Atlantic City, NJ.

[17] K. L. Eager, J. Zahorjan, and E. D. Lazowska, "Speedup Versus Efficiency in Parallel Systems," IEEE Transactions on Computers, Vol 38, No. 3, March 1989.

[18] M. J. Flynn, "Some Computer Organizations and Their Effectiveness," IEEE Trans. on Comp., C-21, no. 9, Sept. 1972, pp. 948–960.

[19] M.R. Garey, D.S. Johnson, "Computers and Intractability: A Guide to the Theorey of NP-Completeness," W.H. Freeman and Company, New York, 1979.

[20] A. Gunzinger, A., "Concept and Realization of a Heterogeneous Multiprocessor System for Real–Time Image Processing," Computer Architectures for Machine Perception. 1991. Paris, France: D.G.A./E.T.C.A., C.N.R.S./I.E.F. and

[21] Halang W. A., P. Laplante, A. D. Stoyenko, "Meeting the Strict Real-Time Requirements of Distributed Continuous Multimedia Applications," 1994 IEEE International Conference on Systems, Man and Cybernetics, San Antonio, Texas, October 1994.

[22] Y.K. Ham, G.T. Park, "Recognition of Raised Characters for Automatic Classification of Rubber Tires," Optical Engineering, vol. 34, no. 1, January 1995.

[23] B.K.P. Horn, Robot Vision, The MIT Press, Cambridge, MA., 1986.

[24] L.H. Jamieson, et al., "A Software Envoironment for Parallel Computer Vision," IEEE Computer, February 1992, pp 73–77.

[25] G. Karsai, "A Configurable Visual Programming Environment," Computer, March 1995, pp. 36–44.

[26] G. Karsai, J. Sztipanovits, S. Padalkar, C. Biegl, K. Okuda, N. Miyasaka, "Model-Based Intelligent Process Control for Cogenerator Plants," Journal of Parallel and Distributed Computing, Vol. 15, No. 6, pp. 90-102, 1992.

[27] P. A. Laplante, "Architectural approaches aid in image processing," Electronic Imaging, vol. 2, no. 4, November 1992, pp. 4–8.

[28] P. A. Laplante, "Issues in Real-Time Image Processing," Proceedings of the 1993 IEEE Systems, Man, and Cybernetics Conference, Le Touquet, France, October, 1993, pp. 323-326.

[29] P. A. Laplante, D. Zalewski, "A Real-Time Image Filtration System," Proceedings of the First IEEE Workshop on Real-Time Applications, New York City, May, 1993, pp. 153-154.

[30] P. A. Laplante, "A Real-Time Image Processing Language?," Proceedings of the NATO ASI on Real-Time Computing, St. Marten, October 1992.

[31] A. Ledeczi, "Parallel Systems With Flexible Topology," Ph. D. Dissertation, Vanderbilt University, 1995.

[32] E. A. Lee, and D. G. Messerschmitt, "Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing," IEEE Transactions on Computers, 36(1), pp. 24–35, January 1987.

[33] K. Murakami, et al, "The Kyushu University Reconfigurable Parallel Processor-Design Philosophy and Architecture," Information Processing 89, Proceedings of the IFIP 11th World Computer Congress, 1989, San Francisco, Ca.

[34] A. Misra, G. Karsai, J. Sztipanovits, A. Ledeczi, M. Moore, E. Long, "A Model–Integrated Information System for Increasing Throughput in Discrete Manufacturing," 1997 International Conference on Engineering of Computer Based Systems (ICBS97), Monterey Ca, March 1997 (In Press).

[35] M.S. Moore, G. Karsai, and J. Sztipanovits, "Model–Based Programming for Parallel Image Processing", Proceedings of the First IEEE International Conference on Image Processing (ICIP94), Nov. 1994.

[36] M.S. Moore, "A DSP–Based Real–Time Image Processing System", Proceedings of the 6th International Conference on Signal Processing Applications and Technology (ICSPAT95), Boston, Ma., Oct. 24–26, 1995.

[37] M.S. Moore, J. Nichols, "Model–Based Synthesis of a Real–Time Image Processing System", Proceedings of the First IEEE International Conference on Engineering of Complex Computer Systems (ICECCS95), Ft. Lauderdale, Fla., Nov. 6–10, 1995.

[38] H.R. Myler and A.R. Weeks, The Pocket Handbook of Image Processing Algorithms in C, PTR Prentice Hall, Englewood Cliffs, NJ., 1993.

[39] P.J. Narayanan, L.T. Chen, and L.S. Davis, "Effective Use of SIMD Parallelism in Low- and Intermediate-Level Vision," Computer, 1992. 25(2): pp 68-73.

[40] S. Padalkar, Sztipanovits, J., Karsai, G., Miyasaka, N., Okuda, K., "Real-Time Fault Diagnostics with Multiple-Aspect Models", Proc. of the 1991 IEEE International Conference on Robotics and Automation, pp. 803-808, Sacramento, CA, April, 1991.

[41] R.A. Peters II, "A new algorithm for image noise reduction using mathematical morphology," IEEE Transatctions on Image Processing, vol 4., No. 3, pp. 554-568, May, 1995.

[42] J. Rasure, D. Argiro, and C. Williams, "Visual Language and Software Development Environment for Image Processing," International Journal of Imaging Systems and Technology, August 1, 1990: pp 183–199.

[43] A.P. Reeves, "Parallel Programming for Computer Vision," IEEE Software, November 1991: pp 51–59.

[44] C.L. Seitz, "Concurrent Architectures,", from VLSI and Parallel Computation, edited by Roberto Suaya, Morgan Kaufmann Publishers, Inc., Palo Alto, Ca., 1990.

[45] H. Siegel, J. Armstrong, and D. Watson, "Mapping Computer–Vision–Related Tasks onto Reconfigurable Parallel–Processing Systems," IEEE Computer, February 1992, pp 54–63.

[46] G.J. Lipovski, M. and Malek, "Parallel Computing: Theory and Comparisons," John Wiley & Sons, New York, 1987.

[47] P. Duclos, et al., "Image Processing on a SIMD/SPMD Architecture: Opsila," Proceedings of the Ninth International Conference on Pattern Recognition, IEEE CS Press, Los Alamitos, Ca., 1988.

[48] J. Sztipanovits, B. Abbott, and C. Biegl, "Programming Environment for Parallel Image Processing," *White Paper* Prepared for Sverdrup Technology, Inc., September 1992.

[49] R.I. Taniguchi, and M. Amimiya, "AMP: An Autonomous Multi-processor for Image Processing and Computer Vision," 10th International Conference on Pattern Recognition. 1990. Atlantic City, NJ.

[50] Texas Instruments, "TMS320C4x User's Guide", Texas Instruments, May 1993.

[51] V. Vuohtoniemi and T. Seppaenen, "Transputer-based Machine Vision Systems Research at the University of Oulu," Nordic Transputer Applications, 1st and 2nd Nordic Transputer Seminars, 1991, Turku, Finland and Trondheim, Norway.

[52] A.M. Wallace, G.J. Michaelson, P. McAndrew, K.G. Waugh, and W.J. Austin, "Dynamic Control and Prototyping of Parallel Algorithms for Intermediate and High–Level Vision," IEEE Computer, February 1992: p. 43–53.

[53] John A. Webb, "Latency and Bandwidth Considerations in Parallel Robotics Image Processing," Supercomputing '93, 1993, Portland, OR: IEEE Computer Society.

[54] John A. Webb, "High Performance Computing in Image Processing and Computer Vision", Proceedings of the International Conference on Pattern Recognition, October 1994, Jerusalem.

[55] John A. Webb, "Steps Toward Architecture–Independent Image Processing," IEEE Computer, February 1992: p. 21–31.

[56] C. Weems, E. Riseman, and A. Hanson, "UI Parallel Processing Benchmark," IEEE Journal of Parallel and Distributed Computing, 1988: pp. 673–688.

[57] C. Weems, E. Riseman, and A. Hanson, "The DARPA Image Understanding Benchmark for Parallel Computers," Journal of Parallel and Distributed Computing, Vol 11, 1991: pp. 1–24.

ELECTRICAL ENGINEERING

MODEL–INTEGRATED PROGRAM SYNTHESIS FOR REAL–TIME IMAGE PROCESSING

MICHAEL S. MOORE

Dissertation under the direction of Professor Janos Sztipanovitz & Dr. Gabor Karsai

Real–time imaging applications, such as robotics, military target tracking, autonomous vehicle control, and on–line video processing, require huge computational performance due to the large datasets involved. For this reason, real–time imaging systems have traditionally been built from custom hardware designed to perform specific algorithms. However, custom hardware implementations are expensive, and the real–time performance is achieved at the cost of end–user programmability and flexibility. A more generalized and flexible approach is to parallelize the computations by taking advantage of their inherent concurrency and to utilize a scalable parallel hardware architecture.

It is clear that developing a general real–time parallel image processing system involves much more than building a high performance parallel hardware architecture. Without high–level programming environments and tools designed specifically for building parallel imaging software, parallel architectures are difficult to program, and thus not cost–effective solutions.

The goal of this dissertation has been to create a flexible and easy to use image processing programming environment for generating real–time parallel software implementations. I show that, through the use of Model–Integrated Program Synthesis (MIPS), parallel implementations of image processing data flows can be synthesized from high–level graphical specifications. The complex details inherent to parallel software development become transparent, enabling the cost–effective exploitation of parallel hardware for building more flexible and powerful real–time imaging systems.

As proof of concept, I present MIRTIS (Model Integrated Real–Time Image Processing System). MIRTIS employs the Multigraph Architecture (MGA), a framework and set of tools for building MIPS systems, to generate parallel real–time image processing software which runs under the control of a parallel run–time kernel on a network of Texas Instruments TMS320C40 Digital Signal Processors. The MIRTIS system provides a *graphical model builder* for declaring (1) the computations to be performed, (2) the available hardware resources, and (3) the performance constraints of the application. A *model interpreter* automatically decomposes, scales, and allocates the computations to the available resources, then generates a real–time parallel C40 implementation.

MIRTIS is a clear example of how the MGA can be used to synthesize parallel real–time image processing systems which are cost–effective to program, configure, and scale.

Approved_____ Date_____
Janos Sztipanovits

Approved_____ Date_____
Gabor Karsai