CARNEGIE MELLON UNIVERSITY

RVMDES: A Tool for Efficient Design of Complete, High Speed, Image Processing Machines

A DISSERTATION
SUBMITTED TO THE GRADUATE SCHOOL
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
for the degree
DOCTOR OF PHILOSOPHY
in
ELECTRICAL AND COMPUTER ENGINEERING

by

RICHARD MADISON

Pittsburgh, Pennsylvania
July, 1997

Abstract

The parts inspection industry requires image processing machines operating at speeds unattainable by microprocessor systems. Parallel processors can provide higher speeds, but often the speeds required can only be achieved using custom hardware. Unfortunately, building machines fi-om custom hardware involves long and expensive design cycles. However, this time and expense can be dramatically reduced by building machines fiom reconfigurable vision systems, consisting of hardware modules that can be connected by plugging them into special motherboards. Using this modular hardware reduces design cycle time **and** cost, by practically eliminating **the** hardware design and fabrication stages, but writing and testing programs to run on modular hardware networks is still time consuming, and requires detailed knowledge of the hardware.

This thesis presents the concepts behind and implementation of a new design environment, in which users can quickly design high speed image processing machines fi-om a set of modular hardware blocks. Like design tools for multiprocessor systems, this environment allows a user to write and test block diagram algorithms, partition them onto hardware networks, simulate the timing for each module, and generate executables for any programmable hardware. Unlike previous tools, the new environment models custom hardware as well **as** processors, assists in optimizing processor software, and includes tools to design all aspects of an image processing machine, such **as** hardware configuration and user interface. The environment, called RVMDES, will be basis for a real tool to design reconfigurable vision machines fiom the modular hardware developed by the Egipt group **at** CMU.

Contents

Acknowledgments

I would like to thank Barrett Trask for implementing and testing the RVM configuration search algorithm, Bill Ross for implementing the software simulator and primitive editor, Dr. Takeo Kanade for the idea of applying the work to the Egipt RVS, and the Raven Lab for providing the equipment on which the research was done.

I. Introduction

A. High Speed Image Processing Machines

Industrial image processing applications require higher frame rates than a single microprocessor or DSP can provide. For example, Kirin Techno Systems' ink jet printing inspection system must process relatively small images (256x160) at 30 frames per second, about 20 times faster than the inspection algorithm runs on a standard DSP (Texas Instruments C40, 50 MHz), programmed by an expert. Other applications must be even faster, completely processing a fiame in a fraction of the interval before the next frame is captured. Such low latency systems are necessary when an inspection station's camera and effector must be mounted close together, because an inspected item travelling on a conveyer belt reaches the effector before the next item reaches the camera.

The reason that microprocessors and DSPs cannot provide these speeds is that they are serial machines. When implementing simple image processing functions, they repeat the same instructions for every pixel in an image. Images contain a huge number of pixels (even a 256x256 image has over 65 thousand pixels), so processor implementations of functions that process one at a time are bound to be slow. Complex hnctions that use fewer pixels often perform complicated math, again one instruction at a time, and so are also slow. While processor speed increases every year, image processing algorithms become increasingly subtle and sophisticated to match, so algorithm speeds do not increase. Thus, to achieve high image processing speeds, industrial designers have turned from single processors to custom hardware, multiprocessing and reconfigurable vision systems.

Custom hardware, such as FPGA or ASIC chips, can perform simple image processing tasks in a fiaction of the time required by a processor, and at lower costs. Unlike general purpose processors, custom hardware only includes the components it needs for its processing task. When an image processing task does not require much hardware, a processor sized chip can hold several copies of the hardware, to process several pixels at once. Alternately, it can hold additional components such that each component is only used once while processing a pixel, and processing can be pipelined. The pipeline can include I/O operations, eliminating the need for memory to buffer the input and output, further decreasing space requirements. If adding components cannot increase processing speed, for instance if speed is limited by I/O bandwidth, the reduced hardware can be implemented on a smaller, less expensive chip. For these reasons, custom hardware provides an attractive replacement for processors.

Custom hardware is made less attractive however, by the difficulties in designing it. There exist software tools to help program individual chips, but they do not allow users to design networks of chips, or to route connections between chips. The user can only determine interaction between chips by staring at waveforms generated by the chip simulators, to see whether they are all compatible. Further, the designer must spend time and money building the chips into a printed circuit board, having the board built, and testing whether his estimates of the interaction were correct. As a result, the majority of the design time is

spent on tasks not related to the algorithm. Worse still, a small change to the algorithm can require a whole new round of hand-calculated simulation of hardware interactions, and a complete redesign of the circuit board. As a final insult to hardware designers, the tools that would simplify the design process are unlikely to be created, because there are too many types of hardware, with too many communication schemes and too many packages to cover them all in one design program.

An alternative architecture for fast image processing is the multiprocessor network. In such a system, an image processing task is split over several processors, each performing a fraction of the work in a fraction of the time required by a single processor. Processors are as fast as custom hardware when tasks require a large number of components, and designed to implement functions of any complexity. Thus a system with arbitrary functionality could be built from a network of processors. Several companies provide the software tools necessary to quickly program such networks. Other companies produce processor modules that can be plugged together to quickly implement the networks.

While multiprocessor networks are thus flexible and easy to build, they are less attractive than custom hardware systems in many ways. Processor networks have higher latencies than custom hardware chips because they must transfer data between processors, which is slower than passing data within a single, custom hardware chip. Also, processors cannot pipeline their data processing and I/O stages, so they must increase their latency or frame time to transfer data. Further, it is simply not cost effective to use several processors to carry out simple operations that a single custom hardware chip could accomplish at the same speed.

One way to provide the processing speed of custom hardware along with the ease of designing and building processor networks is to use a reconfigurable vision system (RVS). An RVS consists of a set of hardware modules with a common communication interface and physical characteristics, each containing one processor or custom hardware chip. The set of hardware chips is chosen to reflect the specialized needs of computer vision and image processing algorithms. As with multiprocessor systems, a designer can quickly plug RVS modules together to build an image processing machine. Because the set of chips involved is finite, it is also possible to build a software design tool that facilitates design of the entire machine. Finally, because the RVS modules include both processors and custom hardware, the user can build machines from an optimal mix of fast custom hardware and slower but more flexible processors.

B. The Egipt RVS

The Egipt group at CMU has created a reconfigurable vision system. Because reconfigurable vision machines do not seem to be a current product or area of research, the Egipt RVS draws mainly on concepts from current modular processor systems. It then adds to these ideas, by tailoring modules to handle large images, providing more ways to connect modules, and providing non-processor modules.

Current modular processor systems typically consist of modules, each containing a pro-

cessor, I/O buffers, memory and connectors, along with special motherboards hosting slots for the modules. The motherboards power the modules and provide traces that connect the I/O ports of the modules in its slots. Additional traces connect module I/O ports to connectors for ribbon cables that transfer data between multiple boards. Modules are often built around Texas Instruments **C40 DSPs,** which have six I/O ports. These systems' motherboards have three or four slots for modules, so that each module's six ports almost fully interconnect it with the other modules on a board, and with ribbon cable ports leading to and from other boards.

The Egipt RVS is based on the Texas Instruments **C44** DSP, a smaller version of the C40. The **C44** has a reduced number of communication ports, allowing less interconnection between modules, but it can be incorporated into a smaller module, so that six modules can fit on one motherboard. **A** single board can then hold several parallel processors along with **an** input broadcaster module and output collector module, with no intervening cables to slow down data transmission. In the most common modular processor motherboards, made by Hunt Engineering, cable transfers require double the time of on-board transfers, which can cause a bottleneck when transfemng large images. The higher number of modules per board will remove this bottleneck.

In addition to the new processor, the Egipt RVS modules contain more memory, faster I/O ports, and extra connector pins. The memory is double that of the most common Hunt Engineering C40 modules, to hold twice the number of intermediate images during processing. The faster I/O ports can send bursts of data at high speeds, to transfer images between modules in short periods of time. The extra connector pins provide new ways to communicate with the custom hardware that can be built into RVS modules in place of the **C44.** For instance, there are new lines to exchange signals with A/D and D/A modules through the motherboard, rather than over separately connected cables. There is also an extra pin for each communication port, to tell whether data or commands are being passed, in case hardware only has one port for both. More pins attach each module to a common bus, which allows most modules to communicate with each other, despite how their regular communication ports are connected.

The Egipt RVS motherboards also differ fiom their processor based predecessors, to handle custom hardware **and** the larger number of modules per board. Custom hardware modules cannot always remap which ports pass which data, **so a fixed** trace between two ports in **two** slots may not suitably connect hardware modules in those slots. To compensate, Egipt motherboards use special overlay cards that map the traces ending at each slot to the ports in that slot. This allows a single trace to properly connect two slots, no matter which ports are involved in the connection. In addition to port complications, RVS motherboards must compensate for the low connectivity between their modules. Each module has only three output ports, **so** traces can only connect it to 3 of the 5 other modules on the board. It cannot even use all of these if the module must communicate with another board or send parallel data streams to a hardware module with fixed port usage. To increase the number of connection options, overlay cards can short traces across slots instead of connecting them, creating new traces. **A** third complication to connectivity is that processor ports are typically bidirectional, but must be connected to either input or output traces, not

both. If a processor has fewer than six ports, some processor ports are assigned as inputs or outputs, while others are assigned a pair of connections. The designer must set jumpers on the module to determine which of the pair is actually used.

The remaining feature of the Egipt RVS is that it contains all of the hardware necessary to build a complete reconfigurable vision machine (RVM). It includes modules or plans for modules built around a C44 processor, fiame grabber, video display, data broadcaster, data merger, FPGA, convolver, and host interface. The host interface module communicates with the RVM operator on a PC and with the other hardware modules. It boots the and test the RVM, sends new parameters to the program running on it, and receives statistics output by the program. It talks to the hardware modules over a global bus, formed by connecting each motherboard's common bus using a VME bus. The host interface is necessary for many image processing applications, including inspection, where important parameters may vary with lighting conditions and need to be tuned by a human, and teleoperation, where the RVM is only semiautonomous.

The various new features of the Egipt RVS allow it to provide performance superior to the current, processor-only, modular vision systems. However, those same features complicate the design of the Egipt RVS. Therefore, effective design of RVMs using the Egipt RVS requires a software tool that supports both the features of a multiprocessor system, and the new features of the RVS.

C. Toward a Design Tool

A design tool for the Egipt RVS must allow users to design the various aspects of a reconfigurable vision machine (RVM). First, it should let the user generate and simulate an algorithm with a minimum amount of work. The tool should also allow the user to choose hardware and partition the algorithm onto it, again with minimal work, then automatically generate the code or parameters each hardware module requires to execute its piece of the algorithm. Code should be optimized, automatically or with minimal user intervention, to take advantage of the structure of image processing functions. The tool should automatically configure hardware modules onto motherboards in a way that provides all necessary connections between modules. The tool should provide timing estimates for a design, and an easy way to identify and remedy timing bottlenecks. Finally, it should build a user interface, since it knows what data should be passed to and from the machine. Again, this should be automatic or require minimal help from the user. There is some existing work in some of these areas, but previous work addresses only a few of the requirements, and some of the requirements seem completely unexplored.

1. Programming Environments

The requirement for an easy to use algorithm programming and simulation interface is well addressed by several visual programming environments, such as Khoros [1] and AVS [2]. These environments use block diagrams to represent algorithms, so that a user can quickly assemble an algorithm without writing low level code. The block diagram also serves as an interface for simulating the algorithm, allowing the user to quickly cycle between building and testing parts of an algorithm. Visual programming environments

will even convert a block diagram into C code or an executable, so that it can run outside the environment. Visual programming environments are excellent for generating code for a microprocessor machine, but lack most of the features needed to generate high speed vision machines. For instance, they do not include any concept of hardware beyond a single microprocessor, and do not optimize their code.

Design environments for modular multiprocessor networks take over where visual programming environments leave off, allowing the user to generate hardware and partition software onto it, all in terms of block diagrams. Some environments customized around the C40 DSP [3][4] also simulate hardware, to provide the user with timing information, even modeling interactions between the multiple processors. From this, the user can identify bottlenecks and determine how to modify the design to compensate. Other environments that are not customized for a particular processor [5] cannot model timing, but can write code for several types of processors, allowing the user more flexibility in his design.

These design environments come closer to meeting the requirements for a design tool for the Egipt RVS, but they still do not solve several parts of the design problem. They only allow the user to design with and simulate processor modules. Their simulation is even of dubious value, since the Texas Instruments C40 emulator they use does not account for cycles spent waiting for pipeline conflicts to clear, which can sometimes account for half of a C40's time. The environments are not affiliated with any particular modular hardware system, so they do not provide the user with a hardware configuration, check that designs can be implemented with a modular system, or generate an interface to the system. They also do not attempt to optimize code to account of the predictable structure of the image processing functions they implement.

2. Optimizations for Image Processing Software

There is a reasonable amount of work on optimizing image processing programs, particularly for silicon compilers. Unfortunately, none of it looks at the high level optimizations available when the functionality of programs is restricted to image processing.

One line of research focuses on optimizing code or facilitating code writing, for image processing and other DSP applications. Some papers [6][7][8] consider how to optimize assembly code or tree representations of equations to be translated into optimal assembly code. Others recommend generic optimizations such as expanding function calls or constants in-line, or removing loop independent calculations from loops [9][10][11]. These optimizations are very low level and can be performed by optimizing compiler, used to compile C code from a visual programming environment. The Texas Instruments C40 compiler, typically used to compile code for C40 and C44 DSPs, handles these optimizations.

A second set of work [12][13][14] analyzes image processing functions to distill a set of primitive structures, fimctions and functionals needed in an image processing programming language. A language with these primitives would allow a user to more explicitly represent his intentions in his code, making the code easier to write. Presumably, a compiler could also pattern match against the new primitives, recognizing opportunities to

optimize them that it could not have seen by looking at their translations. Unfortunately, the primitives are only useful for the proposed RVM design tool in a limited way. The block diagrams of Visual programming environments provide an easier way to write algorithms than the text based, image processing languages, and already include the recommended primitives. A specialized language could be used for function definitions, but if blocks represent image processing primitives, their definitions should not use other primitives, so using a language with these primitives avails nothing. What is useful about this research is the implication that image processing functions should be written in a format specifically designed to help a compiler recognize the special optimizations the apply to them.

A related set of work consists of the two languages Apply and Adapt [15]. These languages are based on the realization that many image processing functions apply some operator to each pixel or window in an image. A user writes programs by coding image processing function kernels and specifying the images they should be applied across. A compiler then decides the optimal way to split the processing over an array of parallel processors. The problem is that splitting a function over parallel processors is of limited value in an RVM, where low interprocessor connectivity makes fanning data out to a large array of processors a hardware intensive, high-latency operation. RVMs are better optimized by pipelining successive functions across several hardware modules. Thus, an RVM design tool should look for optimizations that take advantage of the relationship between functions partitioned onto each module, not try to split individual functions over parallel modules. Thus the actual optimizations made possible by Apply and Adapt are not directly useful for the RVM design tool. However, two languages further the notion that image processing functions should be coded in a format that leverages their special structure to help a compiler recognize opportunities for optimization. And they reveal that this special structure consists, at least in part, of a loop applying a function kernel to each pixel of an image.

3. Missing Areas

The work mentioned above explores the areas of algorithm creation and testing, partitioning of an algorithm onto hardware, code generation, and to some extent software optimization. However, none of the work attempts to cover all of these areas. Further, none of them create actual hardware designs to implement hardware networks, apply high level optimizations to their software, generate an interface for their networks, or provide realistic timing estimates, especially for heterogeneous hardware.

D. RVMDES

This thesis presents the concepts behind and implementation of RVMDES, an integrated design environment allowing efficient design of complete, optimized, high speed image processing machines using Egipt RVS modules and motherboards. The environment is similar to multiprocessor network design environments, providing tools for algorithm design, simulation, partitioning, and code generation. However it adds tools for configuring RVM motherboards, simulating the timing of a heterogeneous hardware system, applying optimizations tailored to image processing programs, and creating programs to

interface with the RVM.

The chapters of the thesis explain the components of the tool. The thesis begins with an account of how RVMDES leverages the properties of image processing functions to facilitate the design of an algorithm and then the hardware to implement it. Next, it shows how RVMDES uses those same properties to automate or facilitate software optimization, code generation, hardware configuration and interface generation. Throughout, it follows the progress of an example application as each design step is applied. The thesis concludes with a complete example of using RVMDES to build as second application, and a discussion of the contribution of the work.

11. Algorithm Design in RVMDES

The first step in designing an RVM is to design the algorithm it will implement. This can involve a lot of design-and-test iterations, because design specifications often contain input-output pairs, not algorithm ideas. The design process thus involves inventing and coding a possible algorithm, then iteratively testing and modifying it until produces the proper set of outputs, or must be scrapped in favor of a completely different algorithm. This iteration can be time consuming, because modifications to the initial algorithm are often extensive, and may involve starting from scratch several times.

To reduce the time required to design an algorithm, RVMDES provides a user interface that makes each design step as efficient as possible. The interface represents algorithms as block diagrams, which are easy to build, simulate and modify. They are also easy to interpret, in case the user forgets what part of the algorithm does as he works on another part, or the algorithm must be reused later. Also, block diagrams provide an excellent structure around which to generate and optimize code, and to map algorithms onto hardware. Block diagrams do not provide the full flexibility of text based languages, but they can be specifically designed to represent all of the structures needed in a limited domain like image processing. This chapter explains how RVMDES block diagrams can be used to describe image processing algorithms, and how the RVMDES software editor facilitates editing and simulation of block diagram algorithms.

A. The RVMDES block diagram

The RVMDES block diagram representation provides blocks for the functions, inputs, outputs, and decision points in an image processing algorithm. All blocks are named for easy identification, and are marked with ports to represent the inputs they require and the outputs they produce. Connections between these ports show how data flows through the blocks. Input ports each read data from one source and thus end one connection, but output ports can each begin multiple connections, providing their data to several blocks. A diagram built from these blocks shows one *frame* of an algorithm, in which one set of inputs is processed to produce one corresponding set of outputs.

Two syntax rules constrain how the blocks are connected, to ensure that RVMDES will be able to simulate an algorithm and turn it into an RVM. The first rule is that a block's input ports must all connect to other blocks' output ports. A block cannot execute until valid data is available to each of its input ports. This data is read through connections to other blocks, and is made valid when those blocks execute. If an input port is unconnected, valid data is never provided, and the block can never run. Inclusion of an unusable module either represents a user oversight or an *unnecessary* obstacle to program readability, so RVMDES disallows it. The second rule is that an output port of a block may not begin a chain of connections and function blocks that ends as an input to the original block, forming a loop. Such a loop would present a paradox wherein a block cannot execute until its inputs are valid, but it must execute to start the chain of blocks that produces those inputs. A meaningful order of execution cannot be constructed for these chains, so RVMDES disallows them. If an algorithm requires a loop, a legal one can be built with the help of

branch and join modules, as explained later.

The block diagram representation was chosen for RVMDES because it makes steps of software design efficient. Block diagrams are easy to program with, because they allow the user to think directly in terms of the algorithm components and data flow, not in terms of the text and programming language idioms that implement it. This saves the user time not only because he does not have to write functions definitions, but also because he does not have to debug them or switch between thinking about the algorithm and thinking about individual functions. Block diagrams are also easy to understand, because the set of functions and the flow of data between them are explicit, not embedded in text. This allows a new user to quickly understand and reuse existing code, and it allows a current user to keep track of what his algorithm is doing. Block diagrams provide a straightforward interface for simulation, allowing a user to step through the diagram, simulating each block. This simulation can be completely integrated into the block diagram editor, so that blocks can be replaced and re-simulated until they produce the desired results, rather than requiring a new simulation with every change to the algorithm. Finally, block diagrams are easily partitioned onto hardware, by assigning each block to a hardware module, and blocks are easy to associate with hardware and/or code that implements them.

B. Block diagram blocks

RVMDES provides several types of blocks from which to build block diagrams. These blocks are not sufficient to implement some programming constructs, such as recursion, but are specifically chosen to be able to represent the constructs required for most image processing programs. This section explains the types of blocks provided by RVMDES, as summarized in table 1.

Block Type	Purpose
Function	Image processing task
Input	Data read at beginning of frame
output	Results written at end of frame
Constant	Data already present at beginning of frame
Branch	Decision point, allowing multiple execution paths
Join	Merge point for multiple execution paths
Wraparound	Way of holding data between frames
Undefined	Place holder for an unavailable function block

Function blocks are the heart of an algorithm, used to perform image processing tasks. RVMDES provides a library of image processing functions. When a user creates a function block, he must specify which library function it will represent. The definition of the

associated function tells RVMDES which executable the block will call during simulation and what code or custom hardware can implement the block in an RVM. The definition also specifies the number and meaning of the function block's input and output ports. As with all module types, RVMDES records and allows the user to change the name of a function block and data types of its output ports.

Input blocks represent data that is read at the start of a fi-ame, either from a file during simulation, or from a camera or other sensor in the RVM. Each block has one output port, from which other modules can read this data. RVMDES records the data type and dimensions of the data represented by each input block, so that the algorithm simulator and code generator know how much memory to allocate and fill. RVMDES also propagates the dimensions through the block's output connections to other modules. In general, the user must set the dimensions, although RVMDES provides the option to read the dimensions as inputs to the input block. This allows the user to equate the dimensions of **two** input blocks before those dimensions are known. Input and constant blocks must have their dimensions set, so that dimensions of data can be propagated through the program.

Output blocks are the simplest type, representing the endpoints of **an** algorithm. Each reads a single data item **from** its single input port, and writes it to a file in simulation, or to an output device such **as** a monitor in the real application.

Constant blocks represent inputs to the algorithm that are not loaded at the beginning of each frame. Examples are convolution masks and parameters that evolve over several frames. Like input blocks, constant blocks have data type and dimension parameters to identify how much memory they require, can read their dimensions **as** inputs, and provide a single output port. Unlike input blocks, constant blocks come in three flavors -- initialized, uninitialized, and initializing. Initialized constant blocks contain user defined data. They are useful for small items like scalars, convolution masks and the initial values of evolving parameters. Uninitialized constant blocks have no initial values, and simply reserve memory. They are sufficient for accumulators that must be initialized each frame, such **as** Hough transform planes, and do not bloat executables with unneeded initialization values. Initializing constant blocks read their values at **run** time, before the first frame is processed, and can be reset by the **RVM** operator between frames. This is useful for parameters that cannot be determined before the RVM runs, such **as** thresholds that vary with lighting conditions.

Branch blocks provide flow control for block diagram programs, allowing loops and other fancy control structures. To this end, each branch has a bundle of input ports, several identical bundles of output ports, and a test associated with each output bundle. These tests are equalities and inequalities like those found **in** 'if statements in C code, except the final test which always evaluates true. More complicated tests like those involving looping operations can be implemented by cascading branches. When executed, a branch block performs its tests in sequence, until one evaluates to true. It then passes its input data to the output bundle associated with the successful test. Because a block can only execute once its inputs have been calculated, only blocks attached to the successful bundle of output ports will be able to execute. The number of output bundles and number of ports

per bundles is variable, allowing the user to choose the number of conditional paths. Branch blocks must have all of their tests defined, and must have at least one input port and one output port, to insure that the branch can execute, and that execution will continue after it.

Join modules are the counterpart to branches, with several bundles of input ports funneling to one bundle of output ports. Once all inputs on a single bundle are valid, the join can be executed, passing the active bundle's data to its output ports. They are used in conjunction with branch blocks, the branch splitting execution over multiple paths, and the join recombining those paths. Like branch modules, they have a user-controlled number of bundles and ports per bundle. **Join** blocks also must have at least one input and one output port, and require that corresponding ports of each input bundle connect to output ports with the same dimensions and data type. The latter is necessary to insure that no matter which bundle receives data, the output bundle will always provide the same kind of data.

Wraparound blocks store data between passes through the algorithm, to implement adaptive parameters or tapped delays. They take **two** inputs -- a constant that reserves space for their data, and another input that provides the data. When a wraparound block executes during a frame, it passes its constant input on to its output. Once the fi-ame has finished, the data from the second input is transferred into the space held by the first. It will then be the data transferred to the wraparound block's output in the next fi-ame. To work properly, wraparound modules require that their first input be a constant, and that no other blocks read that constant directly.

Undefined blocks represent uncertain steps in **an** algorithm. They are useful in top down algorithm design. When the user knows what functionality is required at a certain point in **an** algorithm, but cannot find a suitable function in the library or is unwilling to search the library and lose his train of thought, he can use an undefined module **as** a place holder. Once the top level design is done, he can convert each undefined block into a function block, after locating the proper library function or defining a new one. Alternately, the user can delete **an** undefined block and replace it with **a** block diagram. Undefined blocks have a variable, user-controlled number **of** ports, so that all relevant connections can be made in the top level algorithm. Undefined blocks are not allowed in a completed program, because they have no associated definition, and cannot be simulated or translated into code.

C. Representing control structures

The blocks in an RVMDES block diagram are connected into programs that are a hybrid between a data flow diagram and a flow chart. Like data flow diagrams, they explicitly show how individual variables flow through the steps of **an** algorithm, making the functionality of the algorithm more obvious. Unlike data flow diagrams, they must include some description of control flow, to allow the loops and conditional executions required in image processing algorithms. This means that the graphs must take on some of the character of a flow chart. To make data flow and control flow explicit in the same graph and ensure that it translates into a valid program, certain rules must be imposed on the struc-

ture of the graph. This section explains in more detail the advantages and limitations of the data flow diagram and flow chart, how a mixed representation can retain the benefits of each, and what rules must be imposed on that representation.

1. Data flow diagrams and flow charts

The data flow diagram, as depicted in figure 2.1, uses boxes to represent functions, connected to show how data flows between them. This allows a viewer to quickly see what operations are applied to what data, and thus to quickly understand the algorithm. While the representation does not provide an explicit execution order for its blocks, the order is tightly constrained, because each block executes once, after blocks that create its inputs and before blocks that use its outputs. Beyond that, the order is arbitrary. Thus, to convert a data flow representation to an executable, the blocks can be listed in arbitrary order, rearranged to fit the constraints, producing a valid execution order, then translated one at a time to into machine language, or a compilable programming language.

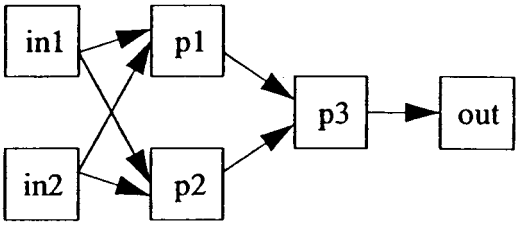


***figure** 2.1 - data flow diagram*

This representation works when each block is used exactly once, but many computer vision applications do not have this simple linear structure. Often programs must iterate until some data dependent condition is satisfied, or must execute only one of a set of functions, again depending on the data. The data flow representation cannot handle these structures, because it cannot specify that only one of several output connections receive output data, to implement a conditional branch, and cannot specify that only one of several input arcs should be read, to close a loop. Further, the simple determination of execution order would not work, because the blocks no longer execute once each.

A second type of block diagram is the flow chart, shown in figure 2.2. In this representation, connections show order of execution of the blocks, not the data flow between them. This allows the inclusion of decision points, represented by diamonds, which can direct program execution along one of several paths. These paths can then be rejoined at nodes, represented by small circles, to implement the loops and parallel paths that computer vision algorithms require. Flow charts can be converted into programs written as code segments representing sequences of connected blocks, glued together by conditional branch statements representing the decision points. Unfortunately, the functionality of an algorithm represented by a flow chart can be difficult to understand, because while all of

the operators are shown graphically, there is no indication which data they operate on.

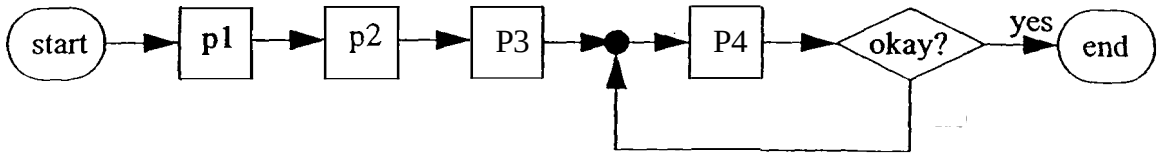


figure 2.2 - flow chart

2. Mixed representation

The **RVMDES** block diagram representation is a compromise between the flow chart and data flow diagram, retaining the flow chart's ability to control which blocks are executed, but at the same time making data flow explicit. The compromise stems **from** two observations about how flow charts can be modified to be more like data flow diagrams without impairing **their** ability to implement control structures.

The first observation is that flow chart connections can be thought of as bundles of connections carrying all of an algorithm's active data. By making this replacement graphical, the diagram can show the existence of separate data items without affecting order of execution. Further, a flow chart's start and ***end*** blocks can be replaced with input, constant and output blocks to terminate the various connections, so that at least those variables will be identifiable. Again this does not hurt the flow chart, because the clusters of inputs and outputs still clearly represent start and end.

The second observation is that a sequence of processing blocks bounded by decision points (diamonds and circles) can be converted to a data flow diagram, reading its inputs from a bundle leaving **the** decision point preceding it and writing its outputs to a bundle entering the decision point ending it. Because the sequence contains no decision points, it can be reassembled from a data flow diagram by constructing **an** execution order **as** discussed earlier. When the sequence begins or ends **with** the clusters of input and output blocks, they can simply be added into the data flow diagram. The resulting graph is a set of data flow diagrams, connected by decision points.

For consistency, the diamonds and circles are replaced with branch and join blocks, completing the transformation to an **RVMDES** block diagram. The **RVMDES** block diagram in figure 2.3 shows the advantages **of this** transformation. Unlike the equivalent flow chart in figure 2.2, the RVMDES block diagram shows the user what the first half of the algorithm is actually doing. And unlike the data flow diagram in figure 2.1, the RVMDES

block diagram is able to implement the second half of the algorithm.

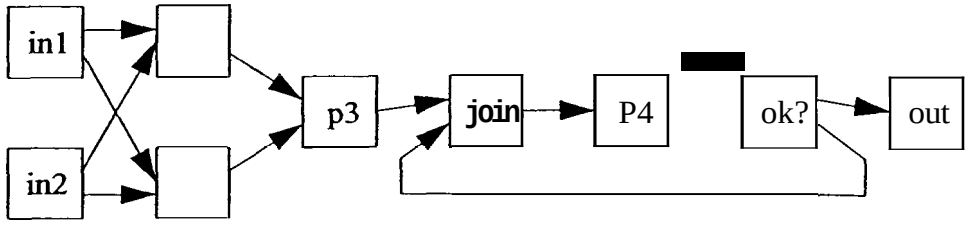


figure 2.3 - RVMDES block diagram

The only problem with the representation is that it looks like a data flow diagram, but is not one. The blocks must be connected in such a way that the diagram can be converted back into a flow chart, with its explicit execution order. Thus additional rules must be imposed on block diagrams to insure that they are properly connected.

3. Separate flows

The first rule binding proper RVMDES block diagrams is that they must be divisible into flows, as in the block diagram in figure 2.4. Each flow consists of a set of blocks, containing no branches or joins, reading all inputs from one output bundle of a branch or join block, and writing all outputs to one bundle of another branch or join block. Because each flow is actually a data flow diagram, an execution order can be constructed for each, as discussed earlier. The flows can then be glued together to form an executable program, using jumps and conditional jumps to represent join and branch blocks. In a properly constructed block diagram, RVMDES can even determine which blocks belong to which flow, by considering each input bundle of each branch and join module, following connections back to a single bundle, and recording any blocks visited along the way

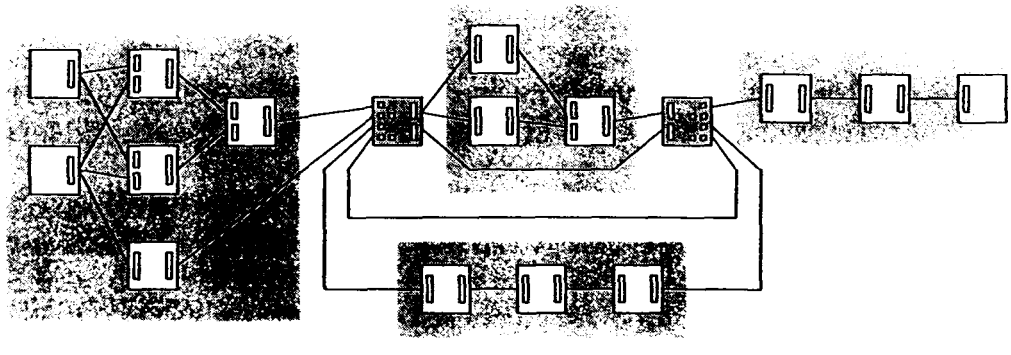


figure 2.4: RVMDES block diagram, consisting of flows (shown as shaded rectangles), separated by branch and join blocks (dark blocks, with many ports)

Actually, it is even possible to extract flows from an improperly constructed diagram, like the one in figure 2.5a. The function block circumventing the branch block (shaded) obviously belongs with the two modules that depend on it. Thus, it could simply be grouped with those modules, and an execution order could still be constructed. On the other hand, it could also be moved to the far side of the branch module as in the equivalent block diagram in figure 2.5b. This breaks the diagram into logical sections, and makes it readily

apparent which modules belong to which sections, what data is being passed between sections, where as the user must work to parse figure 2.5a into sections. Because the entire point of the data flow diagram is to make a program quickly and easily understandable for the user, and because the rule requiring separation of flows helps insure this, it is enforced, even though RVMDES could write programs without it.

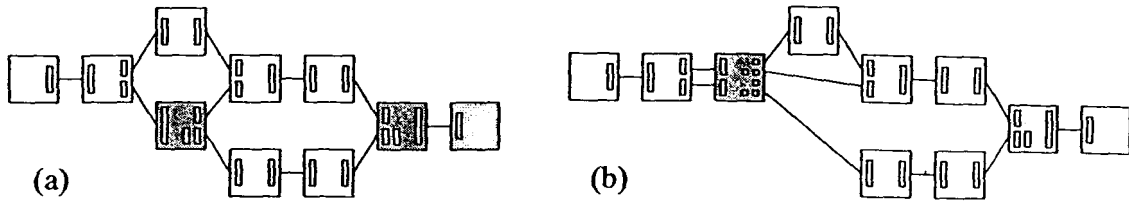


figure 2.5 - (a) invalid block diagram withfunction circumventing branch and (b) the valid version

4. Input and output flows

The second rule binding RVMDES block diagrams is that among its flows, only one may contain input blocks, and only one may contain output blocks. In addition, the input flow may not read from branch orjoin bundles, nor may the output flow write to branch orjoin bundles. Again, this feature is not strictly necessary, but it facilitates code generation and program understanding.

In deriving the RVMDES block diagram, the starting point for a program was replaced with the set of all input modules. Working backwards, the code generator can determine which flow to execute first by locating the flow containing input modules. This restriction is somewhat limiting, because by spreading inputs throughout an algorithm, the implementation could read only the ones it needs, and only as they are needed. In a parallel processing implementation, this may allow a frame to begin earlier, before all inputs are ready. However, in a reconfigurable vision machine, each processor or piece of hardware is expected to read certain inputs at certain times. The only way to be sure that this happens is to read inputs as the first step of the algorithm.

The idea of requiring one output flow also follows from the derivation of the block diagram, but is not strictly necessary and is only implemented to help the user. If an algorithm had several output flows, each would have to have the same number of output blocks, with the same set of data types and dimensions, to ensure that the hardware running the algorithm always output the same type of data. If the outputs have the same data type and dimension, they could be funneled through ajoin module to produce one flow, without much difficulty. This would enhance readability and allow RVMDES to easily order output blocks to check for matching data type and dimensions and to ensure that corresponding blocks are sent in the proper order. Therefore, the RVMDES block diagram specification requires such a reduction to one flow, even though it could write code without it.

D. Two example diagrams

Given the preceding explanation of the types of blocks and their connectivity, it is now possible to examine a few example diagrams, to see that they are easily interpreted. The first example is a motion differencing algorithm, shown in figure 2.6. **An** input block (yellow) reads an image at the beginning of each fiame. The image is then processed by **an** undefined function *PP,* embedded in a loop between branch and a join blocks (all grey). The loop repeats until the results of *PP* meet some condition, probably implementing some kind of smoothing or thinning. The output of this loop flows to a function module (white), where it is subtracted from the output of a wraparound block (dark grey). The wraparound serves as a delay, providing the loop output from the previous frame, or data fiom a constant block (green) during the first frame. Thus the, output of the function block is a difference image. This image feeds an output block (pink), which writes the image to a file or output device, ending the frame. A dotted line feeding the second input of the wraparound block reminds the reader that the data is read into the wraparound's buffer after the fiame has ended, and so does not affect the frame in which it was generated.

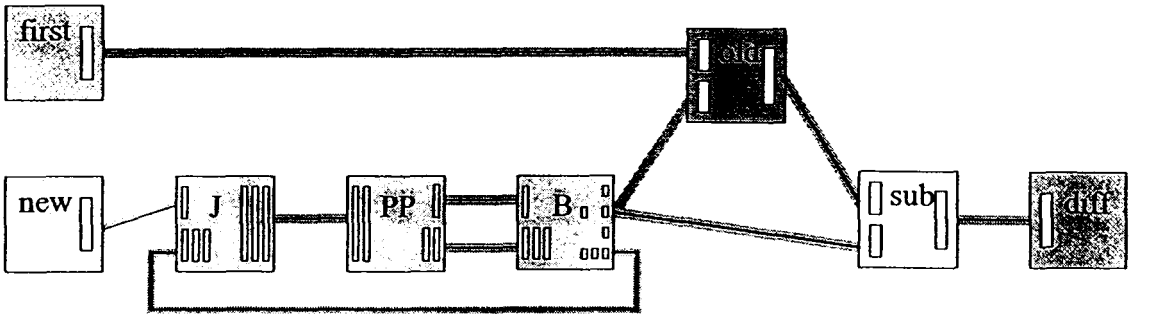


figure 2.6 - a block diagram

A more complex example, shown in figure 2.7, will be used to demonstrate the various capabilities of RVMDES throughout the thesis. Despite its complexity, the block diagram structure allows a user to quickly understand its function. As the algorithm begins, **two** constants are used to generate a **mask** for a convolution. The algorithm then thresholds its single input and finds the moment and a convolved version of the thresheld image. A branch block passes the convolved image and a mystery parameter to one of **two** parallel paths. The moment input to'the branch is not passed through, so presumably it is used to decide which path to take, indicating that the program is intended to recognize **two** types of **objects.** In each path leading fiom the branch, the convolved image is processed by some function IA or *IB*, then passed through a join block to **an** output. Each path also sends a constant value through the join block to an output, showing which path was traversed. The mystery parameter sent through the branch block is incremented along either path, then output and sent to a wraparound block to be used in the next fiame. Apparently, it will record the number of frames processed, plus the initial value in the constant labelled *p0*. From this short analysis, the user can determine that the algorithm distin-

guishes two kinds of objects, processes them differently, then outputs which type it found, the result of the processing, and the number of objects processed so far.

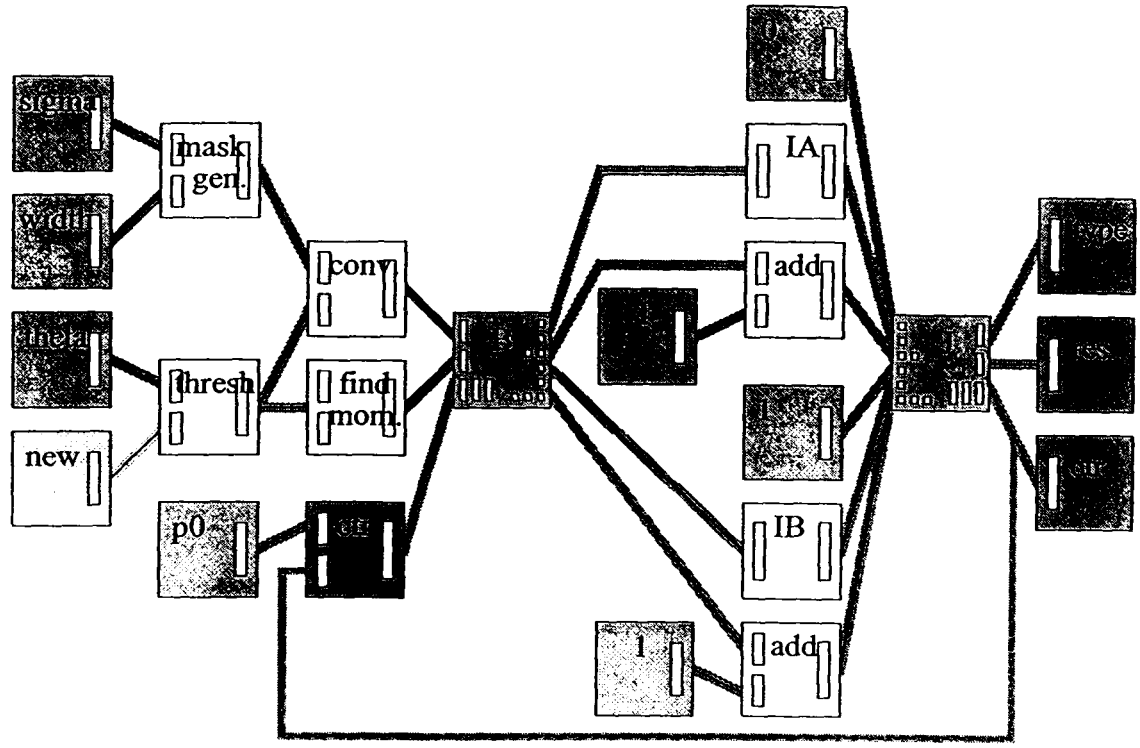


fig 2.7 - a larger block diagram

E. Editing algorithms in RVMDES

RVMDES supports the easy-to-interpretblock diagram algorithm representation by providing an editor that facilitates building and modifying the diagrams. This section explains how the editor works. Specifically, it covers how the user manipulates blocks and connections to build a block diagram, and how he adjusts the special parameters associated with each block.

1. Manipulating modules and connections

The editor, shown in figure 2.8, consists of a module generator (left) and an editor window (right). Buttons in the module generator (left) create instances of the 8 kinds of blocks, which can be dropped onto the editor window's canvas, moved around, and deleted when they are no longer needed. The various kinds of blocks are color coded as in the preceding examples, so the user can quickly search a program for a particular kind of block. In addition, the blocks are named so that the user can quickly see and recall what each is doing. Function blocks are associated with specific library functions, chosen fiom a dialog box that appears when the user creates a new function block. To make the choice easy, the dialog box hashes functions by categories such as thresholding and morphology, and allows the user to select a category, then a function fiom the limited list associated with

that category. These features make it easy to create and work with blocks.

Blocks are connected through their input and output ports, shown respectively along the left and right side of modules. Clicking on a port begins a new connection there, and clicking on a second port ends the connection. If the user clicks in the center of a block or creates a new block while a connection is in progress, RVMDES will try to finish the connection at that block. If the block only has one port, it will finish the connection at that port. Otherwise, it will pop up a list of ports and their associated parameters, allowing the user to choose the ending port by name.

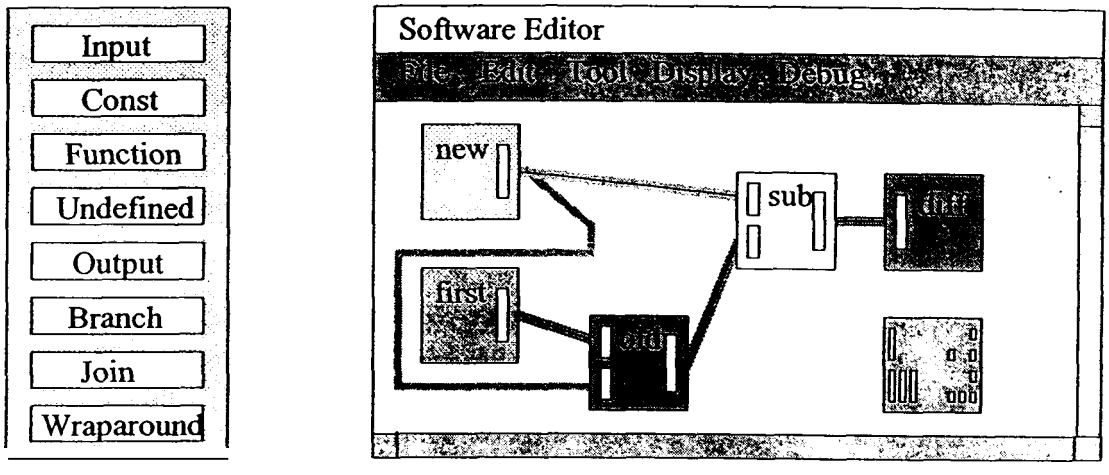


figure 2.8: module generator and main software editor

Branch, join and undefined blocks have extra ports, as shown on the branch module in the bottom right corner of figure 2.8. The outer columns of ports represent inputs and outputs as with all blocks. The inner columns mark boundaries between bundles of ports, and also can be used to connect all ports in a bundle at once. This reduces work when entire bundles must be connected between branch and join blocks. The remaining two columns allow the user to create new ports. For a branch block, adding an input port adds a corresponding port to each output bundle, and adding an output actually adds a new bundle. Joins are exactly opposite. The ability to add extra ports from the block icon is provided because the need for an extra port is often not discovered as the user attempts to finish a connection at a branch or join block that has no free ports. Being able to add a new port at that time allows him to finish the connection, rather than aborting it and starting a new one.

Once connections are made, they are color coded according to the type of block that produces them and the data type of the data they represent. This helps the user follow one of a ribbon of parallel connections, and provides a quick way to check data types. The second input to a wraparound is even shown as a dotted connection to remind the user that it is not used during regular execution. To further enhance program readability, connections can be broken into segments, such as the dotted connection along the left of the editor in figure 2.8. This allows the connections to be routed around other modules and connec-

tions, so the entire connection is visible, and does not cross other connections.

2. Specification windows

To provide information beyond the existence of blocks and connections between them, RVMDES provides a specification window associated with each block. From this window, users can set parameters, add and delete ports and view or create function definitions.

The principal purpose of specification windows is to let the user set the special parameters associated with each block. For instance, the user can change a block's name from its default (the function name for a function block, and null for any other blocks) to something mnemonic. Also, he can change the data type of any of the block's output ports, specify the flavor and define the data of a constant block, or set the names of files that input, output and initializing constant blocks use during simulation. Finally, he can set the dimensions of input and constant blocks. The dimensions of other types of blocks are not considered parameters. Instead, they are set automatically based on the dimensions of the blocks fi-om which they read, to save the user time and prevent him from inserting errors into the program.

The second purpose of the specification window is to provide an interface for the user to view and change a block's ports. For most modules, this means viewing the mapping between port names and numbers prior to deciding which port should begin a connection, or adding special output ports to write the dimensions of any original output port. The latter can be useful when an output is a list whose length is not known until the list is created, but is needed for a later function. In addition, the user can create special input ports for input or constant blocks, to read dimensions from another block's dimension output ports, rather than having them set in the specification window. This allows two input or constant blocks to match dimensions even if the dimensions are unknown, such as when defining a new function with two inputs. One more set of ports that the user can change are those belonging to branch, join and undefined blocks. Each of these blocks has a variable number of input and output ports, and the user can add, reorder and delete both them, as necessary to attach, remove or untangle connections that must go to or through the block.

A final pair of tasks which can be accomplished from specification windows is viewing the function definitions associated with function blocks and converting undefined blocks into function blocks. Functions can be primitives (defined by C code, which RVMDES uses to translate the algorithm into an executable) or macros (block diagrams composed of primitives and other macros), and both types can be viewed by calling a definition window from a function module's specification window. Viewing a primitive definition will invoke a code editor to show the various parts of the definition. Viewing a macro definition will invoke a software editor to show the macro's block diagram. If a function is a macro, its specification window will also contain a button to expand the block, replacing it with its block diagram in the software editor containing the macro. Like function blocks, undefined blocks may be defined as primitives or macros. However, the user must provide these definitions. Buttons in an undefined block's specification window allow the user to invoke a code editor or a block diagram editor, and when the user saves the defini-

tion, the undefined block will be replaced with a function block representing that definition.

F. Simulation

Once the user builds an algorithm, he will generally simulate it to see whether it generates the desired results, and to track down problems and modify the algorithm when it does not. He will then iteratively modify and simulate the algorithm until it performs correctly. To make this iteration as painless as possible, the RVMDES algorithm simulator is embedded in the block diagram editor, allowing the user to simulate using the block diagram itself as the interface. The simulator does not require an algorithm to be compiled, so the user can simulate parts of an algorithm while others are still under construction and contain "errors", such as undefined modules. This section explains how the embedded simulator interface and the simulator itself work.

1. The simulator interface

RVMDES's embedded simulator provides an interface much like a debugger for a standard programming language, where the user steps through lines of source code, executing them and viewing the data they produce. However, the RVMDES simulator differs from source code debuggers in three respects, deriving advantages from each.

First, the user clicks on blocks and connections, not text statements. Clicking through blocks allows him to think in terms of data flow, not in terms of C code implementation. The connections show him quickly which functions use which data, so he need not search for instances of a variable name in source code. The block diagram also hides the code that repeats the algorithm on each new frame, reducing the complexity of the program a little. If the user needs to simulate several consecutive frames to see whether wraparound blocks work, he can use the next-frame command, which switches the inputs to wraparounds, and attempts to update file names for input and output blocks so they can read and write sequences of data files. If the user wants to change the values of initializing constants between frames, he can manually change the file name from which the data is read.

A second difference in RVMDES's simulator is that the order of execution of the blocks in the block diagram might not be fixed. It is constrained because a block may only execute when it has valid data on all input connections, but this often provides multiple executable blocks at any given time. To guide the user through the simulation, RVMDES determines which blocks can be executed at any given time, and highlights them so the user knows which he can execute. RVMDES also tracks which connections hold data that has been written but not read, and highlights them so that the user knows which connections are worth viewing. Data on unhighlighted connections *can* still be viewed, but may be invalid if looping in the algorithm has caused upstream blocks to be reexecuted, and the changes have not propagated to the unhighlighted connection.

The third difference in RVMDES's simulator is that it does not require algorithms to be compiled before simulation. It uses compiled executables to simulate function blocks, but

this is transparent to the user, who steps through the uncompiled, high level algorithm. Because the algorithm is interpreted, the user can change it without leaving the simulation. If a block produces bad results, he can delete it, replace it with a new block, and continue simulation with it. Data that does not depend on the replaced block will remain valid, so unless the block is part of a loop, the user can repeatedly replace a module and simulate it until a desired output is produced. He can even build an entire algorithm this way, adding and testing one block at a time.

2. How the simulator simulates

How **RVMDES** simulates a given block depends the type of that block. An input or constant block reads data fi-om a file or memory, and makes it available on its output port, and any connections beginning at its output port. A function block invokes an executable associated with the particular function it represents, reading data from the module's input connections, and storing results on its output connections. Branch blocks perform their series of tests, and pass data fi-om their input bundle to the first output bundle to pass its test. Join blocks similarly pass data fi-om a single activated input bundle to their single output bundle, while wraparounds just pass their first input on to their output. Undefined blocks cannot be simulated, because they are simply place holders. Attempting to simulate them will return an error message.

3. Error Handling

While it is wise to remove errors such as invalid connections from an algorithm before simulating it, sometimes it is helpful to simulate a program that is still under construction. This is the case in top down design, where an algorithm is built from undefined blocks, which RVMDES considers syntax errors, arid the user would like to simulate each block as it is defined, even though other blocks remain undefined. Thus, while RVMDES provides an error checker to locate errors in an algorithm, it does not require the user to run the checker prior to simulating an algorithm. This is not a problem, because the simulator interprets the algorithm and does not require a particular piece to be syntactically correct until it attempts to simulate that piece.

To allow simulation of algorithms with possible errors, RVMDES checks each block before simulating it. If the block violates any of the simple syntax rules constraining RVMDES block diagrams, the simulator complains and refuses to simulate the block. Further, RVMDES completely ignores the more subtle errors involving conditional execution. This is because they exist only to help **RVMDES** determine an execution order and to make the code readable. During simulation, the user determines the execution order, and any code readability is not critical at every phase of algorithm construction. Readability will be ensured by the automatic error check prior to generating the **RVM.**

G. Summary

RVMDES provides a block diagram representation for computer vision algorithms that makes the exact function of an algorithm very explicit, without sacrificing the ability to implement necessary control structures such as data dependent loops. RVMDES also provides a software editor, from which a user can quickly and easily build block diagram

algorithms, then iteratively simulate and modify them until they produce the desired results. The time spent in this iteration is minimized by allowing the user to simulate algorithms at the block level, to iteratively edit and simulate single blocks, and to simulate pieces of a program that is still under construction. A library of computer vision functions provides predefined blocks, so that the user can think entirely in terms of algorithm building blocks, not the code to implement them. These features ensure that most of the time a user spends in the software phase of the RVM design cycle is spent evaluating algorithms, not writing and debugging code or running simulations. All of this is made possible because a block diagram can completely describe the limited set of functionality needed to implement image processing programs.

III. Streamlined Hardware Design Cycle

Once the user has designed an error free RVM algorithm, he must design the hardware to implement it. The hardware design cycle follows the same steps as software design: creating an initial design, then repeatedly testing it and modifying it until it provides the desired results, Creating an initial hardware design for an RVS means choosing a set of hardware modules, choosing which parts of the algorithm will run on each module, configuring motherboards to hold the modules, and generating code for any processor modules. Testing the design involves simulating the processing on hardware modules and interactions between them, determining whether the whole system runs fast enough to meet design specifications, and if not, determining why not. Modifying the design means changing the software, hardware, or partitioning of software onto hardware, to eliminate processing bottlenecks.

The design cycle may involve many iterations of testing and modifying, because the choice of modifications involves trial and error, and because only small amounts of hardware are added every iteration. The latter ensures that each bottleneck is remedied with a minimum amount of hardware, thereby keeping costs down. To speed the user through the design cycle despite the large number of iterations, RVMDES provides tools that make each step efficient, by taking advantage of two things: the fact that image processing applications require on a limited set of hardware, and the highly structured block diagram representation developed for the software editor. By limiting itself to the set of image processing hardware modules in the Egipt RVS, RVMDES can include enough information to simulate each type of hardware and the interactions between them, so that it can automatically estimate timing for the entire system. In addition, the RVS provides a standard module interface and set of motherboards, so transforming a network of hardware into a final hardware design reduces to a bin packing problem, which can be automated. While the actual choice of hardware and partitioning cannot be automated, **RVMDES's** block diagram representation for software provides a simple interface from which to quickly perform those steps. This chapter explains how exactly RVMDES leverages its RVS and block diagram representation to make each step in the design cycle as fast as possible.

A. Creating an initial design

The initial step of hardware design is to guess what hardware will be needed to run an algorithm in the time required by the RVM design specification. RVMDES chooses the minimum amount necessary to run the algorithm. While it is unlikely that this minimal hardware will run fast enough, it has two advantages. First, it does not waste expensive hardware remedying imagined processing bottlenecks. Instead, the user is forced to examine the system timing and see the actual bottlenecks before committing any hardware to fix them. Secund, the minimum hardware set can be generated semi-automatically, saving the user time and effort.

Once the hardware set is chosen, it must be converted into a hardware design, by partitioning software onto the hardware modules, generating code for any processor modules, and configuring the modules onto motherboards. Again, these steps can all be done semiauto-

matically, saving the user time and effort. This section explains how RVMDES performs these steps, in particular how it chooses the minimum hardware set and partition, how the user must help, and how RVMDES off-loads most of the work of configuring motherboards fiom the user.

1. Minimum hardware set and partition

A minimum hardware configuration for an Egipt RVM has four parts: interface, processing, input and output. The interface consists of a single host interface module, which must be present in any Egipt RVM, so that the RVM can communicate with its operator. The processing part of a minimal configuration is generally a single C44 processor module. All software blocks except inputs and outputs are partitioned onto this processor, so that when timing is generated for the system, it will be timing for the algorithm, uncluttered by delays from passing data between hardware. There are only two cases where a single C44 is not the processing element: when the algorithm contains only inputs and outputs, so processing is unneeded, and when the algorithm cannot fit into the C44's memory. In the latter case, the user must generate an initial system by himself, using the steps outlined later for modifying a hardware set. The input part of an RVM is performed by one or more A/D modules, of the only variety currently in the Egipt RVS. Each provides 3 interchangeable input channels, so one module is provided for every three input blocks in the algorithm.

The output part of an RVM cannot be generated automatically, since more than one type of hardware module is capable of disposing of output data. Thus, the user must provide a mapping between output blocks and their host hardware, as explained in the next section. If in the future there are multiple types of interface, processor or input modules, each type will be assigned using the same type of interface currently used to match outputs to hardware. This will involve more work for the user, but should still be better than assembling hardware fiom scratch.

2. Choosing output hardware

An algorithm output block can be assigned to a D/A module's image or overlay channels for display, or to the host interface module as a statistic, or it can remain unused on the hardware that creates it. In the future, there will likely be additional output devices that can host an output block. It is not possible to specify which hardware an output should use until the hardware is generated, so RVMDES asks the user for this mapping as it generates hardware. RVMDES provides a window, as in figure 3.1, listing the names of output blocks and the hardware that can host them. If a module has multiple channels, like the D/A module in the example, the channels are shown separately. Selectable entries in the two lists identify the output under consideration, and the hardware it is currently assigned to.

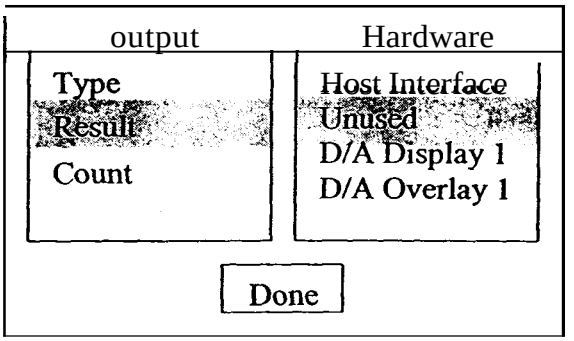


figure 3.1 - The output mapping wizard

3. Finishing a hardware design

When **RVMDES** creates an initial hardware design or the user modifies the algorithm, hardware or partition, the user must command RVMDES to finish the RVM design. RVMDES will then generate executables for any processors, and determine how to configure hardware modules onto motherboards so that all required intermodule connections are made, with a minimum number of cables and overlays. It will display the configuration diagram for the user, who must assemble matching RVS hardware.

Unfortunately, RVMDES cannot always find the optimal configuration. The space of possible configurations can be huge, especially as the number of hardware modules grows, and it is not possible to search much of that space before the user becomes impatient. Therefore, RVMDES does not expect to search the whole space. As it searches, it displays the best valid configuration it has found so far, and allows the user to stop the search any time after finding the first one. The user can then decide whether his time or the possibility of a more efficient design is more valuable.

To help the user decide when to stop searching, RVMDES augments its configuration diagram with a score for the current, best configuration, and the number of suboptimal configurations found since the current best configuration was discovered. As the number of suboptimal solutions grows, the user should have increasing confidence in the current best solution, and feel comfortable stopping the search. Once the user does stop the search, the configuration diagram will remain, so the user can return later to search for a better configuration.

4. Summary

RVMDES automates the drudgery of the initial hardware design and partitioning. It can do this because the limited scope of image processing functions allows a block diagram software representation and a limited set of RVS hardware. The block diagram allows RVMDES to easily recognize an algorithm's input, processing and output needs, and the limited set of hardware allows it to propose hardware to meet those needs. It can then automatically generate the required hardware, or let the user choose between viable alternatives, rather than making him choose hardware and a partition from scratch. Further, by restricting the hardware set and providing a common interface for each module and moth-

erboards to connect the modules, the Egipt RVS reduces hardware configuration to a search problem, which RVMDES can mostly automate. The only part the user must play in finding the configuration is to determine when to stop searching.

B. Estimating Timing

The second step in the hardware design cycle is to simulate each hardware element running its piece of the algorithm and interacting with other hardware. From the results of this simulation, the user must determine whether the frame rate and latency for the entire system meet design specifications, and if not, how to change the hardware or software to reduce the difference. Because **RVMDES** will design with a limited set of hardware, it can incorporate enough information to simulate each type and model the interactions between them, to give the user an accurate picture of the entire system's timing.

Unfortunately, writing and using accurate simulators is a time consuming prospect. As evidence, the Texas Instruments C40 emulator does not even attempt to model timing, and the simulator for the Philip's Trimedia DSP runs two thousand times slower (according to Philips) than the actual chip. In the absence of a viable simulator for the Egipt RVS's single processing module, the C44, RVMDES determines RVM timing by estimation. That is, it estimates the time required for each operation on each hardware module, then schedules the operations onto the modules, inserting delays to account for interactions between hardware modules. This section explains how RVMDES estimates timing for various operations, how it schedules the operations to find total system timing, and how valid the timing estimation is.

1. Timing operations on processors

RVMDES estimates the time required to run software blocks by using formulae associated with each block. This is possible because of two restrictions on the software representation: **RVMDES's** available function blocks are all defined in its code library; and RVMDES always uses the same compiler to turn those functions into executable code. By compiling and testing a function as it is added to the library, it is possible to see what factors affect the time of the resulting executable, and to turn this knowledge into one or more formulae which can be stored with the function's definition. If the **RVS** included multiple kinds of processors, a function could be compiled and tested for each chip, providing additional formulae. When a function is used in a program, **RVMDES** determines which formula to apply, based on conditions in the program, then applies the formula to estimate timing. This section explains what factors are involved in such a formula, and what factors require multiple formulae or modifications to formulae.

The basic formula for software timing is $AXY + BY + C$, where X and Y are the dimensions of an image read or written by the function, and A, B and C are experimentally determined coefficients. The formula reflects the fact that most software blocks are implemented as nested loops over the rows and columns of an image. Time is spent in the loop kernel (A), resetting counters for each row (B), and setting up for and cleaning up behind the function (C). This basic formula has three variations, for functions with data dependencies, secondary inputs, or inputs with reduced dimensions. The first variation adds terms to the

formula to compensate for dependence on the value of certain scalar, integer inputs. As an example, the h c t i o n to generate a Hamming window builds an image of a disk with area proportional to an input T. Pixel values inside the disk take longer to calculate than those outside, but the additional time per pixel is fixed, and the number of pixels inside the disk is proportional to T. The formula AXY + BY + C + DT reflects this. The second variation adds terms to a formula to compensate for secondary input images. This is necessary for windowing operations, which have their main loop over an image, and an internal loop over a window. The main loop kernel, which executes once per pixel in the main image, must spend time processing each window pixel, preparing for each window row, and preparing to apply the window. Thus, for a function using an M by N window, the timing formula expands to AXY + BY + C + $DXYN$ + $EXYMN$. This formula can be reduced somewhat by deleting the fourth term (for resetting window row counters) if RVMDES unrolls the internal loop. The third formula variation occurs when a function that is designed to process an image actually reads an array or scalar instead. RVMDES reduces the looping to compensate, at which point timing follows the formulae $AY+C$ for an array, or C for a scalar. The per-pixel and per-function coefficients may or may not be the same across these reduced formulas, so in some cases the coefficients for each reduced dimension variation must be recorded.

In addition to the need to record several formulas, RVMDES must record several sets of coefficients for each formula, because the coefficients vary in predictable ways according to a function's input parameters. First, coefficients may be reduced if two inputs read the same variable, or if an input that could read an image only reads an array or scalar but does not reduce the dimension of the main loop. The function will not need to load pixel values fiom the repeated or reduced variable, so it will save time. By contrast, coefficients will increase whenever the data types of a block's inputs and outputs do not match the data types specified in the block's definition. In such cases, the compiler inserts type casts to convert the data into usable form, requiring an extra 2 cycles to cast to floating point, or 5 to cast to integer. If fimctions are written without implicit type casts, explicit type casts will always increase coefficients by the predicted amounts, and data need not be recorded in the dictionary file. However, this may be too much to ask from a random user writing a dictionary definition, so instead, the effect of casting each input or output must be recorded. A third factor affecting timing coefficients is the magnitude of constant, scalar, integer inputs. Normally these constants are addressed in immediate mode by the compiler, requiring one cycle to load or otherwise act on. However, operations involving constants valued 0 or 1 are generally replaced with faster instructions that do not require the constant. Conversely, constants that require more than 16 bits (values above 32K) cannot be addressed in immediate mode, and require extra instructions to load, increasing run time. Again, coefficients for each possibility must be recorded.

Despite the predictability of the main formula and its variants expressed above, there are two sources of unpredictability in function timing. One is conditional branching inside functions. In most functions, the execution time of various branches is different, and since the choice of which branch to follow depends on a function's inputs, which cannot be predicted at test time, exact timing cannot be predicted. The other source of unpredictability is C library functions with variable timing, specifically square root, common and natural

logarithm, exponential, and integer division. Again, the time required for these functions depends on input data, and so is unpredictable.

When RVMDES cannot predict exact timing, it records coefficients derived from mean times observed during time testing. Alternately, it could provide best and worst observed coefficients, allowing the user to see the range of possible times, but may not be any more helpful. Note that in some cases where variability is limited to pre or post processing, and has very little effect on a function's run time (typically under 0.5% for an integer division prior to processing a 256x240 image), RVMDES records the worst case coefficient, and counts the function as fixed time.

There is one final set of causes of variability, which are predictable but unmodeled. First, the timing of a h c t i o n probably varies depending on whether its variables are in the same memory banks, and whether they are contiguous in memory, requiring additional sets of coefficients. Because working versions of RVMDES do not support multiple banks or noncontiguous images yet, this is untested and unmodeled. The second cause of variability is software optimization, which hopefully will remove predictable amounts of run time, but as yet does not have predictable effects. The remaining cause of variability is that the compiler will remove multiplications by zero or one after RVMDES has unrolled internal loops in windowing operations. To be accurate, the formulas for unrollable, masked, windowing operations must actually look at each mask entry to determine how each will be compiled. RVMDES does not and probably will not support this.

2. **Hardware timing**

While there are currently no data processing, custom hardware modules in the Egipt RVS, RVMDES includes plans to time those that will eventually be built. In fact, it includes two timing methods, to be used for modules that do and do not buffer data. When hardware buffers data, it acts just like a processor, reading data, processing it, then writing out results, in three stages. In this case, the hardware dictionary entry for the hardware will include a formula for calculating processing time from input size and content, just like the formulas for individual h c t i o n s operating on processors. When the hardware does not buffer data, its input, processing and output stages overlap. In this case, the processing time is the delay between when a single pixel enters the hardware and when it leaves again, which the hardware dictionary will record as a single value.

3. Data transfer timing

Currently, **RVMDES** calculates the time required to transfer a variable between hardware modules as the product of the amount of data to be transferred and the time require for the slower hardware to transfer one word of data. Custom hardware modules operate at fixed speeds so their data transfer rates are fixed. Processors have maximum data transfer rates, which RVMDES executables achieve by transfemng one data item at a time and not overlapping data transfer with processing. These data transfer speeds are recorded in the hardware dictionary with each hardware type. In addition, any connections between modules that run across ribbon cables are limited to 80ns per byte.

In reality, the timing is more complicated than this, because C44s cannot transfer data

asynchronously. A 50MHz C44 can begin a transfer every 20ns, but if a long physical connection, buffers or slower connected module do not allow the transfer to complete during that interval, the C44 must delay until the next 20ns boundary before starting the next transfer. When two C44s with different clock speeds transfer data, the timing is even more complicated as both sides decide when they will initiate and accept a transfer. Luckily, the Egipt RVS only includes one speed C44, and all custom hardware is asynchronous, and either faster than the C44 or able to operate at 80ns per byte, so RVMDES's simple timing scheme works.

4. Total timing

Using the above techniques, RVMDES can determine the time required for every operation that will be performed in the course of processing a frame. Still, RVMDES must determine how these operations fit together, to provide the total time required for a fiame. This involves a three step process, of determining execution order for each hardware module, adding delays to account for interactions between modules, then finding the total time for the fiame.

In the first step, RVMDES determines the order of execution of operations partitioned to each module. Custom hardware modules have at most one processing step, accounting for the timing of the whole module. Processor modules running software from a single flow have a predictable order of operations, which RVMDES will have determined while writing the module's executable. Processors running software with branch blocks do not have a predictable execution order, because the choice of which flows to execute after a branch is made at run time. Therefore, if an algorithm contains a branch block, RVMDES requires the user to provide simulation file names for input blocks, then simulates the algorithm, recording the actual order of operations. This order is used to predict timing. Incidentally, when unbuffered custom hardware modules are added to the Egipt RVS, it will be necessary to insure that their input and output timings are equal. Thus after all data transfer timings have been calculated, those involving custom hardware will need to be modified, possibly propagating changes in timing along a chain of several hardware modules.

In the second step, RVMDES determines the starting time for each operation. Each operation is given a starting time equal to the sum of the run times for all earlier operations in the same hardware module. This timing represents each operation executing in order, with no delays. Next, delays are added so that data transfers between modules occur at the same time on both modules. This typically involves delaying an input (and everything after it) until the output that feeds it is ready.

The final step in timing is to determine latency and frame time for the system. Latency is the delay between when the system's first input begins when its final output ends. In the timing information calculated above, the first input begins at time zero, so the latency matches the time at which the final data transfer (or fimction) completes. RVMDES finds this time by calculating and comparing the ending time of the final operation on each hardware module. Frame time is the time required for a hardware module to process on one frame of data. It is the same for each module, because a module can only cycle as

quickly as the modules it communicates with, and all modules indirectly communicate in an RVM. Thus the slowest module predicts the fi-ame time, and any faster modules will match that time by sitting idle between fiames. RVMDES finds the run time of each module by subtracting the starting time of the module's first operation fiom the ending time of its final operation. It then keeps the longest time as the frame time.

5. Results

To test the effectiveness of timing estimation, I checked how well software timing prediction would work. Of 200 primitive functions in the RVMDES's function library, 104 have predictable timing, and 10 more are convolution operations that could be predicted if RVMDES tested every entry of their mask inputs to determine how they will compile. This still leaves 43% of the primitives whose timing cannot be accurately predicted. Among these unpredictable functions, variability about the midpoint of observed times ran between 7% and 50%, while theoretical variability was even higher. Because timing is a rather critical part of the design loop, the proposed timing method is probably not enough by itself.

Instead, there are two possible timing methods which would provide exact timing: using a real simulator and testing actual timing. While there is no timing simulator for the C44 processor, the Egipt RVS will soon be changing its architecture, and using the Philips Trimedia DSP as its single processor module. A simulator has been written for this chip, and could be interfaced to RVMDES to provide accurate timing information for a given program and set of inputs. The simulator is slow, but should take no more than a few minutes, running on a Pentium, to time an algorithm that must run at frame rate.

A second way to provide exact timing would be to time each function on actual hardware, using a minimal RVM or development board attached to the machine running RVMDES. The user could build an initial system and reduce the time to reasonable values using estimated timing information, then move to a PC with an RVM attached, and finish the design using exact times. Use of a C40 development board to provide exact timings, which provided accurate timing in older versions of RVMDES. was discontinued due to the cost of the extra hardware. However, future versions of RVMDES will include such boards anyway, for real time video capture via the RVM's A/D module, and other tests on pieces of an RVM. Thus it makes sense to at least give the user the option of producing exact times, using that board.

C. Using Timing

RVMDES conveys its recently calculated timing information to the user in the form of a graph. A timing graph begins with a proposed RVM's latency and frame time, fiom which the user can immediately see whether the RVM meets its timing requirements. The remainder of the graph shows how each hardware module spends its time, so the user can locate processing bottlenecks, and determine possible remedies. This section explains how to read the hardware usage part of a timing graph, and how to determine the best way to change the hardware, software or partition to increase an RVM's speed.

1. Reading hardware usage from a timing diagram

The hardware usage part of a timing diagram consists of a set of bars, one for each hardware module, labelled with the module's type and number. Each bar is divided into segments, showing what operation the hardware is involved in at any given time. White segments indicate data transfers, grey ones represent processing and black ones show idle time. Data transfer segments are numbered so the user can quickly locate partner segments. Processing segments are labelled with the names of corresponding function blocks in the algorithm block diagram. Time spent processing branch and join blocks is not represented in the graph, because it is insignificant compared to the time spent on functions. Functions that do not depend on input blocks are not represented either, because they execute before, not during, normal processing.

All hardware cycles over a period equal to the run time of the longest running module. To make this obvious, RVMDES adds idle time to the timing bars for shorter running modules, until all bars have the same length. Also, to keep the graph compact, RVMDES displays what each hardware does during a time slice one frame wide, rather than how it operates on a single frame. To generate the time slice timing information, all segment starting and ending times higher than one frame time are reduced by one frame time, wrapping them around to time zero, and showing processing of the next fiame.

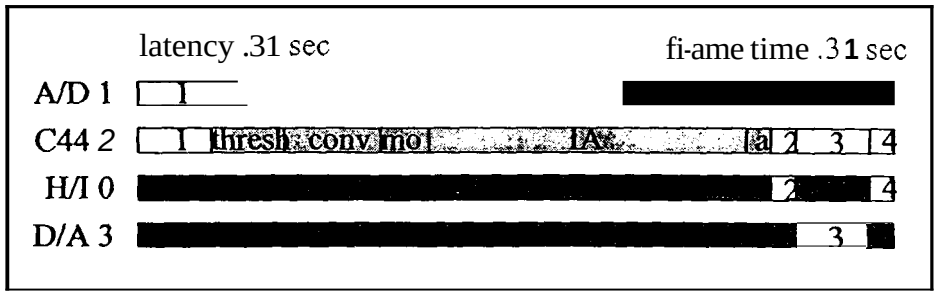


figure 3.2 - an initial hardware timing diagram for the two object inspection algorithm in chapter 2

2. Adding hardware

One way to increase the speed of an RVM is to add more hardware modules and repartition the work among the modules. This is option is available whenever the longest running hardware module is a processor hosting several functions. This will be the case for timing diagrams of initial hardware configurations, such as the one in figure 3.2. Splitting the single processor's work over two processors changes the diagram to the one shown in figure 3.3. Frame time is reduced, because splitting the work decreases the run time of the longest running hardware module, which defines the frame time. Latency is increased, due to the additional time required to transfer data between the hardware modules, but in many inspection systems latency is not critical, so the trade-off is acceptable.

When repartitioning software onto new hardware, the user should try to split work evenly and in parallel. The reason for splitting the work evenly is that frame time is determined

by the longest running hardware module. Splitting evenly ensures that the longest running module is reduced to its minimum possible run time. In addition to splitting evenly, it is best to split separate data paths of an algorithm onto separate hardware. Then paths can be processed in parallel, providing a lower latency than the original implementation required, in addition to the lower fiame time. Splitting processing over a pipeline decreases frame time, as the first module can begin processing a new frame when it passes on the results for the current frame, but increases latency due to the extra time required to transfer data between hardware.

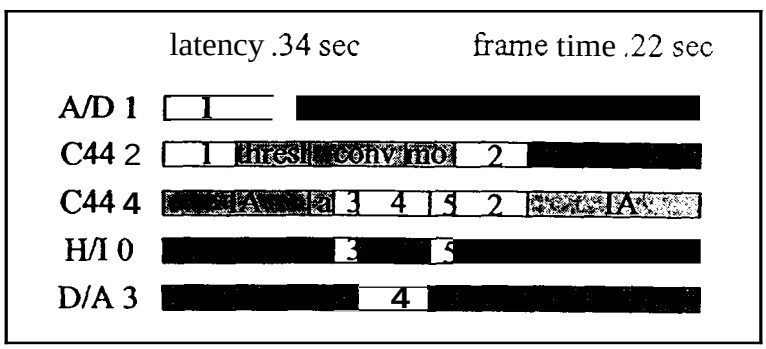


figure 3.3 - timing diagram of slightly improved hardware

3. Duplicating hardware

A second way to increase the speed of a system is to use multiple copies of a piece of hardware, running the same processing steps in parallel on each copy. This is the way to accelerate an RVM whose algorithm has no parallelism, and which cannot afford the latency increase incurred by adding hardware and repartitioning tasks in serial. Duplicating hardware is also the only way to decrease the run time of a module that spends most of its time running a single function, as in figure 3.3. Even though the limiting module runs multiple functions, the time spent on one is almost insignificant. Rather than make the module run faster, multiple copies of the module can process alternating frames. This has no noticeable effect on latency, but cuts the frame time by a factor equal to the number of duplicate modules.

The timing diagram resulting from the duplication is shown in figure 3.4. Halving the time required by the monolithic processor has made the other processor the new bottleneck. To see what the duplicated module is actually doing, the two lines for the processor must be laid end to end. The second processor performs the Same steps, offset by one fi-ame time.

A third use for duplicating hardware is to provide fine grain parallelism. If the set of functions allocated to a processor can execute independently on regions of their input images, it may be possible to split the input onto several processors, and have them all operate on the same fiame at the same time. This slashes both frame time and latency for such processing by a factor equal to the number of processors. Had the modules duplicated after

viewing figure 3.3 been used to pieces of the same frame instead of alternating fiames, the bars would show the same steps, slightly offset in time, rather than the completely different steps shown in figure **3.4.** From this difference, the user can tell which way a module is duplicated.

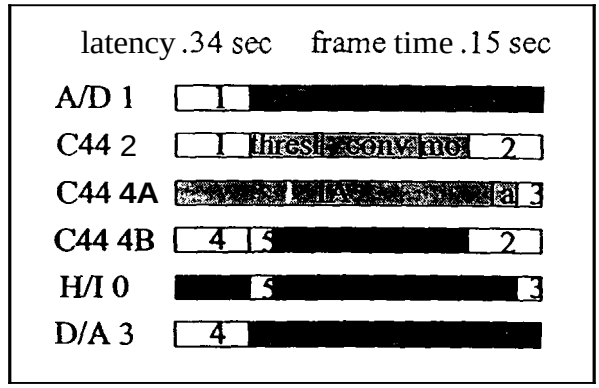


figure 3.4 - timing diagram of more improved hardware

3. Repartitioning for balance

A third way to accelerate a system is to repartition software to more evenly balance processing. This option is available whenever some processors sit idle, while others attempt to perform several functions. This is the situation in figure **3.5,** where ***C44*** 2 is the limiting module, and the duplicated ***C44 4*** has enough idle time to perform some of the former module's processing. Repartitioning would reduce the frame time by using up some of the second module's idle time.

4. Optimizing software

A final way to speed up an RVM is to speed up the functions that it runs. This is particularly useful if a timing display shows that the actual timing is very close to the desired timing. In this case, cost considerations suggest optimizing the current hardware, rather than adding more. RVMDES provides three ways to effect **this** optimization: optimizing functions, optimizing data transfer, and scaling back an algorithm.

The **run** time for fimctions ***can*** be reduced using several optimizations specifically adapted to image processing functions. **RVMDES** will automatically determine which are available, and account for any that the user selects **as** it generates timing estimates. The best optimizations are available when a single processor performs several simple functions in a row, **as** does ***C44*** 2 in figure **3.4.** A *loop* ***combining*** optimization can convert the three functions into a single one, shown in figure **3.5,** which runs faster than the three individual functions. This speedup applies to both frame time and latency, possibly providing enough speed increase to meet specifications.

In addition to function run times, data transfer times can also be reduced in some cases. The time required to transfer data over a cable is double that for transferring data across

any number of motherboard traces. If a data transfer to or from the time limiting module happens across a cable, it may be wise to tell RVMDES to find a better configuration of hardware modules that does not require cables for those transfers. If more than six hardware modules are involved in a design, cables will still be needed, but perhaps they can be used between processors that have idle time, and do not need the fast data transfers.

One additional possibility is that an RVM is too slow, but adding more hardware would exceed its cost or size specification. In this case, the only recourse is to design a simpler algorithm that can run on less hardware.

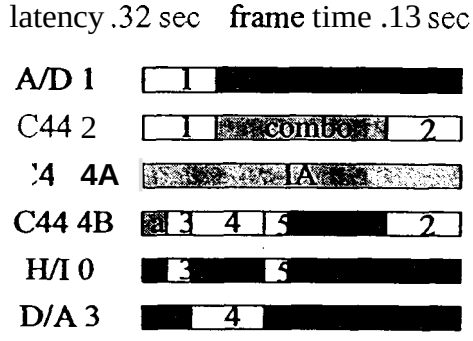


figure 3.5 - timing diagrams of final hardware

D. Modifying Hardware

RVMDES's timing diagram makes it easy to identify whether a system is fast enough, why not, and what remedies to apply. RVMDES also facilitates implementing each of these remedies. For some of these solutions, namely rewriting the algorithm and restarting the search for a better configuration, the user repeats steps he has already taken. This section explains how RVMDES allows the user to quickly and easily implement the other solutions. In particular it explains how the user can add hardware, repartition software onto new or different hardware, and optimize the software on a particular hardware module.

1. How *to* add hardware modules

When timing diagram indicates that a hardware design is much too slow, the solution is generally to add more hardware, either by duplicating existing hardware, or by adding entirely new hardware. RVMDES provides a graphical hardware editor in which the user can quickly accomplish both tasks, and easily keep track of what hardware he has generated.

The editor, shown in figure 3.6, is very similar in form to the software editor. Each hardware module is represented by a box, labelled with the module's type and unique number. This is the same information used to label bars on timing diagrams and identify modules in error messages. To create a new hardware module, the user clicks on a module type in

the list of supported hardware modules, shown on the right. This list is made possible because **RVMDES** works with the finite set of hardware in the Egipt **RVS. As** new modules are added, they are given consecutive identification numbers, and added to the right of or below the four modules of the initial hardware choice, as in the figure. To split a module into a parallel array of modules, the user can pull up a specification window for that module' icon, just as in the software editor. From this window, he can specify the number of parallel modules that the icon should represent, and whether input data should be divided and processed by all modules in every fiame or dealt to a different module for each fiame. A single box will continue to represent the duplicate modules, which still serve one purpose, but when RVMDES generates a hardware configuration or timing information, it will use the full number of modules.

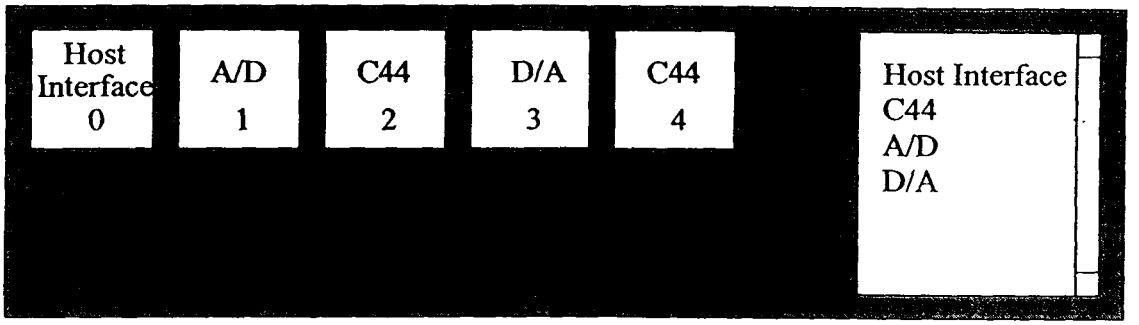


figure 3.6 - the hardware editor showing the final hardware for the two object inspection example

In addition to making it easy to create new hardware, RVMDES automatically integrates new hardware into the system. When the user commands RVMDES to turn his algorithm and hardware set into an **RVM, RVMDES** examines the connections between software modules and how the modules are partitioned, and automatically determines the connections between hardware modules. At the same time, it automatically generates multiple copies of any modules that the user has marked as duplicated, generates any *fork* and merge modules required to split input data across the duplicated modules or collect results fiom them, and generates all of the connections necessary to splice the parallel array and its supporting modules into the rest of the system. Because all work beyond the choice of which modules to add or duplicate is handled by **RVMDES, the** user can concentrate on adding hardware modules to overcome specific processing bottlenecks, rather than mentally detouring to actually design hardware. This will allow the user to test new hardware choices in less time.

2. How to repartition

The second way to improve the speed of a system is to repartition software from an over-used module to a new or under-used one. This can be done easily in **RVMDES,** because the software representation consists of blocks, which can be assigned individually to hardware modules. The software editor includes a display mode wherein each software block displays the number of the hardware module to which it is partitioned, in addition to the

block's name. The user can match this information to timing diagram segment names and bar labels, to quickly locate the software blocks that represent timing segments causing bottlenecks. He can then repartition these blocks onto new hardware by changing each block's hardware number in a pop up window associated with the block. Because timing bars are labelled with hardware module numbers, the user can decide which module to partition a block onto by choosing a bar that is not busy.

This type of repartitioning is simple, involving little work beyond identifying bottlenecks. However, like other facets of the software editor, it is overly general, and allows the user to create partitions that RVMDES cannot convert to an RVM. To compensate, when the user instructs RVMDES to complete a hardware design, RVMDES checks that certain partitioning rules are obeyed, and refuses to generate a design if they are not.

The first rule is that all software must be partitioned onto hardware that can execute it. Software is automatically partitioned when RVMDES creates an initial hardware system, but may become unpartitioned if the user deletes hardware or repartitions software onto nonexistent hardware. In addition, if the user adds software without creating a new initial hardware set, the new software will be unpartitioned. RVMDES checks for these unpartitioned conditions by comparing the assigned hardware number for each software module to the list of hardware modules. The more subtle problem of a hardware module not being able to execute the software partitioned to it is tested with the help of the hardware dictionary, which lists, among other things, the software functions that each hardware module can execute. If the list of software blocks partitioned to a hardware module is not equivalent to or a subset of the list of blocks allowable for that type of module, RVMDES will refuse to generate a hardware design.

The second rule is that the part of the algorithm partitioned to any one hardware module may only have one starting point. This is so you always know where to start a frame, at any give module. The most obvious requirement is that all inputs to a hardware module must belong to the same flow. Were this not the case, RVMDES would need to read some inputs but not others, depending on which flow was active, yet it would have no way of knowing which. With all inputs in one flow, it can read all inputs before beginning processing, knowing that they will all be available. A more subtle requirement is that the input flow cannot leave and reenter the same module. Thus a pipeline of three software blocks could not be partitioned so that the first and third were on the same module, but the second was not. In such a situation, the module would not be able to read all inputs prior to processing, because the input to the third block will not be generated until the first block is processed.

The third rule is that the software partitioned onto a hardware module must not contain any race conditions. These occur when a software block sends one output to multiple blocks on the same hardware, at least two of which modify that data instead of creating new data when they run. Whichever runs first will corrupt the data before the second can use it. An equivalent situation occurs when only one of the multiple blocks modifies the data, but RVMDES cannot set the module's execution order to execute the modifying block last. If RVMDES detects either of these cases, it will refuse to generate code for the

module and to design hardware. The user must insert a copy block to provide a second copy of the data, on which the modifying block can operate. Note that race conditions are considered errors in partitioning, not in an algorithm, because partitioning two connected blocks onto different hardware modules implicitly inserts a copy operation between them, and can negate race conditions. Further, they do not affect the algorithm simulator, which implicitly inserts copy operations across every connection.

The final rule constraining partitioning is that each hardware module must have enough ports to support all of its input and output needs. While the user does not directly control how hardware modules are connected, he does control how many software connections enter and leave each module, by deciding which software blocks connect to which, and which blocks are partitioned onto which hardware. The set of software connections between blocks on any two given hardware modules constitutes a hardware connection, and is assigned a physical port on each module. If in the course of assigning ports, RVMDES runs out of ports before running out of connections, it will refuse to generate a hardware design, and the user must repartition software so that the module needs fewer connections.

3. How to optimize an algorithm

The third way to speed up an RVM is to optimize its software. Software optimizations can increase the speed and decrease the memory requirements of an algorithm, potentially providing the extra push needed to run the algorithm without adding hardware. RVMDES offers such optimization to users by supplying a library of individually optimized image processing functions, and by supporting several *intermodule* optimizations tailored to take advantage of the relationship between the image processing fimctions in an algorithm. RVMDES does not automatically apply these intermodule optimizations, but it does provide an interface fkom which the user can apply them with minimal work or distraction.

Normally, intermodule optimizations are difficult to work with. They modify source code, making it difficult to understand, modify and reuse, and are useless for programs written with the precompiled functions in image processing libraries. Compilers can automate optimization to some extent, but are limited to making local optimizations, where they can interpret a programmer's purpose from his code and ensure that optimized code will accomplish the purpose without any damaging side effects. Modifying or replacing entire functions is too large a task for a generic compiler. RVMDES, on the other hand, can interpret the user's intentions, suggest optimizations and apply them on a larger scale, because it uses a very high level program representation wherein individual tokens identify the user's intentions, and are optimizable in predictable ways.

Thus RVMDES can determine what intermodule optimizations apply to a program, and can apply those optimizations. Unfortunately, it cannot determine the optimal set of optimizations to apply. It cannot simply apply all optimizations because some prevent or enable others, nor can it determine which subset of the available optimizations is optimal because the benefit of an optimization is often data dependent. Hence, RVMDES leaves the problem of choosing optimizations to the user, who must determine, by trial and error, which optimizations work the best for images in his particular application. To make this

process quick and easy, RVMDES automatically detects optimizations that can be applied to an algorithm, and automatically applies any that the user selects. The user only needs to select optimizations fiom a list provided by RVMDES.

When the user decides to optimize software, RVMDES opens a window and displays a list of optimizations currently available to the algorithm- Each entry names the type of optimization and the software and hardware modules involved. If an optimization involves software blocks on separate hardware modules, RVMDES prepends an 'X' to the optimization's list entry, indicating that RVMDES cannot actually apply the optimization, but would be able to if the user repartitioned the software blocks onto one hardware module. In addition, RVMDES prepends a 'P' to optimizations involving blocks that only execute when the RVM boots or receives commands fiom its operator. This flag reminds the user that the optimization will have no effect on the RVM's regular execution speed. To see which software blocks are involved in an optimization, the user selects a list entry, and RVMDES will highlight the appropriate blocks. To apply an optimization, the user simply double clicks its entry. RVMDES then records the optimization, and updates the list of remaining optimizations to remove conflicting optimizations and add newly available ones.

The RVMDES optimization window allows users to apply intermodule optimizations without suffering the difficulties normally associated with them. Specifically, the user does not need to detect possible optimizations, hand optimize code or debug the results, so is not distracted from the task of reducing run time. Further, applying optimizations does not make the algorithm hard to interpret, because RVMDES records optimizations separately from the algorithm, and can display the algorithm with or without optimizations, Finally, an option to clear all optimizations with a single button press removes a major barrier to code reuse -- the time required to unoptimize existing code before modifying it to a new purpose.

E. Summary

RVMDES makes hardware design fast and easy, by off-loading most of the work from the user, and by providing easy to use interfaces for the remainder of the work. The initial hardware design is generated almost automatically, only requiring the user to answer a few questions about how outputs are used, and to decide when RVMDES has found a good enough hardware configuration. Because the set of hardware available in the Egipt RVS is known, RVMDES can simulate timing of each type of hardware and the interactions between them, automatically providing the user with an estimate of the frame time and latency of a hardware design, and a breakdown of how that time is used for each block in the algorithm. From this breakdown, he can quickly determine which software blocks and hardware modules slow the system down, and how those bottlenecks can be removed. Finally, each type of bottleneck removal is made simple: a hardware editor takes advantage of the finite set of hardware types, allowing the user to quickly generate and duplicate hardware modules; the block diagram software representation allows the user to quickly repartition software onto new hardware; and an optimizer tool allows the user to optimize algorithms without touching code or having to read optimized code. These features are

made possible because of the special structure of image processing functions, and the block diagram RVMDES uses to assemble them.

IV. Software Optimization

RVMDES is intended to provide not only an efficient RVM design cycle, but also efficient RVM designs. To this end, it helps the user apply software optimizations to his block diagram algorithms. These optimizations reduce the amount of time and memory required to run the algorithm, allowing it to be implemented on less hardware or to run faster on current hardware.

The optimizations RVMDES provides are very coarse grained, operating at the level of entire functions, rather than C statements or assembly instructions. Optimization on the latter can be built into the definitions of individual functions or added by an optimizing compiler. RVMDES's high level optimizations, on the other hand, are only available because RVMDES limits itself to image processing algorithms, which consist of a predictable set of functions that can be optimized in predictable ways. To provide these optimizations, RVMDES requires three things, which are the subjects of this chapter. First, it requires an identification of the optimizations that leverage the special structure of image processing programs. Second, it requires a language which reflects the special structure. Third, it requires a way to keep track of which optimizations have already been applied, so it can determine what others can be applied.

A. Optimizations for Image Processing Programs

There is a special structure to image processing programs, consisting partly of a predictable structure used to code individual functions, and partly of the limited ways in which functions interact. This predictability also makes the set of optimizations that apply to an image processing program predictable. This section looks at components of the predictable structure, and what optimizations can be derived from them.

1. Looping structure and optimization

A large fraction of image processing functions are coded as loops that touch one pixel of an image during each iteration, plus some pre- and post-processing. Each loop iteration can write one pixel to an output image (as in convolution), read it from an input image (as in histogramming), or do both (as in table lookup). Functions that do not fit this model often consist of a series or iteration of functions that do (as in image normalization or skeleton extraction). Thus loop based functions form something of a set of primitives, from which image processing programs are composed.

It is sometimes possible to combine two of these primitives such that the resulting primitive accomplishes both original tasks, but requires less data storage space and/or memory access. Such larger, less memory intensive, combined loops are also less susceptible to delays from pipeline conflicts caused by locked address registers and busy memory ports. To gain these advantages, a C programmer will often combine loops in his code, though it makes the code difficult to follow. A library or tool kit user could also use combined loops, if a library provided code for every available combination. Unfortunately, if a user added a new function to such library, he would have to add code for every combination it could be involved in, which is not reasonable. Thus the user must choose between an

expandable library, and an optimizing library. Many optimizing compilers combine loops to some degree, but are they are limited because of the difficulty in determining whether two arbitrary loops can be combined without corrupting the results of the second, and whether there is any advantage to combination. However, within the limited domain of image processing, further limited by a well defined library format, the loops are not. arbitrary, and it is possible to decide when combination is beneficial. This section lists two cases in which loop combination is a optimization, one optimization that supports combination, and three more combinations that are possible, but not beneficial. RVMDES implements only the first three.

The first optimization, serial combination, combines two loops when one writes an image and the other reads that image. This optimization is available if the two loops touch exactly one pixel of the intermediate image during each loop iteration, as depicted in figure 4.1. If this condition is met, the two loops necessarily have the same dimensions, and the first produces information in time for the second to use it, so the kernels of the two loops can be concatenated to produce one, combined loop. In addition to any speedups gained from halving the number of post-iteration branches, combining loops eliminates a memory read because the pixel created by the first half-kernel will already be in a register when it is required for the second half-kernel. The few processor cycles this saves can be significant in a tight loop. In addition, if the intermediate image is read only by the second loop, it need not be stored after combination, so the new loop can use a single register to represent each pixel as it is needed, as depicted on the right side of figure 4.1. This eliminates the need for memory to hold the image, and reduces time requirements by eliminating a slow memory write. These features make serial combination the most useful of the image processing optimizations.

The second optimization, parallel combination, merges two loops that read the same image. It is possible when two loops read the same data, one pixel at time, and neither reads data created by the other. This situation is depicted in figure 4.2, where two operations create two separate images using data from a common input. The kernels of the two operations can be concatenated in either order, and each input pixel can be read from memory once for the two kernels, saving a memory access.

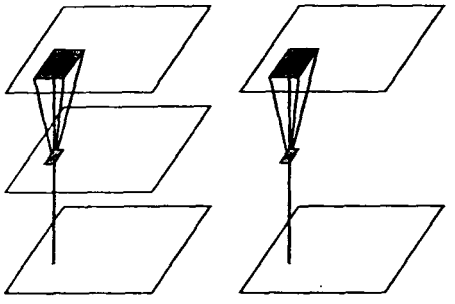


figure 4.1 - serial combination

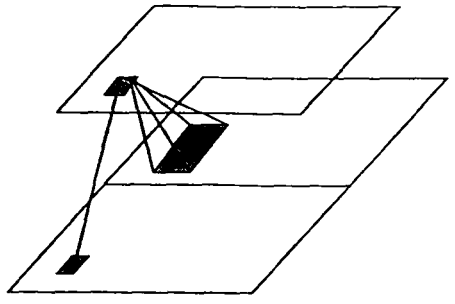


figure 4.2 -parallel combination

The third optimization, called inversion, is not a combination but aims to provide more opportunity for combinations by allowing loops to be recast to either touch their output

pixels once, or their input pixels once. As an example, consider the most commonly used invertible function, convolution. It is usually written as a loop over an output image, filling each output pixel with the dot product of a mask and a corresponding window in the input image, as depicted on the left of figure 4.3. However, it can instead touch each input pixel once, incrementing the pixels of a window in the output image by the values in the mask, scaled by the input pixel's value, as shown on the figure's right. Unfortunately, inverted masking operations involve many write operations, while their uninverted counterparts involve mostly read operations. On DSPs like the C44, writes take twice as long as reads, so the extra overhead introduced by inversion generally outweighs any gains made in subsequent combinations. The one exception is when input images contain a large number of zeroes. Inverted convolution can be written to skip iterations where the input value is zero, as those pixels will have no effect on the output. This can significantly reduce the amount of calculation required on binary or sparsely populated images.

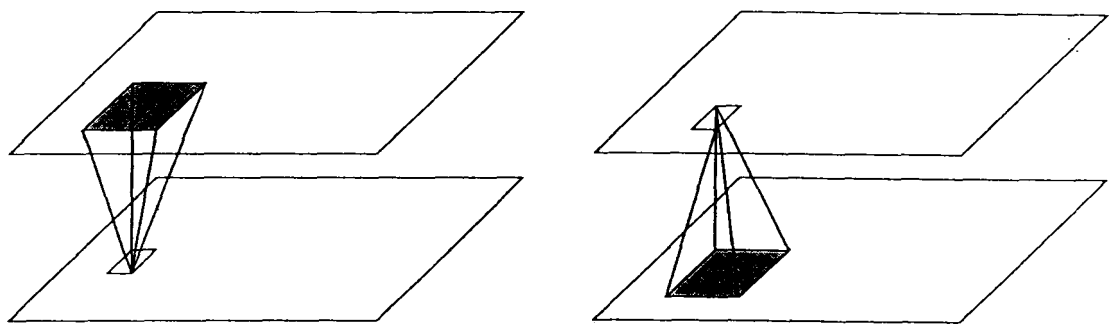


figure 4.3 - an invertible function

The fourth combination operation, shifted combination, is possible when one loop writes one or more images one pixel at a time, and the next loop reads them as part of a masking operation, as in figure 4.3. The second loop's dimensions will be smaller than the first's by the size of the mask, so the two loops cannot be directly combined. However, the first loop can be split into a main body, with the same dimensions as the second loop, and a border covering the remaining area. The boundary between the data processed by these loops is shown by dotted lines in the figure. If the border is completely processed before the main body, then the initial iteration of the second loop only needs one additional pixel value, calculated by the first iteration of the main body of the first loop. As the second loop continues to iterate, it advances the mask, each time requiring one new pixel from the corresponding iteration of the main body of the first loop. This situation is analogous to the one allowing serial combination, so the main body of the first loop can be merged with the second loop in the same manner. The combination is termed shifted because the first half kernel will refer *to* coordinates of main body pixels, whose indices are shifted from the loop coordinates by the mask dimensions.

Unfortunately, while the idea of separating loops to allow combination may be interesting, it is not very useful. The intermediate image cannot be eliminated, because a swath the

height of the mask must be maintained for use when the mask advances to the next row. It would be useful if the C44's rotating buffers could be employed, but this is difficult to do fiom C code. Thus **shifted** combination cannot eliminate memory requirements or memory writes. In addition, it is difficult to write functions so that a single element of the image is read from a register while the rest are read fiom memory, so **shifted** combination does not even save a read access. In short, without some very fancy programming, shifted combination provides no benefits.

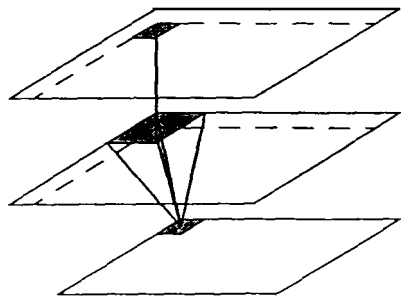


figure 4.3 - shifted combination

The fourth optimization, trailing combination, is another generalization on serial combination. Like serial combination, trailing combination requires two loops over the same image, with one loop creating the image and the other reading it. The difference is that in any given iteration, the second loop may read the pixels created by the first loop during that iteration or any previous ones. This would save a read access but no write access or memory, just like a serial combination whose intermediate image was used by a third loop. Trailing combination could conceivably be useful, except that none of the functions in **RVMDES's** function library have any use for it.

The final combination operation, which is generally not an optimization, combines two loops that have the same dimension, but no other relation. There is a possibility that by decreasing the amount of post-iteration branching, combining loops can decrease their required processing time. However, if loops have no common images, combining them cannot save **any** memory accesses or space. Also, while it is possible to avoid some pipe-Line conflicts by interleaving the kernels of very tight, unrelated loops such **as** single pixel operations or convolutions with **1x3** filters, optimizing compilers may be smart enough to do that automatically. On **the** other **hand,** combining two large loops that each use over half of the available registers force the processor to spill register data to memory, actually decreasing the speed of the combined loop.

2. Sequence of filters structure

A second structural element of image processing programs is that they consist of or at least begin **with,** sequences of simple filters. These filters can be broken apart and in some cases reordered to reduce the number of operations required to execute them.

Many filtering operations that use two dimensional masks can be separated into two consecutive operations, each using a one dimensional mask. The sequence of functions can often run in dramatically less time than the single function, because the single function

performs some calculation at each of $M*N$ pixels under an M by N mask, while the two smaller functions look at only M+N pixels. For medium sized, 7 by 7 masks, separation slashes the number of operations required, and thus the time required, by around 70%. Separation is mainly applied to convolutions, shown in figure 4.4. It is possible whenever two one dimensional masks could be convolved with each other to make the two dimensional mask. Beyond this, separation is also available for nonlinear filtering operations like finding the variance or maximum value in a moving window.

figure 4.4 - normal and separated convolutions

In addition to separating convolutions, it is often possible gain some advantage by scaling them by some constant, then scaling the output by the reciprocal constant. This situation is depicted at the top of figure 4.4b. The values of a derivative filter are doubled, and the factor of one half is multiplied into the result as an additional step. A good compiler will eliminate multiplication by 1 or -1 if the calculation is hard coded, saving two instructions per pixel. The appended multiplication will still cost one cycle, but the scaling will have saved one cycle per pixel, in a loop that should only require 6, providing a 16% time savings. This may be a best case scenario, but any convolution masks that will be unrolled by the compiler or the RVMDES code generator will benefit if even two weights are scaled to ones.

Once constants have been separated, they can often be moved around in a block diagram to increase program speed, as in the second half of figure 4.5. In this case, multiplication by a factor of one half can occur before or after finding the maximum value of an image. The maximum value will not change, but scaling the single maximum value rather than the whole image essentially eliminates one multiplication per pixel. Had the maximum finding operation instead been a normalizing operation, the multiplication could have simply been eliminated. Were it a thresholding operation, the threshold could have been doubled rather than halving each pixel. All of these operation effectively remove the time required for the multiplication step.

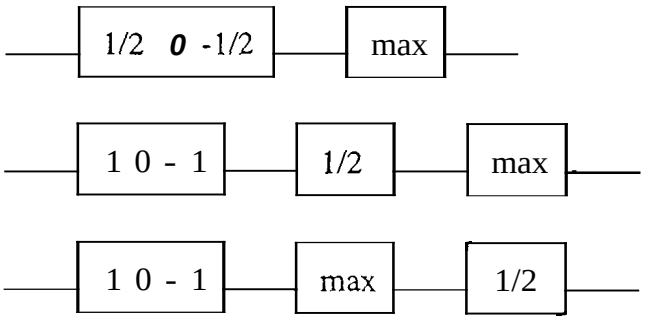


figure 4.5 - scaling and reordering operations

When reordering cannot obviate operations, it may still provide opportunity for optimization. Consider for example the first system in figure **4.6,** which applies non-maximum suppression to an image, and finds the maximum value of that same image. Neither operation cannot be separated, scaled, inverted. The **two** operations also cannot be combined, because the non-maximum suppression loops over the dimensions of its output, while the maximum finding loops over the larger dimensions of the common input. However, the maximum finding operation could operate on the output of the non-maximum suppression, where it would produce the same result and loop over the same image. This would then allow the **two** operations to be combined, forming the final system in figure 4.6.

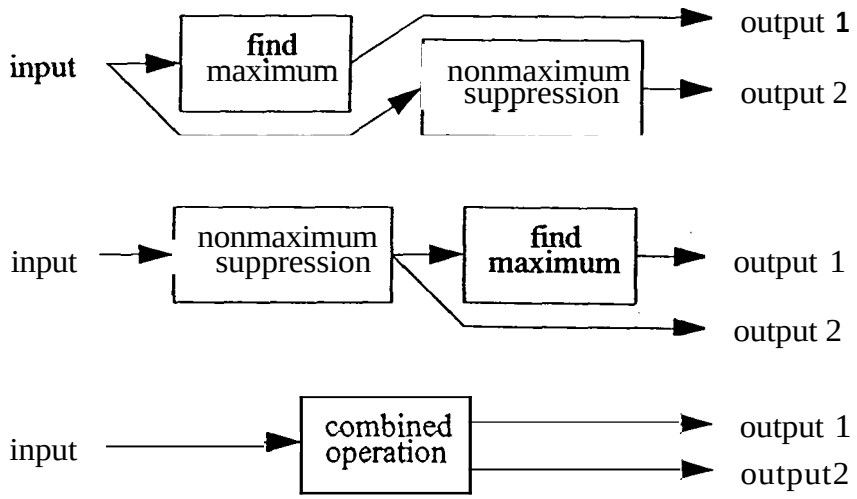


figure 4.6 - reordering to allow other optimizations

3. Discrete structure

Another structural regularity of image processing programs is that they tend to operate on images with 256 grey values. This means that complicated and time consuming functions of one or **two** pixels, such **as** logarithm calculation and some nonlinear edge detecting algorithms, need not be applied at every pixel in **an** image. Instead, the mapping **from** all possible inputs to outputs can be precalculated, and the function can be implemented as a

lookup table.

Additionally, for medium sized images (256x240), coordinate transforms can be implemented as lookup tables. The table would consist of an array with the same dimensions of the output image, telling which input pixel maps to each output pixel. The lookup table would increase the speed of linear transforms like warping two camera views into alignment, or warping an oblique view of a road into an overhead view. It would be even more beneficial for transforms involving slow operations like trigonometry or division, used to convert between polar and Cartesian coordinates, or to undo perspective effects. Coordinate transforms that map one input pixel to one output pixel may be less reliable than those that average neighborhoods, but often the increased in speed warrants the decrease in accuracy.

4. Accumulator structure

The next useful feature of many image processing functions is that they involve a lot of copying. This includes operations like clipping or overlaying, where a function generates parts of the output image, and simply copies other parts fiom the input image. Other functions copy images into windows of larger images, or into larger arrays sized for a particular output device.

This copying is fertile ground for optimization, because copy operations can be reduced or eliminated if the same memory is used for input and output images. For instance, a clipping operation could just change the pixels which must be clipped, and leave the rest alone. If only a small number of pixels must be clipped, most of the function's memory writes can be eliminated. Similarly, a line drawing operation can simply modify the few pixels that form the line. If a copy operation is used to extract a window fiom an image, and neither the original data nor its memory space are needed for later calculations, the window can be used while it is still embedded in its host image, eliminating the copy operations. Finally, if an image will eventually be inserted into a larger image and padded for display purposes, it can be created inside that larger image, eliminating the final copy. These optimizations reduce a program's run time and memory requirements.

Note that while using the same image for input and output can eliminate the need for large chunks of memory, it is only a good idea when it eliminates or reduces a copy operation. One reason is that separate images can be placed in separate banks of memory. On the C44 DSP, alternately reading and writing to one bank of memory stalls the processor every other cycle, while alternately reading fiom one bank and writing to another creates no stalls, doubling the execution speed. Thus, using a single bank is bad unless a copy operation eliminates most of the processor-stalling memory writes. A second reason for not overwriting the input image is that multiple functions may read it. In this case, the image must not be overwritten until all functions have read it.

5. Non-looping structure

Most image processing functions follow the structure presented in section 1 -- a loop, in reading order, over the pixels of an image, touching each pixel once. This structure allows several loop combining optimizations, and is compatible with the other optimizations dis-

cussed so far. The functions that do not follow the structure fall into several groups, which, though they do not have obvious optimizations, are listed here for completeness.

The first set consists of data dependent functions, such as line following. These functions have a main loop, but its dimensions are unpredictable, so it cannot be combined with other loops. One possible source of optimization is that outputs of data dependent functions are often lists, written one pixel at a time. It may be possible to combine the functions with later loops over the output lists, but this is not currently possible using the structure RVMDES has built around standard looping functions.

A second set of nonstandard functions is the orthogonal transforms, which are based on butterfly operations. They can be written as a series of loops, but each dimension requires a pair of nested loops, which is not easily reconciled with the single loop required by a normal function. Further, many **DSPs** provide special instructions for handling butterfly operations, so that they are best implemented in assembly anyway. Thus RVMDES makes no attempt to optimize with them.

A third group of contains functions that do not follow reading order, typically looping by columns, or reading from the bottom of an image. It is possible expand the notion of a standard function to allow arbitrary reading order, but this would require extra work to track which orders each function can operate in. The set of looping functions that do not follow reading order is sufficiently small that they do not warrant the added complication at this point.

A final set of nonstandard functions are those that behave differently according to the value of a state variable. *An* example is run length encoding, which either increments a counter, or resets the counter and writes output depending on whether the current and previous pixels have the same value. Such fimctions could be written to modulate their behavious by checking the state in each loop iteration. However, if the state remains constant for a large number of iterations, the function would actually go faster if the code to handle each state included a loop that operated until the state lapsed. This would avoid unnecessary state checks, but also makes the function difficult to reconcile with standard looping functions.

B. Optimization oriented function definitions

RVMDES lets the user apply three types of intennodule optimizations to his algorithms: serial combination, parallel combination and inversion. One reason it can provide these optimizations is that RVMDES algorithms are written in a block diagram representation, which makes the boundaries and connections between functions very clear. This allows it to quickly determine where it could try to optimize. The other reason is that RVMDES's library of function block definitions uses a definition format designed specifically to show what optimizations can apply to a function, and to translate into code with the optimizations applied.

This section explains the definition format used in RVMDES's library. Each definition

begins with a header that identifies the function and provides information necessary to detect applicable optimizations. This header is followed by a primitive definition body or macro definition body, encoding the actual function definition. The former represents image processing primitives, and is designed to translate easily into C code that reflects any combination and inversion optimizations the user has chosen to apply. The latter represents higher level image processing functions, in a way that facilitates optimizations normally applicable only to primitives.

1. The function definition header

The function definition header is intended mainly to identify a function's name and ports for editing purposes. However, it is also designed to help RVMDES identify any optimizations that apply to a function. This help takes the form of four flags describing the function: macro, separability, invertibility, and pixelwiseness.

The first two flags mark glaring optimizability in an algorithm, which RVMDES will remedy without even consulting the user. The macro flag tells whether a function is defined by a primitive definition or a macro definition. RVMDES will use this flag to identify and break apart macros, because macros cannot be optimized, but their component primitives often can. The separability flag marks any function that uses a moving two dimensional window, but might be separated into two functions in series, each using a one dimensional window. This idea hails from convolution, wherein convolving an image with a horizontal mask and then a vertical mask is functionally equivalent to, but requires less calculation than, convolving the two masks with each other and then with the image. Separability also logically extends to many other window based operations such as finding the maximum value in a moving window. When a function can be separated, doing so is almost always beneficial, so the function definition header identifies any potentially separable functions.

The *invertibility* flag tells whether a function can be inverted, to loop over an input instead of an output. Thepixelwiseness flags tell whether each input and output of the function is pixelwise. A variable is pixelwise in a function the function consists of a loop of the same dimensions as the variable, touching exactly one pixel of the variable in each iteration, and touching each pixel exactly once. Invertibility and pixelwiseness will be the main, algorithm independent factors in determining whether inversion and combination optimizations apply to a function, so function definition header makes them easy to access.

2. The primitive definition body

The primitive definition body is designed to encode the definition of a standard image processing function, in a way that facilitates generation of optimized code. Standard image processing function, as used here, refers to a loop that touches each pixel of an image once, surrounded by some pre- and post-processing. To encode this, the primitive definition body consists of four parts: local variable declarations, initializations, a loop kernel, and post-processing. Individual lines of the definition are written in C code, so to convert a definition to an actual C function, RVMDES *can* simply transcribe the four parts, in order, adding a looping statement around the loop kernel and a function name and braces around the whole function. The dimensions of the main loop will match those of a pixel-

wise input or output of the function (by definition), and the definition header tells which variables can supply these dimensions. Although the definition is designed for standard image processing functions, it can also represent functions that do not follow the regular model, by writing arbitrary code in the loop kernel field, and not marking any variables as pixelwise, so that RVMDES will not add a looping statement around the kernel.

The primitive definition body also facilitates writing code for functions that have been optimized by serial or parallel combination, inversion, or both. If two functions are combined, either in serial or in parallel, RVMDES can interleave the four fields of each definition, adding a single looping statement around the two adjacent loop kernels. Both functions will agree on the dimensions of the loop, as this is a precondition for combining functions. Also, the combined function will maintain the required code order, of variables, initializations, main loop and post-processing.

To facilitate inversion optimizations, invertible functions are defined by two primitive definition bodies, providing code for the function's regular and inverted forms. It would have also been possible to record an invertible function as two functions and have some standard naming convention or record the name of the inverted h c t i o n , but then the user would have to choose which function to put in his algorithm. Because inversion is an optimization issue, it should be hidden fiom the user as he builds or reads an algorithm, so the inverted and uninverted definitions are maintained as one function. To write code for the definition, RVMDES must simply decide whether to transcribe the regular or inverted body

3. The macro definition body

The second way to define a function is with a macro definition body. This is useful for encoding the many functions that do not follow the "standard image processing function" model represented by a primitive definition, but can be built from a set of such functions. An example is normalizing an image to 8-bit grey scale, which actually consists of finding the image's minimum and maximum values, then rescaling the image from its old range to the new range. The macro definition represents such compound functions as block diagrams, as if they were algorithms, recording the existence and parameters of each block in the diagram, and the connections between blocks. A macro definition does not translate to code, and cannot be optimized, but RVMDES can replace a macro block in an algorithm with its equivalent block diagram, whose blocks will be optimizable, translatable primitives, or more, expandable macros. This is preferable to recording compound functions as non-looping primitives, which could translate directly to code, but would not be eligible for loop combination optimizations.

C. The optimized module list

RVMDES applies optimizations as it translates primitive definitions into C code, not as the user chooses optimizations to apply. It must record optimizations that the user chooses, in a way that allows it to recognize the effects of those optimizations when it determines what additional optimizations could be applied, and apply the optimizations when it generates code. The record should also be made in a way that does not affect the

internal representation, in case a user wants to view the unoptimized algorithm, which is likely to be more easily understood, or quickly undo optimizations prior to code reuse.

RVMDES records the chosen optimizations using an ***optimized module list.*** An optimized module list has an initial structure similar to RVMDES's internal representation of the algorithm, but includes some information fi-om function definition headers. RVMDES can access the header information to determine what optimization applies. It can also modify the list to record and reflect the effects of optimization, so that RVMDES can search for additional optimizations or generate code.

1. The initial list

The initial optimized module records the software blocks and connections that comprise the algorithm, plus information fiom primitive definition headers to help RVMDES determine what optimizations it can apply to the algorithm. To create the list, RVMDES determines which modules will be on the list, adds them to the list, then marks whether each will be a prep or process module.

a. Choosing modules for the list

RVMDES builds the optimized module list fiom the software blocks likely to provide the most optimized version of the algorithm, not the ones actually present in the algorithm. This desired set of blocks is identified in a three step process of expanding macros, separating functions, and determining whether blocks are needed.

RVMDES begins by expanding any function blocks whose functions are defined as macros. This increases the chance that optimizations will apply to an algorithm, because macros cannot be optimized, but their component primitives often can. To locate macro blocks, RVMDES simply reads the macro flag in each function block's associated function's primitive definition header. When the flag indicates a macro definition, RVMDES creates an instance of the block diagram recorded in the macro definition, and splices it into the algorithm in place of the original function block. The expansion is recursive, so that even if a macro is defined in terms of macros, it will eventually be reduced to primitives.

With all macros removed, RVMDES attempts to separate any separable function blocks. This is the single most effective optimization applicable to image processing programs, reducing the order of operations fiom N^2 for a 2D window operation to $2N$ for a sequence of 1D window operations. This halves the run time of an operation using a 4x4 window, and does even better for larger windows. The optimization is so useful that RVMDES applies it automatically, rather than leaving it to the user's discretion as with other optimizations. Of course the user can disable this feature to prevent the optimization, for instance if memory but not run time is limited, because a separated operation requires extra memory to hold the intermediate data.

To separate blocks, RVMDES first searches for blocks whose associated primitive definition headers mark them as separable. If the function takes an image as its second input, RVMDES assumes that this is a convolution mask, and will separate the function only if

its mask is constant, and can be broken into horizontal and vertical components. To actually separate a block, RVMDES creates two function blocks whose functions are the horizontal and vertical components of the original block, named as the original function, with "_horiz" and "–vert" appended. The first output of one is connected to the first input to the other, separated masks or dimensions are attached to the second input of each, if the function uses a mask or variable window dimensions, and the whole lot is spliced into the algorithm in place of the original block.

With all software blocks now in their most basic form, **RVMDES** chooses which blocks will be represented on the optimized module list. Blocks that cannot affect an output or wraparound block represent wasted processing, and are left off the list. To decide which will be listed, RVMDES marks all software blocks as unnecessary, except wraparound and output blocks, which it marks as necessary and stores in a queue. While blocks remain on the queue, RVMDES enqueues and marks as necessary any blocks that provide inputs to the first element on the queue, then discards that first element. When the queue empties, the blocks marked as necessary represent the set of blocks needed to generate the most optimized version of the original algorithm.

b. Contents of the list

Once RVMDES determines which blocks are necessary, it creates an optimized module list, which records the structure of the algorithm and the information necessary to optimize and generate code for it. The list has one node for each necessary software block in the algorithm. To represent the structure of the algorithm, each node records a pointer to its software block, lists of the block's inputs and outputs, and information telling how each connects to other nodes' inputs and outputs. To help detect possible optimizations, each node also records the invertibility and pixelwiseness flags fiom the function definition header of the fimction represented by its block. Finally, for code generation purposes, each node records the position of the first pixelwise input or output in the fimction definition header (inputs are listed first), or -1 if no parameters are pixelwise. This variable will be used to determine the module's looping dimensions.

The only point worth additional mention is that the optimized module's invertibility and pixelwiseness flags are not always copied directly from the h c t i o n definition header. One reason is that the optimized module's flags are three-state. The invertibility flag can show ***default, inverted,*** and ***uninvertible,*** to indicate not only whether a module can be inverted, but also whether it has been. Any functions marked invertible in their function definition header are listed as ***default*** in their optimized module. Pixelwiseness flags also three state, the additional state being ***possibly-pixelwise.*** Pixelwise inputs of invertible fimctions take this state because they are not *pixelwise*, in the optimized module's default, uninverted state, but may become pixelwise if the module is inverted. A second reason why flags might not be copied directly from function definition headers is that optimized modules representing blocks other than function blocks do not have associated function definition headers from which to copy flags. To reflect the fact that only modules representing functions can be optimized, the flags of all non-function modules are set to not-invertible and not-pixelwise, the same flags used by function modules that are not eligible for any optimizations.

c. *The* prep *flag*

There is one additional field in an optimized module, whose value cannot be determined solely from the module's software block and primitive definition. This is the module's prep flag, marking it as a prep module or a *process* module. Prep modules contain constant blocks and any other blocks whose outputs do not vary as frames are processed. These blocks only need to execute once when the **RVM** boots, and again if the RVM operator sends new constant values fiom the RVM's interface. Process modules depend on an algorithm's inputs, so must run every frame. The optimized module list records whether each of its modules is prep or process, so that **RVMDES** can ignore "optimizations" that combine prep and process modules, forcing the prep module to execute during every frame, and so that it can warn users that optimizations involving prep modules are comparatively unimportant, since the modules run infrequently.

To separate prep and process modules, **RVMDES** uses a queue much as it did to determine which software blocks would become optimized module nodes. **RVMDES** begins by marking all optimized modules as prep, then listing and marking as process all blocks whose outputs must definitely be recalculated with each new frame. This includes input and wraparound modules, whose job is to provide new data for each frame, modules that read inputs fiom other hardware, because they must read the same set of data every frame, regardless of whether that data changes, and modules whose output variables are modified by later modules, and so must be reset. The list also includes modules providing inputs to wraparounds, which modify their input variables between frames. While this list of blocks is not empty, **RVMDES** pops the first element, and lists and marks as process any prep modules which read the first element's outputs. When the list is empty, all modules whose outputs change with each new fi-ame are marked process, while those that do not change remain prep.

Knowing which modules are prep and which are process would probably be useful during algorithm simulation, because prep module do not need to be simulated in each new kame. In fact, the software editor provides a display mode that labels software modules with 'P' for prep, and 'N' for normal (process), for just this reason. Unfortunately, prep/process status can only be determined after software is assigned to hardware, because a block that reads only constants may be process or prep, depending on whether those constants are partitioned onto the same hardware as the block, and **RVMDES** does not check that a partition is viable until immediately before making the optimized module list.

2. Finding optimizations

One purpose of the optimized module list is to enable RVMDES to determine which optimizations it can apply to an algorithm. This section explains how the **RVMDES** can determine the applicability of each of the optimizations it supports, using information stored at the optimized module nodes.

The first optimization **RVMDES** supports is serial combination. Two functions can be combined in serial if at least one input of one connects to an output of the other, and all inputs and outputs connecting the two are pixelwise. Further, **RVMDES** requires that the functions be both prep or both process, not a mixed pair, so that no prep modules need to

be executed during regular program execution. Each module on the optimized module list records enough information to let the user know whether it can participate in such a combination. First, it records what modules it reads inputs fiom, which is the set of modules it could theoretically combine with. Second, it records whether each of its inputs and outputs is pixelwise, so RVMDES can quickly test all connections between it and a connected module. Third, it records whether it is prep or process, so RVMDES can quickly ensure that both modules of a pair share have the same flag value. Thus RVMDES can quickly decide which modules can be optimized by serial combination.

The second optimization RVMDES supports is parallel combination. Two modules can be combined in parallel whenever the read from at least one common source, and all such common connections are read pixelwise by both. As with serial combination, RVMDES also requires that parallel combination only take place between two modules with the same prep flag. Also as with serial combination, RVMDES can determine which modules have the proper connectivity by searching each node's list of inputs and pixelwiseness of those inputs, and can check whether the prep flags of pairs of modules match.

The third optimization that RVMDES supports is inversion, where a function switches whether it loops over an input or an output. This is possible whenever a function's primitive definition says that it is. The optimized module list makes detection of inversion possibilities trivial, by including a copy of the invertibility flag at each node. If a module represents one function block, and its invertibility flag registers *default* or *inverted*, the module can be inverted or uninverted, respectively. If the module contains multiple functions blocks, as a result of combination operations, the module cannot be inverted. This is because there is only one set of inversions that will allow a set of functions to combine. Any attempt to invert one or more of the combined h c t i o n s would render the combination invalid.

In addition to the three directly supported optimizations, RVMDES supports a combine-and-invert optimization. This is applicable to pairs of modules that is not combinable, but would be if one or both were inverted. The optimized module list makes detection of such situations easy by marking the inputs of an invertible module and the outputs of an inverted, invertible module as possibly-pixelwise. If a pair of modules would be combinable except that their connected inputs or outputs are marked possibly-pixelwise instead of pixelwise, RVMDES can knows that it can invert the offending module(s), then combine them.

3. recording optimizations

The second purpose of the optimized module list is to keep track of the optimizations the user selects, so that RVMDES can see what optimizations apply to the optimized, optimized module list, and so that the code generator can see what optimizations have been applied. To support this, the structure of the optimized module is designed to reflect the results of combination and inversion optimizations.

a. Supports combination

To reflect combination optimizations, a single optimized module can represent an arbi-

trary number of combined software blocks.. When **RVMDES** combines two modules, it simply removes one fiom the optimized module list, and merges its data onto the first module's lists. All fields of the optimized module are designed with this merging in mind.

To update lists of function blocks and invertibility flags, **RVMDES** appends the removed module's lists to those of the remaining module. If the modules are being combined in serial, the downstream module will be the one removed from the list, and appending its function blocks to the function block list ensures that its loop kernels will be executed after those of the upstream module. Appending the invertibility flag keeps the two lists synchronized.

The lists of inputs and outputs for the module staying on the list are also updated by appending lists fiom the removed module, with three caveats. First, inputs common to the two modules are not duplicated on the input list. Second, variables connecting the two modules in serial are stripped fiom both lists. Third, any inputs or outputs marked ***possibly-pixelwise*** are relabelled ***not-pixelwise,*** to prevent **RVMDES** fiom trying to involve them in an invert-and-combine optimization. After all, a combined module cannot be inverted.

The remaining fields of the optimized module are the looping variable and the prep flag. In both cases, the module staying on the list can maintain its initial values of these fields. The looping variable remains valid because it is only needed for its dimensions, and combination optimizations do not alter loop dimensions. The prep flag can maintain its value because combination is only possible between two modules with the same flag.

Modules that have been combined in this manner are available for additional optimizations, because they keep the same invertibility and pixelwiseness fields used to detect optimizations in single-block modules. Further, the code generator can write a single function for a single module, regardless of how many blocks are contained in that module, since the block names are all listed in the module, and coding a set of combined blocks is just a matter of interleaving their fields.

b. Supporting inversion

To record inversion optimizations, the optimized module uses two fields: invertibility flags and pixelwiseness flags. **RVMDES** marks whether a module is inverted by toggling the former between ***default*** and ***inverted.*** It also toggles some of the latter, converting any variables marked pixelwise to possibly-pixelwise, and vice versa. This reflects the fact that only inputs or outputs can be pixelwise, but also allows **RVMDES** to distinguish variables that are currently not pixelwise due to inversion fiom those that are permanently not pixelwise, in case it must invert the module again. If an inversion is required as part of an invert-and-combine operation, the inversion is performed first, as described here, then the two modules are combined.

Modules containing inverted functions are also easily understood by the code generator, because the modules record the names of functions, and whether each should be inverted. This is enough to code the inversion optimization, which is just a matter of deciding which

of a function definition's two bodies to use. If inversions and combinations are present in the same module, inverted forms are used when writing the combined code, which again is straightforward for the code generator.

D. Conclusion

The special structure of image processing programs provides the potential for several software optimizations. Because the types and applicability of optimizations are predictable, the rules governing when and how to apply them *can* be coded into RVMDES. The information necessary to apply these rules is encoded in the connectivity of a block diagram algorithm and the specially formatted header of function definitions. This information is used to initialize an optimized module list, which is modified to reflect chosen optimizations. This list can be used to identify additional optimizations, or to determine how the specially formatted function definition bodies should be used to generate optimized code.

V. Automatic Code Generation

Chapter 5 - Code Generation
As now, but use main example when building process0
Can provide code generator for each type of processor.
The block diagram can be used to automatically generate code for processor modules, so the user need not write two copies of the algorithm.

RVMDES generates an executable for each processor module generated during hardware design. It writes processor independent C code to implement image processing functions and glue them together, adds processor dependent C code to handle data transfer, and compiles the whole program for the target processor. This method allows RVMDES to build programs for a variety of chips, using a single library of processor independent image processing routines and a set of compilers for its known set of processor modules.

Figure 5.1 shows a template for the code that RVMDES generates. First comes the declaration of global variables to represent the connections between software blocks, along with the functions needed to initialize them by transfemng data in and out of the processor. Second are definitions of procedures to implement each function block assigned to the hardware, as modified by any optimizations applied in the optimized module list. After these comes the definitions of a function to process an entire frame of the algorithm by calling procedures for function blocks and implementing the control flow described by branches and joins, and a function to prepare the processor for the first kame by initializing its global variables. Finally the main function executes these preparation and processing functions, surrounding the latter with an infinite loop to process consecutive games, and including code that prepares for and cleans up after each frame.

The information necessary *to* generate this code comes partly from the optimized module list, and partly from several other data structures generated with the optimized module list. This chapter begins by discussing what this data is, and how it was created. Succeeding sections explain how the data is used to generate each part of a program, including separate files that allocate memory for the program.

```
VARIABLE DECLARATIONS
PROCEDURE DEFINITIONS
PROCESS() DEFINITION
PREP() DEFINITION
void main () {
  prep();
  while (1) {
    LOOP SETUP;
    process();
    LOOP WRAPUP;
    }
  }
```

figure 5.1 - overview of C code

A. Required information

RVMDES calculates several pieces of information while or before creating the optimized module list. These items were omitted from the preceding chapter because they had no bearing on optimization, but they will be needed for code generation, so they are described here. The pieces of information consist of a local, optimized module list for each hardware module, information about each module's hardware port usage, and dimensions of the variables represented by connections between software modules.

1. Local optimized module list

The principal source of information needed to write code for a hardware module is the set of optimized modules representing software blocks partitioned onto that hardware. The optimized module list contains this set, but also contains a lot of other modules, making it cumbersome. To limit the list to relevant modules, each hardware module maintains a list of the optimized modules partitioned onto it. **RVMDES** generates this list as it builds the initial optimized module list, and removes any modules fiom the sub-list as they are removed fiom the main list by combination optimizations. This double-list scheme is useful because the complete list allows RVMDES to notice potential optimizations that span hardware modules, while individual lists provide the exact set of optimized modules used for code generation on a given processor.

2. Port information

As RVMDES makes optimized modules and assigns them to their two lists, it also notices connections between optimized modules partitioned onto different hardware modules, and records them on lists of inputs and outputs for those hardware modules. At the same time, it assigns the numbers of the hardware ports that the two modules will use to transfer the data. The mapping between input/output variables and the ports they use will be needed to

write the code that transfers these variables onto and off of the hardware. It will also be used to determine the required connections between hardware when RVMDES assigns hardware modules to motherboard slots. Should RVMDES fail to find hardware ports that can make the connection, it Will abort its attempt to make the optimized module list, reporting the problem and requiring the user to repartition software onto hardware so that fewer ports will be needed.

In general, finding ports to connect processors is straightforward. If the modules already share a connection, the old connection's ports are reused, rather than risk running out of ports or complicating circuit board design by connecting a second set. If on the other hand a new connection is between unconnected processor modules, RVMDES locates unused ports on both modules, and records for each the hardware module and port to which they attach. Of course it also records the port numbers with the input and output that use them.

Finding ports for custom hardware modules will be easier but more failure prone, because the modules will require that certain inputs and outputs be assigned to certain ports. To accommodate this, RVMDES maintains an internal mapping for each hardware module type, showing which connections to which of its supported software blocks must use which ports. This makes the choice of ports trivial, but potentially too restricting, in two ways. First, a single hardware output port can only connect to one hardware input port, so if a hardware module specifies its output ports, only one module can read that output. If the user did not partition all software that uses that output onto one hardware module, port assignment will fail. Second, one hardware input port can only connect to one hardware output port, so if a hardware module specifies its output ports and sends two outputs one hardware module, the latter module must waste two input ports.

3. Variable dimensions

Prior to creating the optimized module list, RVMDES calculates the dimensions of variables represented by connections in the algorithm block diagram. Each output port of each software block is the source of one such variable, and RVMDES stores the dimensions of each port's variable in a data structure associated with the port. RVMDES uses these dimensions to check for errors in the algorithm, prior to creating the optimized module list. Such errors could include variables with negative dimension, two dimensional variables leading to input ports that only accept scalars, or branch modules whose input bundles do not all share the same set of dimensions.

To begin finding variable dimensions, RVMDES copies the user defined dimensions of input and constant blocks to the blocks' output ports. The output dimensions of a wraparound block's single output match those of its first input, which must be connected to a constant, whose dimensions were just calculated. Therefore, RVMDES fills in the output port dimensions of wraparounds. Next, it makes a list of the blocks whose output port dimensions must still be set -- all branch, join and function blocks. RVMDES then loops repeatedly through the blocks on the list, finding output dimensions for any blocks that it can and removing those blocks fiom the list, until after several passes all blocks have their output port dimensions recorded.

Whether and how to calculate output port dimensions for a given block varies with the type of block. A branch block's dimensions can be calculated if its input ports all connect to output ports of other blocks whose dimensions have already been calculated. In this case, the set of dimensions for each input port is copied to the corresponding port in each output bundle. A join block's dimensions can be set if at least one input bundle has all members connected to output ports with calculated dimensions. The dimensions of those ports are then transferred to the join's output ports. A function block's dimensions can be calculated whenever all of the block's input ports connect to output ports whose dimensions have already been calculated. However, the process of converting to input dimensions to output dimensions involves more than copying.

To convert a function block's input dimensions to output dimensions, RVMDES uses formulas provided by the block's function definition header. This header includes, along with the names and order of input and output parameters, symbolic dimensions for those parameters. Input dimensions are represented by single tokens, while output dimensions can be formulas involving input dimension tokens and constants. Basically, RVMDES builds a lookup table pairing the symbolic input dimensions with the actual dimensions stored in output ports connected to the block's input ports, then uses that table to substitute into formulas for output dimensions, providing actual values for each output dimension.

The only complication is that some input parameters may have the same symbolic definitions, but RVMDES only records one table entry for each input dimension symbol. Thus, each time RVMDES finds a repeated symbol is encountered, it checks the new dimension's value against the one already on the table. If the two values agree, the new one is discarded. If they disagree and neither value is 1, RVMDES notifies the user, and will abort its variable dimension finding and the ensuing optimized module list generation. If the two values disagree but either is 1, RVMDES records the other value. This feature allows the functions to operate on inputs of reduced dimensions, for instance adding a scalar to an image instead of adding two images together, or finding the average value of an array instead of an image. The output size will be calculated based on the input with the highest number of dimensions.

B. Declaring and initializing variables

The first pieces of code **RVMDES** writes into a program declare global variables and provide some simple routines needed to fill them. **RVMDES** begins by declaring variables to represent most, but not all, of the connections between software blocks assigned to the hardware module. Then it writes a command interpreter that, with the help of a parameter table, allows the program to read new initialization values for initializing constant blocks fkom an **RVM** operator. After this, **RVMDES** writes processor specific code needed to transfer data in and out of the processor to begin and end a frame of processing.

1. Globals and Constants

The first task in code generation is to declare variables corresponding to the connections in the block diagram. This task is complicated because not all connections will be declared as variables, and of those that are, some will be arrays, and some will be pointers,

with space allocated later. This section explains which variables are declared, then why and how the variables are declared in two different ways.

a. making the variable list

RVMDES first makes a list of the variables it will declare. It considers each connection to and fiom every software block partitioned onto the hardware module being coded, and decides whether the connection warrants being represented by a variable. Normally it will, and RVMDES Will add an element to a list of variables, recording a unique variable name and the identity of the output port that spawns the connection. RVMDES will also record the identity of the new list element in the data structure representing that output port. However, there are three situations in which a connection does not represent a vanable, and RVMDES must behave differently.

The first situation is when a variable is an accumulator -- a variable that is input, modified and output by a function block, rather than being created from scratch by it. RVMDES does not create a new variable node for an accumulator, because each variable should only be represented by one node, and the software block that originally writes the variable will make that node. Instead of making a new node, RVMDES follows the input port that reads the variable to find the output port that supplies it, and copies the identity of the variable node recorded there. Should that output not have a variable node assigned yet, RVMDES creates one and stores its identity there- This process is recursive, such that if a variable is modified by a chain of functions using it as an accumulator, RVMDES will follow to the top of the chain before creating the node, then record the node's identity at each output port in the chain. To determine whether an output port represents an accumulator, RVMDES checks whether the name of the parameter for that port, on the function block's function definition header, also appears as an input to that function. Also, it checks whether the port is on a block partitioned onto the hardware being coded, because if not, the output represents the first instance of the variable on the hardware, and can be treated as a non-accumulator.

The second situation is when an output port is attached to a wraparound, branch or join module. In theory, each copies data from its input(s) to its output(s) when executed. To save time however, RVMDES reuses input variables as outputs. For wraparounds, this means the output and first input form an accumulator, and can be handled as one. For branch blocks, there is only one set of variables, shared between the various bundles, so each output port forms an accumulator with its corresponding port in the input bundle. RVMDES handles these ports as accumulators, but as it finds the identity of a variable feeding the input bundle, it records that identity in each output bundle. Join blocks also share one set of variables between their several bundles, but are more difficult to work with because several input ports must all share the same variable. To accommodate this, RVMDES will not create a variable node for an output port whose connection leads to a join module on the same hardware, either directly or through a branch block or accumulator. Instead, it will use the variable node stored in the proper output port of that join block, or make a new one there and then record it, just as was done with accumulators. The only exception is that the output port of a join block can produce a variable node, even if its connections lead back to the same join block, as long as they do not lead to any other join

blocks.

The third situation is when an output corresponds to a localized variable -- one that is read and written only by software blocks assigned to the same optimized module. Such variables must still have variable nodes, for use in writing procedures for their optimized modules, but these nodes are attached to a second list of variables, which will not be declared as variables.

b. Declaring Variables

Having generated a list of variables, RVMDES must declare them in C. This is somewhat complicated, because even with the memory reducing optimizations from chapter 4, the combined size of all of the variables (many of which will be images) often exceeds the memory capacity of the hardware module. Fortunately, each variable is typically only used during part of a frame, so several variables can often use the same memory, at different times. However, not all variables can afford to have their space reused, so RVMDES must identify whether each variable can be doubled up like this, and declare it appropriately. Those that can be recycled will be declared as globals, and those that cannot will be constants.

RVMDES determines a variable's recyclability as it adds the variable to the variable list. If a software block's optimized module is marked 'prep', variables representing its outputs are marked as nonrecyclable. This is because 'prep' modules only execute when the hardware boots or the operator sends commands in between frames, and if their outputs are overwritten during one frame, they will not be reset for the next frame. Conversely, modules marked 'process' run before their outputs are needed in any frame, overwriting any evidence of recycling. Outputs of these modules are marked as recyclable, unless they are scalars, which are not worth the effort of recycling. In addition, any variables produced by software blocks partitioned to different hardware are marked as recyclable, since they must reloaded for every frame, regardless of whether their host block runs.

With the variables marked, RVMDES can declare each variable on its variable list, as a constant or a global. Constants are declared as scalars, arrays or matrices, using a variable node's name, and the data type and dimensions stored in the output port pointed to by the variable node. If the constant comes from an initialized constant module, the output port also records the initial value(s) for the variable, which are included in the declaration. A typical declaration may read "int mask[3][3] = { 1, 2, 1, 0, 0, 0, -1, -2, -1 };". Globals are declared as one dimensional pointers, with no initial values. This is necessary so that they can be scheduled into memory later, once the amount of memory available to hold them is determined. As with constants, the name and data type of a global are read from the variable node and its associated output port-

In addition to these variables, there are a few more declarations which must be made in this section. In particular, a declaration of the form "extern void mem_init_X();", for each bank of memory X in the hardware (specified in the structure pointed to by the hardware's data structure's definition field). These functions will allocate the space required for the global variables, and initialize those variables to point into the space, before the first pass

through the algorithm. The contents of these functions will be described in section G.

2. Parameter Table and Command Interpreter

Sometimes the user is unable to provide the best value(s) for a constant when he builds a program, for instance because they depend on unknown lighting conditions, there are insufficient test images to predict the behavior of certain values, or the constant is actually an image taken at run time. In all of these cases, you can use initializing constant modules to specify that the value should be sent at run time, and should be available to the user to change, in the MMI. Having promised the user this, the program running on the hardware must be able to read commands telling it to reset the values of these constants, then store the new values in the constants. This functionality is provided by the parameter table and the command interpreter.

The parameter table actually consists of three arrays -- parameter_table, parameter–lengths and parameterqrep -- which list the addresses and dimensions of variables representing initializing constant modules, and tell whether those variables are globals or constants. Parameter–table is simply declared, because it will need to be filled at run time, after any global variables it contains have been assigned addresses. Parameter_lengths is declared and filled, each entry receiving the length, in words, of the variable it represents. This dimension is calculated from the dimension and data type of each of the initializing constants. Similarly, parameterqrep is declared and filled, by matching each initializing constant with a variable on the variable list, to decide whether it is a global or constant.

The command interpreter is assigned the task of listening for and acting on commands to change constant values. The first word of such a command is a parameter number, which the interpreter uses to index into the three parameter tables. From this, it knows how much additional data to read, and where it should begin storing that data. Further, it knows whether it has read a constant that was marked 'prep'. If so, it will cause the program to reexecute any function modules marked 'prep', to account for the new values. The command interpreter reads all of its information from a register attached to the RVM's VME bus, and polls another register to see if any commands are available. It will continue to interpret commands until none are available.

3. DMA

One of the few processor dependent sections of the C code is the declaration of variables and hctions involved in moving data in and out of the hardware. In the case of the C44 module, the only processor module available so far, this involves setting up and using the DMA. This section explains the various pieces of the DMA related code: DMA blocks, interrupt service routines, the setup_dma() function, and the dma_in() and dma_out() functions

A DMA block holds the 5 values that must be loaded into a DMA channel to transfer a block of data between a comport and memory: memory location, step size, control register, data length, and link pointer. **RVMDES** creates one of these blocks for each input to and output from the hardware, setting the 5 values appropriately for each transfer. Mem-

ory location is set to 0 and reassigned in setup_dma(), because most of the variables being read or written are globals, whose addresses are not determined until run time. Step value indicates how far the read/write pointer should advance in memory after each transfer, and is set to 1. The control register has its bits set to indicate that it should transfer data between a comport and memory, which comport it should use, whether to transfer to or fiom memory, and that it should begin each transfer by loading the **DMA** block pointed to by its link pointer and end each transfer by sending an interrupt signal. The values of the remaining two fields (data length and link pointer) depend on whether a variable occupies a single block in memory. When it does, the block is moved in one transfer, so the data length is set to the total size of the variable, and the link pointer is set to point to the next **DMA** block that uses the same channel, or zero if this is the last block in the channel. Otherwise, when a variable is part of a larger parent image (when the number of columns in the variable and its parent differ), RVMDES will transfer one line at a time, so the data length is set to the length of one line, and the link pointer addresses the block itself, so the block will repeat to transfer one line at a time. Incidentally, these blocks are declared once as arrays, then a second time as initialized arrays. This insures that when they are used to initialize link pointers on the second declaration, the compiler will recognized their names fiom the first declaration.

The second piece to declare is the interrupt service routines (ISRs), which **RVMDES** must do for each **DMA** channel the program will use. Since **DMA** channels correspond to port numbers, and all hardware inputs and outputs have their port number recorded, RVMDES can simply loop through the set of ports, declaring an ISR for any port that is used by at least one input or output. The declaration takes the form "c_int0X () { reset–iif_flag (X); };", where X is the port number, and the strange function name has special meaning to the compiler.

The third piece of code is the setup_dma() function, which will be run prior to the first pass through the algorithm, but after variables are allocated, to prepare the dma blocks and ISRs. To prepare the DMA blocks, it fills their memory-location fields in a series of assignment statements matching variable names to the **DMA** blocks that read or write them. These variable names were not used in the **DMA** block declarations, because they are evaluated at compile time, before the variables point allocated memory. However, assignment statements in setup_dma() are evaluated at run time, after the variables have been allocated. Once the DMA blocks are ready, RVMDES prepares the ISRs by installing them and enabling interrupts for each port used by the hardware. To decide which ports are affected by this step, RVMDES records which ports require ISRs as it generates them, and uses that same list here.

The final piece of the **DMA** related code consists of the dma_in() and dma_out() functions, which read all inputs before, and write all outputs after each pass through the algorithm. Since they are structurally the same, I will only explain dma in(). You can follow along in the example code in figure 5.2. The function begins by initializing the **DMA** registers for all ports that will read inputs, which were listed previously. Initialization means clearing the control and transfer counter (data length) registers, and setting the link pointer to the first **DMA** block that uses the port, so that when the DMA a started by changing the

control register, it will load the DMA block and begin the first transfer. The remainder of dma_in() consists of code segments to read each input variable. If a variable's DMA block loads all of the variable's data, the its code segment starts the transfer by copying the DMA block's control register value into the DMA's control register, then idles the processor until an interrupt indicates that the transfer is done. If the DMA block loads one line at a time, the code segment uses a loop to read each line. The loop begins a transfer, idles until it finishes, then moves the memory pointer forward one line. After the loop, the DMA block's memory location is reset for the next frame, and the channel's link pointer is set to the next DMA block using that channel. In the example, there is no next DMA block, so the link pointer is set (arbitrarily) to 0.

```
/* initialization */
 *(dma_ctl_reg(P)) = 0;
 *(dma_in_xfer(P)) = 0;
 *(dma_in_link(P)) = dma_inC;

/* single block transfer */
 *(dma_ctl(P)) = dma_inC[0];
 CPU_IDLE();

/* line by line transfer */
for (ctr = Y; ctr; ctr--) {
  *(dma_ctl(P)) = dma_inD[0];
  CPU_IDLE();
  *(dma_in_mem(P)) += X;
  }
*(dma_in_mem(P)) = X*Y;
*(dma_in_link(P)) = 0;
```

figure 5.2 - an example dma_in(), initializing one channel, then reading one contiguous and one noncontiguous variable

C. Image processing procedures

Once RVMDES has declared a program's variables, it defines the procedures that operate on those variables. RVMDES writes one such procedure for each optimized module on the hardware module's local optimized module list whose list of software blocks contains one or more function blocks. The function definitions for these function blocks can be interleaved, inverted and transcribed, as described earlier, to quickly generate C code. However, generating ***compilable*** code is slightly more complicated than that. It requires adding a few pieces of supporting code, and replacing instances of several variable names and commands.

1. Function template

To generate a complete procedure, RVMDES inserts the interleaved fields of an optimized module's function blocks' function definitions into a template, shown in figure 5.3. In addition to the four fields filled from function definitions, shown in capitals, the template contains three fields that must be filled fiom other parts of the optimized module: the procedure header, the loop statement and counters, and the localized variables.

```
procedure header [e.g. void module_1 ()] {
    loop counters [e.g. int x, y];
    localized variables;
    VARIABLES;
    INITIALIZATIONS;
    loop statement [e.g. for (y=0; y<Y; y++) for (x=0; x<X; x++)] {
        KERNEL;
        }
    POST-PROCESSING;
    }
```

figure 5.3 - the procedure template

a. procedure header

The procedure header consists of a name, a return value type and a parameter list. RVMDES names its procedures "module_X", where 'X' is the position of the procedure's optimized module in the local optimized module list. This guarantees a unique name for each procedure, and makes it easy to recreate the name when writing the code that calls the procedure. A procedure's return type and parameter list are even more straightforward, being void and an empty list, respectively. Globals variables were already declared to hold any data passed between procedures, and they do not need to be redeclared as parameters.

b. loop statement and counter variables

A loop statement and braces wrap the procedure's kernel, to apply the kernel to each pixel in the optimized module's looping variable, and declarations of the counters needed for the loop form the first statement in the procedure's code. The exact contents of these pieces of code depends on the number of dimensions of the looping variable. Typically this variable is an image, requiring the a dimensional nested loop and corresponding loop counter declarations as shown in brackets in figure 5.3. If the looping variable is a one dimensional array instead, the loop over and declaration of the counter 'x' are omitted. If the variable is a scalar or there is no looping variable, the loop counter declarations, loop statements and the braces around the kernel are all omitted.

To determine the dimensions of the looping variable, RVMDES must locate the software block output port that Writes that variable. RVMDES stored the dimensions there when it calculated all variable dimensions prior to building the optimized module list. The first block listed in the optimized module will be the block from which the variable was drawn, since any other blocks in the module will have been appended as a result of combinations, after the looping variable was fixed. To find the appropriate port, RVMDES consults the looping variable field of the optimized module, which recorded the number of a parameter in the block's function definition header corresponding the actual looping variable. Inputs are declared first in headers, so if the looping variable value is less than the number of input parameters, the value represents the input port number that reads the variable. RVMDES can follow the connection leading back from that port to discover the proper output port and the variable dimensions stored there. If the looping variable value is larger

than the block's number of inputs, the difference gives the output port number wherein the dimensions are stored. Finally, a looping variable value of -1 means the function does not loop, so its dimensions are both listed as 1.

c. localized variables

Localized variables are those that originally connected two modules, and were only used by those two modules, so were eliminated as parameters when the modules were combined in series. Such variables are declared here because they are not declared as globals but are used inside the procedure. They are declared as scalars, no matter their original dimensions. A localized variable only uses one pixel in any iteration of the procedure's loop, as required for serial combination, and the pixels are not needed after the loop, so the same pixel can be reused in each iteration, saving memory. RVMDES identifies localized variables by searching the output ports of software blocks listed in the optimized module, reporting any that connect only to other blocks in the same optimized module. From these output ports, RVMDES can read the names and data types of the localized variables.

2. Variable substitution

In addition to writing several pieces of supporting code, RVMDES must modify references to variables in function definitions as it interleaves their fields into the templates. These modifications include renaming local variables, replacing parameters with global variables names, and modifying the dimensions of global and localized variables.

a. local variable names

RVMDES renames the local variables in function definitions when it transcribes them into the template because many function definitions use the same local variable names (e.g. "sum" or "counter"), which would conflict if their functions were combined into a single loop. This is only a problem when a module contains multiple software blocks, but replacing local variables in every function is simpler than handling single- and multiple-function modules separately. To effect the translation, RVMDES first creates a lookup table for each function block in an optimized module, pairing each function definition's local variable names, parsed fiom its local variable field, to variable names that are unique within the optimized module. RVMDES uses each function's table as it transcribes the function's fields into the template, replacing any local variable names with the new, unique names.

b. global variable names

RVMDES also replaces any references to a function definition's parameters and their dimensions with the names and dimensions of the global variables those parameters represent. To do this, it extends the lookup tables created for local variables, and replaces the local variables and parameters in a single pass. To find the parameter pairs, RVMDES again considers each software block, recording parameter names and symbolic dimensions from its function definition header, and global variable names and dimensions fiom output ports connected to the block's input ports. As when calculating each software block's dimensions, RVMDES does not record duplicate dimensions names unless they conflict with previous ones, where they either replace dimensions of value 1, or cause an error. Also as earlier, only input dimensions are actually recorded, since output dimensions must be copies of or formulas involving them.

c. reduced dimensions

To facilitate memory allocation, RVMDES declares its non-scalar global variables as one dimensional arrays, so it must address them as such. However, function definitions that use images address them with two dimensional array notation, so RVMDES reduces instances of this notation to one dimensional addressing as it transcribes function definition fields into the template. To implement this, RVMDES checks for two sets of brackets following any parameters that it will replace with global variable names. It then replaces the pattern *param*[*ycoord*][*xcoord*] with *global*[(*ycoord*)**cols*+*xcoord*], where ***cols*** is the number of columns in global, and *global* is the translation for *param*.

The number of columns in an image is the same the image's x-dimension, except in two situations. First, if an image is a subimage (a situation not currently supported), the number of columns will be the width of the parent image. Second, if a function is inverted, the formula for calculating the number of columns for an output image is the opposite of that for calculating the image's width. For instance, if an output variable had a symbolic dimension Y-2, based on input dimension Y, the number of columns needed in the inverted form would be *Y+2*. The discrepancy reflects the fact that a border around the output image is filled with garbage, as a side effect of the inverted operation.

In addition to reducing array addressing to one dimension, addressing is deleted in some cases. This occurs when the number of dimensions of a global variable is less than expected by the corresponding parameter. Thus if a two dimensional parameter is filled by an array or scalar variable, one or both sets of addressing coordinates are deleted, to make the addressing in the definition agree with the dimensionality of the variable. Localized variables have their dimensions eliminated in this way, because they are considered scalars.

3. Iotramodule optimizations

Like the elements of most image processing libraries, RVMDES's function definitions are optimized as much as possible. For example, they work with register variables then write to globals once calculation is done, and they use multiplication by a float constant instead of dividing by an integer constant wherever possible. However, they also are built to be general, for instance allowing inputs of arbitrary dimension, size and data type, which necessarily limits the amount of optimization possible. RVMDES attempts to replace any optimizations lost in this way by optimizing functions once their inputs have been set and they have been transcribed into the template. This optimization consists of two macros that RVMDES expands, and a few optimizations it leaves to the compiler.

a. the *foru* macro

The foru macro is similar to the *for* command in C, but indicates the presence of a loop that should be unrolled. The compiler used by RVMDES will only unroll small loops, presumably because the code size suffers from large unrolled loops. However, in image processing programs, code size is usually insignificant compared to data size, and the small increase from unrolling a few loops will not even be noticeable. Further, unrolling a function's internal loops allows the compiler to use the C44's single no-overhead looping

register to implement the loop over each pixel, saving a few cycles at each pixel. In addition, an internal loop that reads data fkom a mask can often be written to include that data into its instructions, if the data is hard coded into the statements of an unrolled loop. This decreases memory accesses and corresponding pipeline delays during execution.

b. the typeof macro

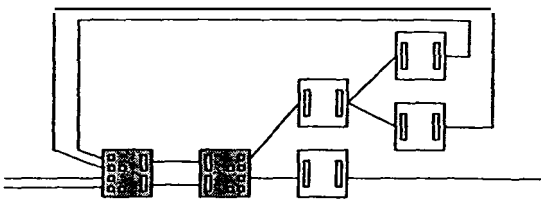
The data type of a function's inputs is determined by the functions that produce them, and its output data types can be set by the user fkom the software editor. This is a good thing, because it allows a single function to operate on several types of data. Unfortunately, the best choice of local variable types depends on the input data types, which are unknown when the function definition is written. If ail inputs and outputs are integers, local variables should generally be integers, to avoid slow type casts, but if any variables are floating point, local variables should also be floating point, to avoid losing precision. To solve this dilemma, RVMDES includes the typeof macro, which takes a parameter as its argument, and returns the data type of that parameter. RVMDES makes this substitution as it transcribes local variable statements into the template, allowing a definition to change to optimize itself around its parameter data types.

c. ***compiler optimizations***

When RVMDES applies unrolling or combination optimizations, it creates the potential for additional optimizations, which it should also apply. However, they are the small, localized kinds of optimizations that a compiler can and does apply, so RVMDES simply ignores them. They are only mentioned here for completeness. The first optimization is to condense superfluous math resulting from unrolled loops. These loops often perform dot products, and the unrolled versions often `sum` terms that are multiplied by 1, `-1` or 0. all three multiplications can be eliminated, and in the latter two cases, the summing can be replaced with subtraction or eliminated altogether. The second optimization is to remove superfluous assignments. In a combined fun quantity, from a register variable to a localized image, back to another register variable, as both functions expected to transfer the data through a global variable, which became localized but never went away. Both assignments are unnecessary, but again this is small enough for the compiler to notice and remove, so RVMDES does not bother.

D. Process()

In the third code generation step, RVMDES implements the heart of an algorithm -- its fimction, branch and join modules -- in a function named process(). This function is a direct translation of a block diagram into C code, consisting of calls to the functions described above (representing function modules), glued together with labels and "goto" commands (replacing join and branch modules). While using gotos may obfuscate the program structure, it facilitates translation, and the direct mapping to a block diagram should make the program interpretable. This section explains how RVMDES writes process(), in particular how it translates joins, branches and flows, and how it sequences them, following the example translation in figure 5.4.



join branch functions

figure 5.4 - the part of a program partitioned onto a processor, and its translation into C code

```
process () {

  goto branch_1 ;

  flow_1:
  combo_1();
  combo–20;
  combo–30;
  goto branch_1 ;

  flow–2:
  combo–40;
  return;

  branch_1 :
  if (in2>3) goto flow_1;
  goto flow_2;

  }
```

A join module simply indicates which branch module or set of functions should be executed next. It does not even copy data fiom input to output bundles, as in simulation, because the bundles all share the same data space. Given that, there are three possible translations for a join module. First, if all outputs of the join leave the hardware module, the join translates to "return;", ending the processing on that hardware. Second, if at least one output leads to a function or output module on the same hardware, the join translates to "goto flow–X", where 'X' is the flow number of that fimction or output, determined during software error checking. Third, if all outputs lead to a single branch module on the same hardware, as in figure 5.4, the join translates into "goto branch–X;", where 'X' uniquely identifies the branch module. The only other situation which would survive error checking is if all outputs lead to second join module. In this case, RVMDES simply follows the latter module's outputs, repeating this until it finds outputs that cue one of the three translations above.

A branch modules is more complex, performing several tests, and directing execution to a new part of the program associated with the first valid test. Therefore, it translates to a series of conditional branches of the form "if (TEST) *JUMP",* one for each output bundle. The tests strings *are* provided by the user in the branch module's specification window in the software editor, except for the final test which always evaluates true. RVMDES simply substitutes variable names as in part B, and inserts the test strings. The associated jumps are determined by following the outputs of each bundle, exactly as was done for join modules. In addition, when branch modules read all of their inputs from a single branch or join module, their translation begins with a label "branch–X:", where 'X' uniquely identifies the branch module. This allows it to be addressed by the upstream branch or join, using a goto command. In figure 5.4 the single branch module is translated

as the last piece of code, with a label, conditional test, then the final, unconditional one.

A flow consists of a network of function modules, reading its inputs and writing its outputs to a branch or join module, or another piece of hardware. Therefore, it is translated into a label, a series of calls to functions, and a jump to the next flow. The label takes the form "flow_X", where X is the flow number determined in software error checking. The function calls invoke functions described in part B, in the execution order determined during hardware error checking. Actually, functions representing optimized modules marked 'prep' are not called here, because they will be called later in prep(). The jump at the end of the flow depends on the destination of the flow's outputs. It can be a goto (implementing a join), a return (for the output flow) or a set of conditional gotos (implementing a branch). The only interesting point is that a trailing branch will not be labelled, because it will only be used by this flow. Figure 5.4 shows two flows, one leading to a branch, and the other being the output flow.

The order in which the translations of these branches, joins and flows are sequenced is as straightforward as the translations themselves. If all inputs to the hardware flow into a branch or join module, as in the example, the translation begins with the goto statement that represents a join, or a "goto branch_X" statement to direct execution to that branch. Otherwise it begins with whichever flow reads inputs for the hardware. Next, the remaining flows are translated, in the arbitrary order determined in hardware error checking. Finally, any branch modules which are not the endpoints of flows are appended. This may not be the most efficient ordering for code, because it does not minimize branching, but time spent branching is minute compared to processing time in image processing applications, and this order is the easiest to produce.

E. Prep()

A function called prep() is charged with setting up any data required to run the algorithm, and is implemented as shown in figure 5.5. The first time prep() runs, it allocates space for global variables, records variables for initializing constants in the parameter table, reads values for those variables, then sets up DMA channels to read inputs and write outputs. Every time prep0 runs, it executes modules marked 'prep' during hardware error checking. This setup allows prep() to rewrite any variables that depend on the values of initializing constants, whenever those constants are changed, without redoing initializations or separating the preparation sections. The paragraphs of this section explain each of the labelled lines in 5.5.

```
void prep () {
  static int first-1;
  if (first) {
    first=0;
    INITIALIZE MEMORY          (1)
    FILL PARAMETER TABLE       (2)
    READ CONSTANTS             (3)
    SETUP DMA                  (4)
    }
  PREP MODULES                 (5)
  }
```

figure 5.5 - implementation of prep()

Memory initialization consists of calls to mem_init_X(), for each bank of memory X on the hardware. As in section A, this number can be looked up in the hardware's definition structure. Their job is to allocate space to hold globals, and to point the globals into that space. They are declared external to the C file, and will be covered in section F.

The parameter table, discussed earlier, contains the addresses of all initializing constants, which are not necessarily known until after the globals are initialized. Once they are initialized, prep() inserts the addresses with statements of the form "param_table[x] = var_name".

With the parameter table full, the command interpreter is invoked to read the values of initializing constants. The MMI will send all of the parameters for a hardware module immediately following its boot code, so the command interpreter will read all commands in sequence, loading all of its initializing constants.

The DMA setup is the most complicated part of initialization, because it is processor dependent. **RVMDES** is only set up to handle DMA for the C44 processor. Basically, setup consists of finishing anything left unfinished in the declarations. This includes installing interrupt vectors, setting interrupt enable registers, and adding source and destination addresses to the DMA control blocks, just as they were assigned to the parameter table.

Finally, prep modules are called, just as regular modules were called in process(). They were declared in part B along with all of the other modules, and they are all occur in the starting flow for a hardware module, so there are no control flow issues. Further, they are declared in execution order, so RVMDES can simply go down the list of modules, calling any that are marked prep.

F. The main loop

The main function, process(), has two shortcomings. First, it only handles function, branch and join modules, and second it only handles one fiame. Thus it is embedded in a main loop, depicted in figure 5.6, padded with functions to implement input, output, constant and wraparound modules, and looping to process frame after fi-ame until the RVM is turned off. This section describes the padding functions, executed before and after pro-

cess.

The setup functions, executed at the beginning of each frame, are in charge of producing any inputs needed form the frame. The call to get–commands0 invokes the command interpreter, to read any new initializing constant values sent by the user, and to rerun prep() if necessary. In general, no new values will be available, and the interpreter will immediately return. Next the call to dma_in() reads all inputs to the hardware, using processor dependent code described in section A3. The third step clears and starts timers on the processor, if they are used in the tests of branch modules. Typically they will be used in programs that iterate to increase the precision of answers, but run under time constraints so that iteration must be stopped after a certain time. If the timers are not mentioned in any branch tests, they are not mentioned here either, so the step may be missing.

The wrapup functions, executed at the end of each frame, are in charge of disposing of any outputs produced by the fiame. Mainly this involves calling dma–out(), declared in conjunction with dma_in(), to send any hardware outputs on their way. However, it also includes swapping pointers to the inputs of wraparound modules, so that the second input in the current frame becomes the first input in the next frame. Because most inputs to wraparounds are declared as global variables, this simply involves a 3-line pointer swap. The exception is when the inputs are scalars, in which case the values are swapped with a similar 3 lines, concluding the processing for a frame.

```
while (1) {
 get_commands();
 dma_in();          } LOOP SETUP
 START TIMERS
 process();
 dma_out();         } LOOP WRAPUP
 swap–wraparounds();
 }
```

figure 5.6 - the *main loop*

G. Additional files

Section A explained that to fit all of a program's variables into the memory available on a processor, it is often necessary to use the same space to hold multiple variables. To facilitate this, it decided that some variables were recyclable, declared them as pointers, and promised to allocate space for them in functions called mem_init_X(), for each bank of memory X on the processor. This section explains how RVMDES makes those files, covering how it determines when variables are active and how much memory is available, how it schedules the variables into that memory, how it translates the schedules into memory allocation routines, and in retrospect, why it was necessary to declare these recyclable variables as pointers.

1. Scheduling preliminaries

The first step in scheduling is to determine when variables are active, which is actually carried out as the variables are initially added to the variable list. The information is created and stored in an array attached to each variable node. Each node on the array represents an optimized module, and is marked TRUE or FALSE to indicate whether the variable is active while that module executes. The array is organized by flow, and by execution order within each flow, so the optimized modules appear in the same order on the array and in the optimized module list. To fill the array, RVMDES initially marks all entries false. Next, a stretch of entries in each flow that uses the variable is marked true. If the variable is created by a module in the flow, the entry for that module begins the stretch. If on the other hand the variable enters the flow as a constant or hardware input, or through a branch or join module, the first entry in the flow begins the stretch, to show that the variable is active fi-om the beginning of the flow. Similarly, the stretch ends with the final entry in the flow if the variable leaves the flow as a hardware output or through a branch or join module, and ends with the final optimized module that reads the variable if it does not leave the flow. Following the variable along these paths is not so bad, because it must already be done to see if the variable should be declared. The array is simply marked as the search progresses. Incidentally, constant blocks that feed wraparound blocks are marked as being created during the first optimized module of the input flow, no matter which flow they are partitioned to, because they will be filled with data prior to the input flow, and should not be reused until after their subsequent use in a wraparound.

In addition to recording the activity of each variable, RVMDES also records the size of each variable, then adds the variable to the list in order of decreasing size. The size can be calculated from the dimensions of the variable and the size of its elements, based on its data type. The nodes are ordered by decreasing size so that as they are scheduled, the larger and therefore harder to schedule variables will be scheduled first.

Once all variables are accounted for, RVMDES determines how much memory is available for them. The hardware dictionary tells the number and size of memory banks for each type of processor, as well as which bank will hold the executable code, which provides an initial indication of the memory available. RVMDES compiles the C code into an object file, to see how much space the code will take, and subtracts that from the appropriate memory bank. The mem_init_X() functions, yet undeclared, take a predictable amount-of space, so this is also subtracted, yielding the actual amount of memory available for scheduling in each bank.

2. Scheduling

RVMDES tries two ways to schedule the variables of the variable list into the various memory banks. Should they both fail to provide a viable schedule, RVMDES provides the user with a list of additional schedulers which, if implemented, might be able to provide better schedulers, then tells the user to fix his program if he wants it to compile. The advantage of using several schedulers like this is that they are tried in increasing order of complexity, so if the program is simple, the user does not have to wait for a long search by a complicated scheduler to provide a solution.

The first scheduler tries to declare the memory as non-overlapping, which may be sufficient for programs that do array processing or use small images. It initializes the starting position of free memory in each bank to 0, then for each module, finds the first bank of memory with enough space between the starting position and memory size to fit the variable. If it finds space, it records the bank number and starting position with the variable, and increments the starting position of the host bank by the size of the variable. This continues until all variables have been hosted, or the available memory is exhausted. In the second situation, RVMDES tries the second scheduler.

The second scheduling algorithm is more reminiscent of malloc(), placing modules in the first piece of unused (not unallocated) memory large enough to fit it. To do this, RVMDES makes copies of the activity arrays at each variable, and connects those copies into tables, one for each memory bank. For the first variable, copy its Line into a structure containing the line, the variable size and a space for starting address, and attach it to the first bank with enough space to hold it. Increment the starting value for the bank, record the old starting value in the copied structure, and mark the bank and starting value in the.variable node. For each remaining variable, copy its line and size, then see if any of the lines in any of the banks have equal or greater size, and no true entries in common with the candidate line. If none is found, attach the line to the first available bank, as with the first variable. However if one is found, it must be broken into common memory, and memory allocated only to the larger variable. First, insert the new line before the old one on the table, and mark on it the starting point of any entries marked true in the old one. Second, reduce the old line's size and increment its starting point by the size of the new one. Third, record the bank and position of the new line in the variable node. The extra space can then be used by another variable that overlaps the smaller but not the larger variable. If the two variables are the same size, this process can be reduced to recording the bank and position of the old line with the new variable, and marking the old line with any entries marked true in the new variable. This process continues until all variables have been scheduled, or a variable will not fit into any memory bank.

Should the second scheduling algorithm fail, there are other tactics which might still succeed. For instance, the order of operations could be changed, to reduce the size of stretches of activity, and provide more opportunity for overlap. Also, if the region of overlap is a single optimized module, where one variable is being consumed pixelwise and the second is being produced pixelwise later in the optimized module, the two variables can share the same memory. Memory fragmentation could even be reduced, by allowing variables being scheduled to span multiple lines,. However, these options are not implemented in this version. RVMDES simply informs the user of these possible upgrades, and tells him to fix his code to use less space.

3. Writing memory allocation files

Having recorded a bank number and starting address for each variable, RVMDES can now generate a file (base_var_X.c) for each memory bank X, containing code to allocate space for all variables stored in that bank. Each file contains two parts, declarations and mem–init–X(), as shown in figure 5.7. The declarations redeclare all of the variables assigned to the bank, just as they were declared in the main C file, except that they are

declared as externals. The definition of mem_init_X() consists of one line to allocate a block of memory, then one line for each variable in the bank, pointing it into the allocated block. These locations correspond to the start address recorded for each variable during scheduling.

```
extern int   *var1;
extern float *var2;
extern char  *var3;

void mem_init_0 () {
   char *base = (char *) malloc (552960);
   var1 = base+0;
   var2 = base+-245760;
   var3 = base+-491520;
   }
```

figure **5.7** - *example base-var_0.c*

In addition to these files, RVMDES generates a command file that tells the linker what banks of memory are available, and in which banks to put each part of a program. RVMDES writes the bank definitions using the same information it used to schedule into the banks. Then it writes lines to funnel all parts of the program except space allocated by malloc() into the single memory bank designated by the hardware definition structure. This space is allocated to a different bank for every base_var_X.c file, which is the only way RVMDES can tell the compiler to allocate to the various memory banks. With all of these files written, RVMDES compiles the base_var_X.c files, links them with the original C file, and adds a few words to either end to produce an executable, ready to be run on a processor.

4. Why scheduled variables use pointers

In retrospect, it is clear that to make the best use of available memory, variables had to be recycled and had to be allocated outside the main file, but it is not clear why they were declared as pointers instead of arrays. The reason is that the scheduler is able to pack memory more efficiently when the blocks it is packing are addressed by pointers. In fact, it is possible to schedule memory into arrays, and recycle those arrays by using the #define command in C to make several variable names synonyms for the same array. However, **RVMDES's** scheduler does not do this, because using arrays wastes a lot of memory, in two ways.

First, all variables assigned to use an array start at the beginning of that array. If one variable does not use the entire array, memory is wasted because there is no way to begin another variable partway through the array. The array could be broken into two, with addresses for the first purposely overrunning into the memory space of the second, except that C does not guarantee that the two blocks will be contiguous in memory, so the implementation would be non-portable. By contrast, when variables are declared as pointers, they can point anywhere in memory, so a single block of memory can host one variable, or multiple variables beginning anywhere inside the block, and memory is only wasted when the leftover memory in a block is to small to host any variables.

Second, when an array hosts multiple variables, its dimensions must be large enough to accommodate valid indices fiom any of the hosted variables. Thus an array hosting a 128x256 image and a 256x128 image must be 256x256, even though only half of the array will be used at any time, and a quarter will never be used at all. This problem exists because the array is a single data structure, and array indices are converted to memory locations using a single array width. However, when variables are declared as pointers, **RVMDES** must translate indices to memory locations during code generation, and can use a different width for each variable. Thus a 32K chunk of memory is sufficient to host a 256x128 image, then a 128x256 image, with no waste.

As a final note, when variables are declared as pointers, the addresses of the two inputs of wraparound modules can easily be switched. Conversely, it is not clear that wraparound could even be implemented using arrays, without some sort of pointers overlaid.

H. Summary

RVMDES automatically generates code for each processor contained in an RVM design. It manages this because the block diagram is easy to translate into a program skeleton, and function blocks are easily translated to fill out the skeleton. This is made possible because of image processing having a specific set of functions. The C code is compiled on compilers for the supported set of hardware, which can be included in the tool because we know the set we need, because we are using an **RVS.**

VI. Configuring RVM Hardware

Once a designer has chosen hardware modules and partitioned his algorithm onto them, he must determine how to configure the hardware and other Egipt RVS components to make the necessary connections between modules. This step can be a bottleneck in the design cycle because it is time consuming if done by hand, and it must be repeated every time the user changes the algorithm, partition or hardware set to improve system timing. Therefore, to make the design cycle more efficient, RVMDES automates the process. This is possible because RVMDES uses the Egipt RVS, whose interchangeable modules and motherboards with preset connections reduce configuring a network of modules to a graph matching problem. This chapter explains the components of an RVM configuration, the algorithm RVMDES to find them, and why that algorithm was chosen over others.

A. Components of an Egipt RVM configuration

An Egipt RVM consists of several circuit boards, connected by ribbon cables and by a common bus in the form of a 12 slot VME cage. Some circuit boards are modules in themselves, and others are motherboards with slots to hold modules. Configuring an RVM means translating a list of these hardware modules and the required connections between them into a description of how to assemble the modules, along with all supporting hardware, to create and RVM. The difficult part is to determine how to arrange modules on motherboards so that all of the required connections can be made. Once module placement is decided, the configuration must be completed by adding hardware to implement the motherboard connections, selecting jumper positions to customize modules, and assigning circuit boards into VME cage slots. This section explains what information each of these configuration steps produces and how RVMDES conveys the configuration to the user.

1. Hardware modules and connections

The first and hardest task in configuration is to assign the network of hardware modules to motherboard slots, such that the motherboards provide all required connections between the modules, using as little supporting hardware as possible. With six slots on each Egipt RVS motherboard, it is possible to place many modules on a small number of boards, linked by a small number of cables. However, as shown in figure 6.1, motherboard slots have only limited interconnection. It is often impossible to pack modules densely onto boards and have the small number of on-board traces provide the required connections. To solve this problem, the Egipt RVS provide two ways to generate additional, if less attractive, motherboards traces: connecting ribbon cables and shorting across slots.

The first option is to connect ribbon cables between the I/O ports on motherboards. Each motherboard has four input ports and four output ports, through which data can enter and leave the board. These ports are connected to various slots, as shown in figure 6.1 by arrows leading to or from nowhere. The ports are typically connected to send data between two boards, but can also be used to add a trace between two slots of the same board. This allows the motherboard to provide connections between formerly unconnected slots, and to provide secondary connections between slots, to in case hardware

modules with fixed port usage require doubled connections. On the negative side, cable connections are unsightly, provide only half the data transfer rate of on-board traces, and cause the **RVM** to fail if they are not well seated. For these reasons, they are used only as a last resort.

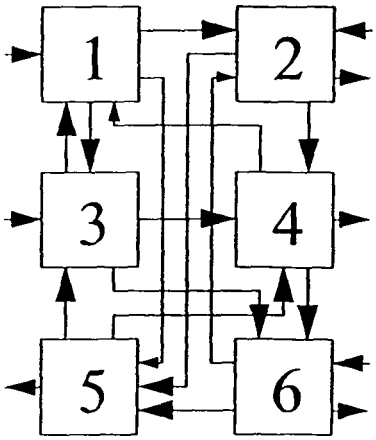


figure 6.1 - schematic diagram of a baseboard. arrows represent the unidirectional traces connecting the six slots.

The second way to generate additional connections is to short traces across slots. Shorts are more flexible than cables, because there are often several shorts that could connect two modules. For instance, as shown in figure 6.1, slot 1 has no direct connection to slot 4, but both slots have traces to slots 2, 3 and 5, so a short across any of those slots would create the desired connection. Shorts are also advantageous because they provide data transfer rates comparable to on board traces. However, shorts can cause an **RVM** to fail if improperly seated, and complicate debugging by adding more segments to traces. Further, shorts can only be made across empty slots, or slots whose hardware module does not use all of its inputs or all outputs. For these reasons, shorts are still inferior to single, on-board traces.

Because traces are not fixed, the actual first task in configuring hardware is to generate a slot assignment for each hardware module, and lists of traces that must be shorted and ports that must be cabled together to implement all required intermodule connections with a minimum amount of each type of hardware. **RVMDES** locates this configuration using a search over the set of possible configurations, as will be described in section B.

2. Customizing traces with overlays

Traces that link module slots on Egipt RVS motherboards do not actually connect to the ports of those slots. The traces must be completed by inserting small ***overlay*** cards into connectors on the back of the motherboard. As shown in **figure 6.2,** overlays map traces onto ports, allowing a particular port to use **any** trace available to the slot. This feature is highly useful for Egipt RVS hardware modules that have specific ports for specific data, because it lets them communicate that data with any of three neighboring slots, not just the one slot at the end of the port's one trace. Alternately, a single overlay can connect the input and output traces directly, shorting them all across the slot but isolating any module in the slot.

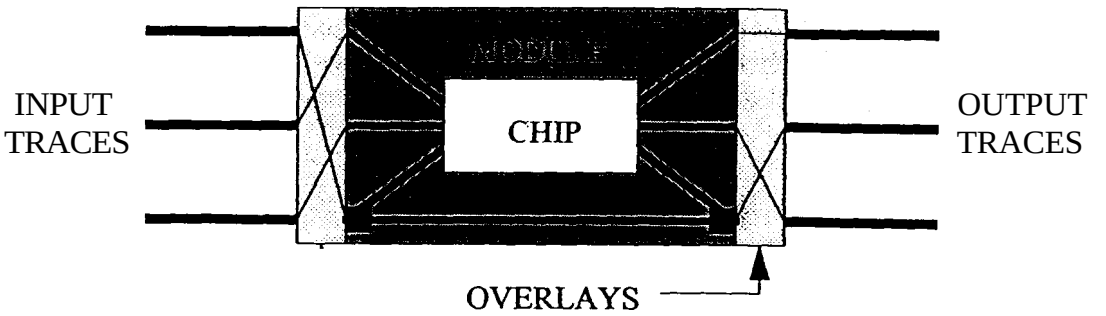


figure 6.2 - how overlays connect daughter cards to motherboards

While overlays make the RVS motherboards more flexible, they also complicate the configuration process. There are six kinds of overlays, mapping each combination of 3 inputs to 3 outputs, and the designer much choose which overlay or pair of overlays should connect the traces at each slot. This is a reasonably straightforward process once modules are assigned to slots, but it is also tedious, so RVMDES automates it.

To determine the pairs of overlays required by slots containing modules, RVMDES first matches traces to ports, using the list of modules used to populate those boards. Each node on the list represents one hardware module, and records what slot the module is assigned to, which ports the module uses, and which ports of which modules each connects to. To match ports to traces, RVMDES chooses a module, finds the slot it occupies, follows each of the slot's input traces back through any shorts to the slots that originate them, and determines which modules are assigned to each of those slots. RVMDES then matches the input port and output port that should connect each pair of modules, according to the list, with the traces that actually lead to them. Thus, each trace determines one port-to-trace connection for the main module's input overlay, and one connection for the source module's output overlay.

Because the overlays map three traces to three ports, two port-to-trace connections imply the third, and identify the overlay type they require, and RVMDES uses a table to map between connections and overlay types. If a module uses only one output port, there are two ways to decide which overlay(s) to use. If the module is a C44 and a pass-through is recorded across the slot, the pass-through must use the third input and third output port, providing a port-to-trace match on both sides. Otherwise, the slot truly has only one constraint on either side, and RVMDES simply chooses the first type of overlay that matches that constraint.

If a slot contains a short but no module, it will still require a single overlay short the input and output traces. Because shorts are recorded in terms of the slots they must connect, RVMDES determines which traces lead to the two slots, and records one trace-to-trace connection. Lacking a second constraint, RVMDES chooses the lower numbered of the two overlay types that can make that connection. Technically, an empty slot can make three shorts with a single overlay, so RVMDES could look at the first two to uniquely determine the required overlay. However, the current search method only allows one

short per slot.

3. Setting C44 port directions with jumpers

The C44 **DSP,** which is at the center of the main hardware module in the Egipt RVS, has four bidirectional communication ports, not the six unidirectional ones allowed to an Egipt RVS module. To maximize the flexibility of the C44's ports, Egipt RVS C44 modules can map two C44 ports as module inputs or outputs. The direction of these ports is controlled by two jumpers, as shown as a pair of black switches in figure 6.3. The right hand jumper controls port 2, the left hand side controls port 5, dark part up means input, down means output.

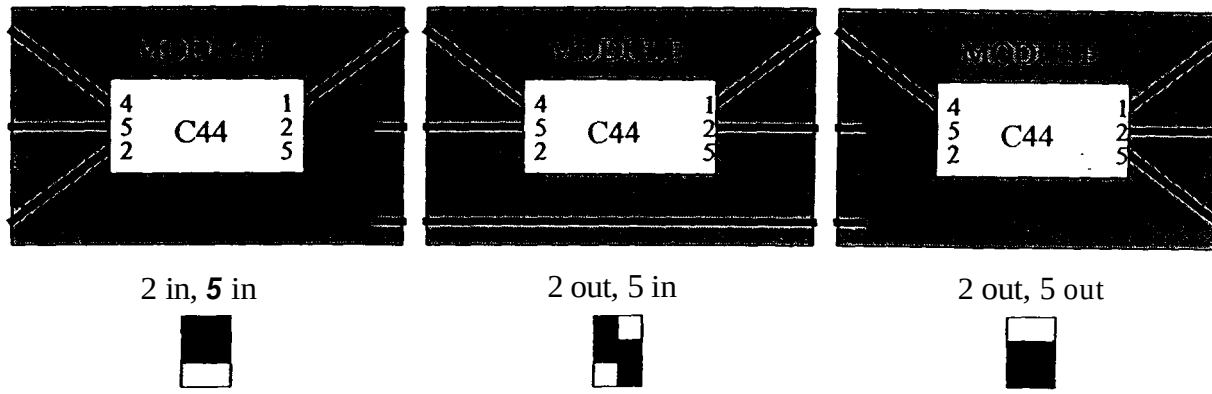


figure 6.3 - port configuration on C44 modules, and the jumper settings that implement them

RVMDES determines the jumper settings for each C44 module from the module's connectivity in the hardware network. If a module uses all three input or output ports, its jumpers are set to the first or third configurations, to make three connections. If not, RVMDES chooses the second configuration, providing two input and two output ports, and shorting across the unused ports. There is a fourth possible configuration, which would map port 2 as an input and port 5. This has no advantage over the second configuration, and does not provide the shorted ports, so RVMDES does not use that jumper configuration. When generating code, RVMDES will assign port 2 as an input or port 5 as an output only if the other ports are used, so that the fourth configuration will never be needed.

4. Connecting boards

The remaining configuration component is the assignment of circuit boards to slots on a VME bus. The RVM interface must know which bus address to use to communicate with each module, so RVMDES must fix the modules position. However, the actual choice of positions is almost arbitrary. At present, the host interface board must be just to the left of the RVM's single A/D board, to permit certain external connections. Following an established pattern for hand-designed Egipt RVMs, RVMDES places the host interface and A/D board in slots 0 and 1. It then spaces the remaining boards evenly in the remaining 10 slots, with motherboards in the center and D/A boards on the right.

In the future, all boards except motherboards will be reduced to modules, allowing multiple A/D boards and completely arbitrary placement. At that point, RVMDES will evenly space boards, in random order, through the 12 slots.

5. RVMDES configuration display

Once RVMDES has completed the four configuration steps, it displays the configuration for the user, in a diagram like the one in figure **6.4. The four** large rectangles that dominate the diagram represent circuit boards: host interface, A/D, motherboard and D/A. Small boxes on the left and right of each board represent input and output ports, respectively, and thick lines connecting them represent ribbon cables.

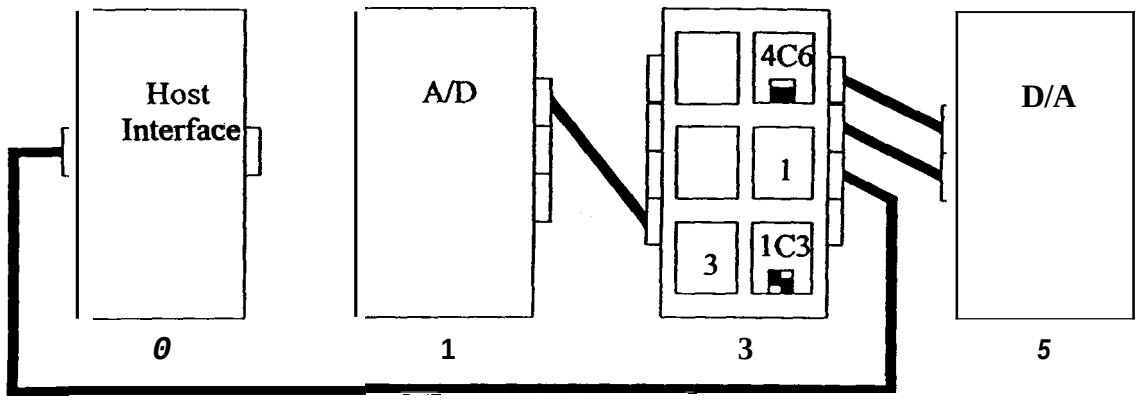


figure 6.4 - A configuration display

The boards, ports and cables can be seen assembled in a VME cage in figure 6.5. The numbers under the boards specify the VME slots in which the boards should be placed, the left-most being slot 0. For motherboards, input and output ports form **two** columns, with the inputs on the left. For the host interface, the input port is below the output port, and the remaining boards only have input or output ports. Most of these boards can simply by seated in the VME cage, and connected to each other.

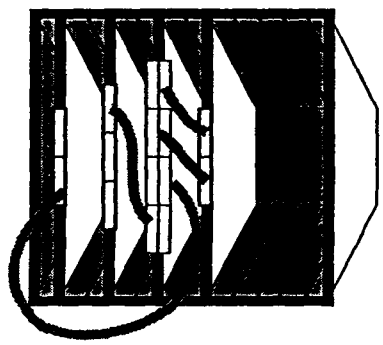


figure 6.5 - boards in their VME cage

The third board in figure 6.4 is a motherboard, **and** requires additional assembly. Figure **6.4** shows how the baseboard schematic translates into **a** real board, **fiom** front **and** back views. The front side requires **two** assembly steps: inserting hardware modules and set-

ting their. The schematic indicates where modules and overlays must be inserted using labels of the form 'xYz', where 'x' and 'z' are numbers and 'Y' is a letter. The letter 'C' in figure 6.6 calls for C44 modules in the two right corner slots, as reflected in the front view of the actual board. Each slot containing a C44 module is also marked with a small box indicating the settings of its jumpers. A blow up of the actual board shows the jumpers for the C44 in slot 6, in the default configuration.

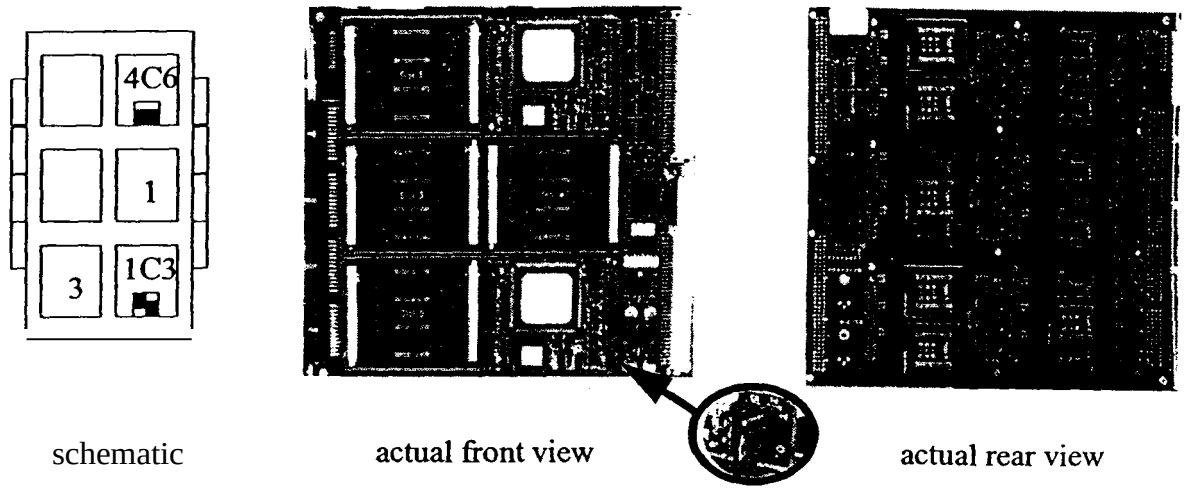


figure 6.6 - a baseboard schematic diagram and actual view

On the back side of a motherboard, each slot has a column of 4 connectors for connecting overlays. A schematic slot labeled 'xYz' requires overlays of type 'x' and type 'y' (there are 6 types), connected across its upper and lower pairs of connectors. Thus in figure 6.6, the top right slot, labeled '4C6', indicates a type 4 overlay above a type 6 overlay on the reverse side of the slot. A slot with a single number requires a single overlay across its middle two connectors, as in the single type 1 connector in the bottom left of the schematic, and bottom right of the rear view. The only tricky part about overlays is that when a baseboard is flipped over, the slots reverse sides. Thus the top right slot on the front of the board is in the top left on the back side. That is why the overlays are shown on the left side of the board, even though the schematic would seem to indicate them on the right side.

A second configuration display, in figure 6.7, shows a configuration for the two object inspection RVM that has appeared throughout the thesis. At the end of the hardware design cycle, hardware for this RVM included an A/D module, three C44s, a D/A module and a host interface. These appear in the configuration display much like in the previous example. All C44 jumpers are set into the default two input, two output configuration because no modules require three inputs or outputs. Also of interest are the *fork* and ***merge*** modules on the motherboard. RVMDES adds these automatically to deal data to parallel C44s and collect results from them.

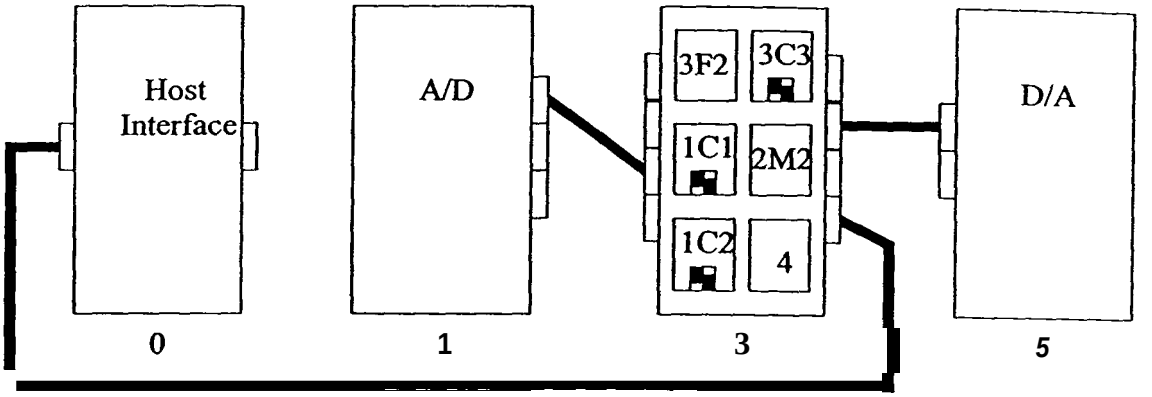


figure 6.7 - Configuration display for two object inspection RVM

B. Assigning slots by hill climbing

The difficult part of configuring an **RVM** is the first step -- deciding how to assign hardware modules to motherboard slots so that the motherboards can provide all required interconnections. The most direct way is to search the space of possible hardware configurations, testing whether each provides the required connections. This search will be long, because the number of possible hardware assignments grows exponentially with the number of modules, as does the set of cables **and** shorts that might connect modules. Further, the search must continue even if it finds a viable configuration, because it cannot know whether it has found the minimum hardware configuration until it has tested all configurations. Because the search will be long, the user is likely to grow impatient before it has covered much of the search space. **RVMDES** must provide **a** way for the user to stop the search and take the best solution found so far.

To help the user make intelligent decisions about when to stop the search, RVMDES provides a configuration window showing the best solution so far, the score for that solution, the number of suboptimal solutions found since finding the optimal solution being displayed, and a stop button. As the frequency of new, improved solutions decreases and the number of suboptimal solutions increases, the user can be increasingly confident that the current best solution is near optimal, and can stop the search using the stop button. Again, because the search space is large, and little of the search space *can* be covered before the user stops the search, it is important to select **a** search algorithm that can quickly provide optimal or near optimal solutions from across the search space, rather **than** exhaustively testing one region of the space.

RVMDES performs the search using a hill climbing algorithm. The algorithm chooses a random initial assignment of modules **to** slots, then repeatedly tweaks the assignment and retains the modified version if it provides more of the required connections or uses less hardware. If after much tweaking the assignment does not improve and does not provide a valid configuration, the algorithm randomizes a new assignment, and begins again. Such a search can quickly find local maxima spread across the search space, possibly finding

optimal or near optimal solutions early in the search. It is not exhaustive, and so does not have a stopping criteria, but because the user is likely to cut any search short, a stopping criteria is not necessary. The search will continue to produce improved or optimal solutions as long as the user cares to wait. This section explains how the hill climbing algorithm generates and evolves possible configurations, how it adds shorts and cables to maximize the connectivity of those configurations, how it scores them, and how well the implementation of the algorithm works.

1. Evolving a configuration

To begin the evolution of a configuration, RVMDES assigns each hardware module to a motherboard slot. RVMDES assumes that motherboards are flexible enough to configure the modules on the minimum number of motherboards necessary to physically hold them. Each module is then assigned to a random board, and a random slot on that board. Because this random placement will almost certainly not provide a valid configuration, RVMDES begins evolving it.

RVMDES cycles through the list of modules, perturbing each, and deciding whether the perturbed solution is better. The perturbation actually generates two new candidate configurations. The first has the unperturbed module configuration, with different routing (cables and shorts). RVMDES removes any routing attached to the current module's slot, then lays down new routing. This may provide a better solution if changes to other modules have opened up new routing possibilities. The second perturbation also clears and replaces routing, but reassigns the module to a new, random slot in between. This may provide a better solution if the module can make more of its required connections fiom its new slot. After generating the two perturbed configuration, RVMDES scores both, along with the unperturbed version, and retains the one with the highest score. In this way configurations either improve or stay the same with each step.

If the search cycles many times through the module list and fails to change anything, it has probably reached a local maximum, and must be reset. This can happen in two ways. In general, the search does not produce a viable solution, and RVMDES reassigns a few modules to new slots, giving each module a 16% chance of moving. The idea is that if the configuration is stuck at a local maximum, then changing a few modules will shake the solution up enough to move it to a neighboring maximum, but will retain enough converged structure to ensure a fast convergence. If several such resets do not produce a valid solution, RVMDES abandons the configuration and begins evolving with a new, completely randomized solution. The remaining possibility is that the search may produce a valid configuration -- one that allows all required connections, assigns each module to a different slot, and only uses each trace or cable to connect one pair of modules. In this case, RVMDES completely randomizes the configuration, assuming that it has found the highest local maximum.

In addition to resetting the search once it converges, RVMDES informs the user of the convergence by updating the configuration window displayed throughout the search. If a converged configuration has a score is higher than the displayed configuration, RVMDES displays the new solution and its score, and zeroes the number of suboptimal solutions.

Otherwise it increments the number of suboptimal solutions.

2. Adding overlays and cables

RVMDES reduces the size of the configuration search space by assigning cables and shorts to make specific connections, rather than letting random perturbations place them. Thus for each perturbation, RVMDES attempts to add cables and shorts to connect the perturbed module with each partner module that it needs to connect to. To connect with a given partner, RVMDES first checks whether the connection not is already made, via an on board trace or preexisting routing. If so, it simply marks the connection as used. Otherwise, it uses one of two breadth first search algorithms to find the shortest path between the two modules.

If the two modules are on the same board, RVMDES builds a search tree extending from the source. Each node represents a path from the source module's slot to another slot, and is expanded by following the final slot's output traces to three new slots. The search is sharply pruned because it may not follow traces that are used by other connections, lead to nodes already visited, lead to nodes that already contain shorts (except the destination node), or lead to motherboard output ports. This pruning also limits the depth of the tree, since a path can only follow five links before triggering one of the pruning conditions. Once the first path is found, RVMDES stops the search, marks each element of the path as used, and records which traces of which slots must be shorted to produce the path.

If the search fails, or if the two modules are on different boards, RVMDES uses a similar search to find the shortest path from the source module to an output port and the shortest path fiom the destination module to an input port. The only difference in the search is that instead of pruning once a port is found, the search returns the path to the port. If both searches are successful, RVMDES records shorts and used connections as before, and also records that a cable connects the ports at the ends of the two paths.

3. Scoring a configuration

To determine whether a perturbation improves a configuration, RVMDES tests the perturbed module with a scoring function. In general, a search would apply a scoring function to the entire configuration, but to save time RVMDES only tests one module. Because any given change only applies to one slot, comparing single module scores and comparing full configuration scores are equivalent unless multiple modules reside in the same slot. In this case, perturbing one module will remove the routing used by both, and only replace the routing for the perturbed module. This could lower the overall score, even though the score for the perturbed module increases. However, when the second module is perturbed, it will reestablish its routing, unless another module uses some of the traces, and either way, the number of connections made at the end of a cycle should be constant. Thus in the one case where the faster, single module scoring is not strictly accurate, it at least will not decrease the overall score.

Because the search algorithm scores one module at a time, the cost function rewards and punished situations that occur at each module. There are five such situations, explained in order of desirability, and summarized in table 2. A module receives a reward for every

required connection that is made without leaving a motherboard. This is the best type of connection, and the high score encourages configurations to use as many as possible. A module also receives a reward for every required connection that is made With the help of a cable. The reward for a cable connection is just over half that for an on board connection because, while using cables is undesirable, a configuration that uses two cables to make two connections is still preferable to one that provides one on-board connection and one unimplemented connection. Overlays facilitate both types of connections, but each makes the design a little more expensive and a little harder to debug. To represent this, each overlay used in a required connection incurs a small penalty. The penalty is sufficiently small that an on-board connection using overlays is still better than any cable connection, and that connections with overlays are always beneficial, but connections without overlays are even better.

Besides rewarding modules for making connections, the algorithm penalizes modules for overusing resources. If any segment of any scored connection is used to connect more than one connection, the module is penalized the equivalent of one on-board connection. This situation only occurs when a trace connects two modules and is used in a routed connection. When either module involved in the routed connection is perturbed, it will fail to regenerate the connection, which will clear the double use. This actually increases the module's score, because the penalty for double use is higher than the reward for any routed connection, so the configuration with no double use will be chosen. The remaining penalty is similar, punishing a module when it is not the only module in its slot. The penalty is high enough to cancel any rewards a module can obtain fiom connections. This prevents sets of duplicated modules fiom all settling on the same spot, where their connections are all made, but which cannot provide a valid configuration.

Table 2: Scoring function

Condition	Award
Each on-board connection	+20
Each cable connection	+12
Each overlay on those connections	-1
Each double-used connection	-20
Double used slot	-80

The only problem with the scoring function is that when multiple modules in the same slot are heavily penalized, configurations stop evolving as soon as each module occupies its own slot, even if connections cannot be made. This is undesirable, but the penalty must be high to prevent multiple modules from settling in the same slot. To keep the configuration evolving despite the high penalty, RVMDES introduces the concept of *invisibility*. At the beginning of each cycle through the module list, one module is chosen to be invisible. When RVMDES scores a module, it looks other module in the same slot, but ignores the invisible module. This allows any module to jump into the invisible module's slot without

penalty. Further, in the next cycle the formerly invisible module or its new slot-mate can jump to any other slot that improves its connectivity, because it will take the same double-slot penalty in its current position or in a new position. Eventually the module will jump into an empty slot and be locked into place again, but with one new module made invisible in each cycle, there should always be several mobile modules, until modules are in sufficiently good positions that connectivity considerations hold them there. As a final note, the high penalty for multiply using a slot ensures that a module can never move to a new slot containing at least as many visible modules as its old slot. Therefore, the search algorithm does not waste time calculating scores for any such perturbations.

4. Results

To determine the effectiveness of the RVMDES implementation of the algorithm, its performance was analyzed finding configurations for three separate hardware networks. These included a twelve module network that the RVS motherboards were specifically designed to handle, a nine module network that the boards were not designed for, and the five module network implementing the two object inspection RVM designed in chapter 3. Note that the module count only includes modules that are inserted into motherboards, because modules that are boards unto themselves are not assigned slots by the search algorithm. For each of the three networks, statistics were tabulated to record the speed of convergence of the hill climbing algorithm, on a 200MHz Pentiurn, and the distribution of scores for convergences representing valid hardware configurations.

a. Twelve module system

First, the algorithm was tested on a twelve module network. Table 3 shows various statistics about the time required to find viable hardware configurations, when the algorithm uses zero, one or two invisible modules, with and without soft resets. Results are averaged over 400 trials per row. The baseline algorithm is shown on the second line, with one invisible module and soft resets. It required the shortest time to find a viable solution, producing solutions every four seconds, on the average. Measurement error is ±0.5 sec /

Table 3: Convergence speed for 12 module network

invisible modules	soft resets	time per solution (sec)	convergences per solution (m, σ)	cycles per convergence (m, σ)
0	y	15.73	115, 109	78.9, 229
1	y			478,436
2	y			1810,1331
0	n	30.3 1	211,207	80.0, 297
1	n	5.16	5.63, 4.58	541,437
2	n	14.58	4.05, 3.35	1940,1330

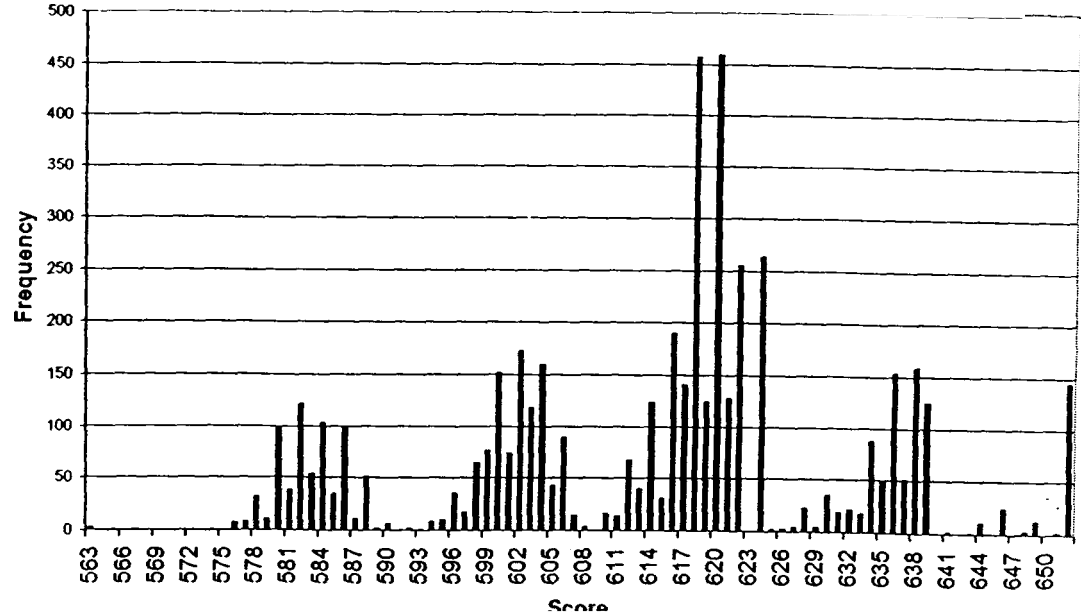
The remaining rows show that variants of the algorithm performed worse, and the remaining columns show why. Removing soft resets increases the time required per solution, by raising the number of cycles per convergence and convergence per solution. These increases result fiom the decreased likelihood of a reset configuration being near a valid solution when it is completely randomized. The effect diminishes as an increasing number of invisible modules reduces the number of resets per solution.

Eliminating the invisible module or adding a second one both dramatically raise the time to solution, but for different reasons. Removing the invisible module forces configurations to converge as soon as each module occupies one slot. This raises the number of convergences required to find a valid solution. It also decreases the number of cycles required per solution, but because the algorithm requires 60 cycles with no changes to declare a configuration converged, dampening the effect of a decrease in the number of cycles required to find the converged solution. By contrast, setting two modules invisible each cycle increased the mobility of modules in a configuration, which increased the chance of converging to a valid solution, but drove up the settling time. In this case, the extra mobility was insufficient to move configurations away from invalid local maxima in the search space, so the number of convergences per solution did not decrease enough to match cover the increased time to convergence.

The notation of the table deserves a little explanation. The number of resets per solution and cycles per reset are shown as mean and deviation. The deviations are high, showing that the numbers are highly variable, but the means are all stable to three significant figures. The high deviations, coupled with the fact that resets per solution and cycles per reset are non-negative, shows that samples are bunched below the mean, and trail off toward higher numbers.

At four seconds per solution, the baseline algorithm should find plenty of solutions before the user grows impatient, but there is some question of how good those solutions are. To test this, the algorithm was allowed to find 5000 solutions, and record their scores. The resulting histogram of scores is shown in figure 7.7. The probability of finding various scores resembles a sinc function, with one lobe for each number of cables. The spread within the lobes indicates solutions with varying numbers of supporting overlays. Odd scores are less frequent than their even neighbors, because they only occur when overlays route connections to separate A/D and D/A boards, and these connections can generally be made without using overlays. The optimal solution was reached 145 times during the 5000 trials, 2.9% of the time. The average number of trials required to attain a particular score is the reciprocal of the chance of one trial attaining the score, in this case 34 trials. That means the method should find an optimal solution, on the average, in just over two minutes.

figure 7.7 - Solution scores for twelve module network



*b. **Nine*** module network

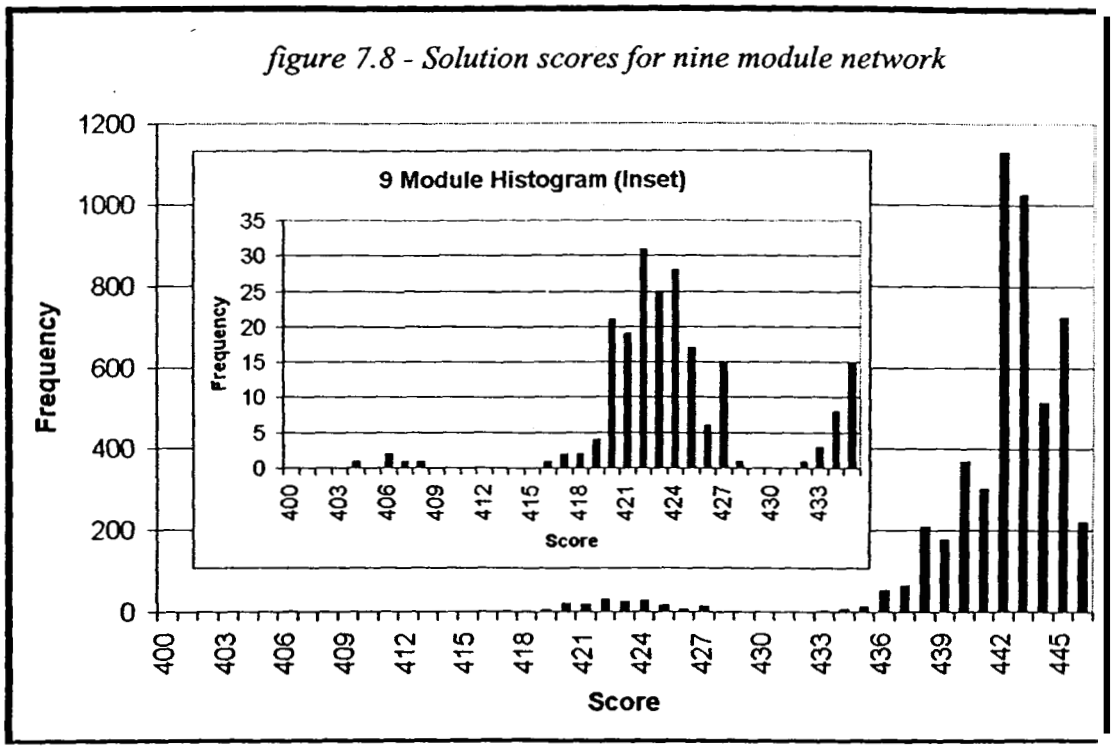
A second round of tests checked the times to 400 solutions and a histogram of 5000 scores, this time for a nine module network that the boards were not specifically designed to handle. The network must be hosted on two motherboards, with twelve slots, leaving three slots empty. The algorithm **can** then fill the most useful nine slots with modules, and short across the others, so the design problem should be easier **than** the twelve module network problem. In fact, as table 4 shows, average time to solution improved dramatically in all cases.

number of invisible modules	soft resets	time per solution (sec)	convergences per solution (m, σ)	cycles per convergence (m, σ)
0	y	.37	2.95, 2.65	77.9, 18.0
1	y	.36	1.75, 1.34	114, 54
2	y	.31	1.62, 1.15	127, 67
0	n	.29	2.39, 1.82	82.3, 16.7
1	n	.30	1.57, 0.94	125, 65
2	n	.36	1.48, 0.83	140, 74

pronounced as before, because the three vacant slots effectively added three invisible modules to the system. Unlike before, omitting soft resets decreased the number of resets per solution, probably because the increased number of valid solutions meant a decreased average distance between a random reset and a viable solution. The total timing did not indicate anything, because a measurement error of .05 seconds made the times per solution essentially equal. Because the search algorithm variants all exhibit about the same time to solution on the nine module system, RVMDES will continue to use the initial baseline algorithm with one invisible module and soft resets, which was the obvious best choice in the first experiment.

A second series of trial runs produced another histogram of scores, this time finding only two groups of scores corresponding to use of one or two cables to connect the two motherboards. Of the 5000 solutions, 222 were optimal, indicating an average of 22.5 tests, lasting 8 seconds, to find an optimal solution. This should be well within the user's attention span.

figure 7.8 - Solution scores for nine module network



c. Five module network

One final round of tests checked statistics for the five module network implementing the two object inspection algorithm used throughout the thesis. Table 5 summarizes the results, which basically indicate times too small to reliably measure and compare. Almost all attempts converged from their initial starting positions, so the presence or absence of soft resets had almost effect. The absence of an invisible module slowed the search by requiring more cycles per convergence, but did not prevent convergence because an open slot on the board acted just like an invisible module. The effect of adding a second invisible module was too small to consider useful. Because the times were all small in this trial and the differences between variants were negligible, RVMDES will continue to use its

original algorithm.

Table 5: Convergence speed for a five module network

number of invisible modules	soft resets	time per solution (sec)	convergences per solution (m, σ)	cycles per convergence (m, σ)
0	y	.14	1.11, 0.60	225, 775
1	y	.07	1.09, 0.30	112, 70
2	y	.08	**1.07,** 0.27	109, 67
0	n	.16	1.12, 0.34	304,937
1	n	.07	1.07, 0.28	117, 71
2	n	*.06*	1.10, 0.32	113, 59

A histogram of scores for the five module system indicates that most attempts converged to optimal scores. In fact, 3847 of the 5000 scores were optimal, indicating an optimal solution every 1.29 trials or .09 seconds, on the average.

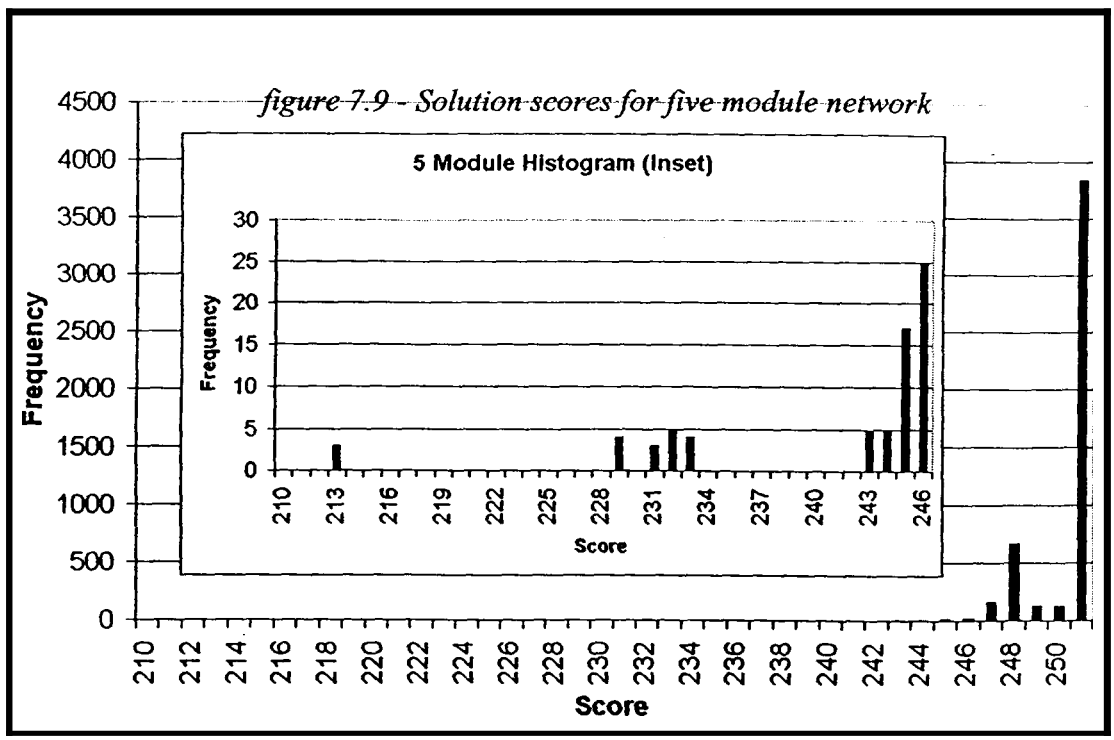


figure 7.9 - Solution scores for five module network

C. Other search techniques

RVMDES was fitted with a hill climbing search algorithm because it seemed most likely to provide wide coverage of the search space in a short amount of time. This decision was made after considering several techniques, all of which seemed likely to produce results

more slowly than hill climbing. This section describes the three categories of search techniques that were rejected in favor of hill climbing, and why they were rejected.

1. Exhaustive search

An exhaustive search could consider every possible placement of modules, overlays and cables onto motherboards, score each based on the amount of hardware used, and return the configuration with the highest score. The search would always produce an optimal solution, because it would test all solutions. Also, the search need not test every possible configuration. A branch and bound search could place one module at a time until all required connections to a module could not be made, a module had insufficient unused ports to connect to unplaced modules, or the amount of hardware required to make the connections exceeded the amount required in the best working configuration found so far. This pruned search would still find the optimal solution, and would eliminate a large fi-action of the search space.

Because this essentially exhaustive search seemed likely to be fast and to always provide an answer, an early implementation of RVMDES used it. The implementation revealed that exhaustive search is slow, even when heavily pruned, because the search space grows exponentially with the number of hardware modules. After several hours of searching for configurations for an 18 module network, on a 90MHz Pentium, all solutions contained the same placement of the first six modules, indicating that an insignificant fi-action of the search space had been explored. The implementation did find a solution within the first few minutes, and several more solutions in the next few hours, but using them would destroy the only advantage of exhaustive search, which is the fact that it tests all solutions.

2. Enhanced hill climbing

A second alternative search strategy would be to apply one of many common enhancements to hill climbing. None of these were implemented, because they all seemed unlikely to improve the performance of RVMDES's algorithm.

Simulated annealing allows the hill climbing algorithm to accept new configurations even if their score is worse than the current configuration. The amount that a score can drop becomes less as time goes by, so that eventually only changes that improve the score are accepted. The idea is that early in the search, configurations can jump between the slopes of local maxima, but are more likely to move toward higher maxima. As the allowable drop in score approaches zero, configurations will climb to their nearest local maximum, which should be the largest one nearby. This is a sensible tactic, but it only applies when the search must decide whether or not to move, and RVMDES must decide between three solutions at each step. Also, RVMDES's hill climbing algorithm already explores neighboring local minima, by only changing part of a configuration when it finds an unsuccessful solution.

Steepest ascent chooses the best direction to move in a search space, rather than testing arbitrary directions as RVMDES's hill climbing does. To do this, it must test every possible move, then choose the one that gives the highest score. This may be useful for searches where the cost of making a move significantly outweighs the cost of simulating it

to decide the cost, but RVMDES must actually build a configuration to calculate its score. Thus in the time steepest ascent would take to choose the best module to move in a configuration, regular hill climbing could move each module several times, possibly finishing the search.

A third take on hill climbing is to remove the background information from the search, and simply represent the configuration as a data string. At each step, that string is perturbed, then scored, and kept if the score increased. This is the basic idea behind search algorithms like PBIL. It also logically extends to genetic algorithms, which attempt to climb hills in several directions at once, again with no problem specific information applied during the move. While these are valid search techniques, they are unlikely to converge on a good answer any faster than the current hill climbing algorithm, which takes advantage of problem specific knowledge, such as how overlays and cables couple with module positions to affect connectivity.

3. Heuristics

The RVM baseboard was designed specifically to host certain common patterns of modules, such as a pipeline of up to six modules, or a fork module dealing images to three identical processors, which all feed results to a single merge module. When configuring by hand, the user breaks a hardware network into subgraphs consisting of the common patterns, and looks up how to configure them on boards. Perhaps then a graph matching algorithm could be used instead of a regular search, taking advantage of this extra, domain specific knowledge.

This was not done in RVMDES, however, because it is not expandable. Configurations have only been recorded for a few common patterns of hardware modules, because the Egipt RVS is new, and only a few real algorithms have been proposed for it. Because RVMDES must allow larger and more complicated networks in the future, it is important that the configuration method be scalable to novel network topologies. Note that this reasoning does not preclude implementing the search as a graph matching problem against the set of all networks that can be made fiom a motherboard, but this is a huge set, and a regular search is likely to find a solution just as quickly.

D. Conclusion

RVMDES makes the hardware design cycle more efficient by automatically generating an RVM configuration fiom a user's chosen hardware modules. Three of the four steps of configuration are simple, and RVMDES is able to automate them easily. The remaining step, placing modules on motherboards, is more difficult and requires a search over the space of possible configurations. The search method produced optimal configurations in short amounts of time, and other methods seem unlikely to perform any better. All four steps are made possible because RVMDES draws fiom the interchangeable hardware of the Egipt RVS, which in turn is made possible because only a limited number of hardware types are needed for a system specializing in image processing.

VII. Interfacing with an RVM

A complete RVM design includes interface software that allows an operator to control the RVM. RVMDES can generate this software automatically, because the types of interface functions required for image processing machines is predictable, and the exact controls required for a particular algorithm are easily determined from the algorithm's block diagram. The resulting software actually consists of a set of data files, not an executable. A separate, precompiled program contains the engine that passes data and commands between the user and the RVM. This program customizes itself to a particular RVM by reading the data files, generated by RVMDES, that describe the exact controls needed for a particular algorithm. This chapter explains how the customized program allows an operator to control an RVM, how RVMDES mostly automates designing a customized interface screen, and what data files convey the screen to the precompiled executable.

A. The man-machine interface

A man-machine interface for an RVM provides a user with three capabilities -- testing an RVM, running it, and modifying its functionality as it runs. RVMDES provides Egipt RVMs with such an interface in the form of a program called MMI. MMI puts up a screen for the user, so he can send commands to and receive return values fiom an RVM, by communicating with the RVM's host interface module. This section explains how MMI mediates between the user and the host interface module, to provide the three interface functions.

1. Testing the RVM

The first purpose of the MMI is to let the user test whether the RVM he has assembled matches the RVM configuration generated by RVMDES. The user activates this test function by pushing a button labelled "test" on the MMI screen, causing the MMI to test the RVM configuration and translate the results into a form the human can use to debug the hardware.

To test the configuration, the MMI sends a test file to the RVM's host interface module, which deals pieces of the file to each other RVM module over the global bus. The test file provides a set of test programs, and a list of modules to send them to. The host interface queries each module to determine its type, and if it matches the expected type, sends the module its designated test program and its position on the list. The test program depends only on the type of module, and for customized hardware modules it may actually be a single command to activate a test mode. Once begun, a module's test program attempts to write the module's list position and a port number to each output port, and to read corresponding data over each input port. The module then waits for the host interface to contact it on the global bus, and reports the messages it received on each input port. The host interface accumulates these test results from each module, and returns them and any inconsistencies between expected and actual module types to the MMI.

When the MMI receives these results, it opens a window, showing the expected RVM configuration, just as RVMDES created it. It then compares the required connections

between RVM modules with the messages actually received, and reports any errors by deciding which parts of the configuration could have caused them, and drawing them in red. It also draws any modules in red if the expected and actual module types do not match. Once all is checked, the MMI writes the number of errors to the top of the window, so the user can quickly see whether the RVM is correctly built, then look for possible errors if not.

2. Booting the RVM

The second purpose of an interface is to allow the user to boot the RVM and view run time statistics to monitor its progress. The user activates this function of the MMI using by clicking the button marked "run". In response, the MMI sends a boot file to the RVM's host interface module, which again divides the file and routes appropriate parts to appropriate modules, booting each module of the RVM. This file also tells the RVM the amount of statistics data is should expect to return to the MMI after each fiame. After sending the boot file, the MMI will also send command packets to initialize any custom hardware modules or initializing constants in processor modules, then sends the commands to start the RVM's input modules, setting the entire RVM in motion.

As the RVM finishes processing each frame of data, it will send the host interface module data corresponding to any output blocks in the RVM's algorithm that were partitioned to the host interface. The host interface knows how much to expect each frame, reads these into buffers, and passes them back to the MMI as soon as it can. The MMI in turn maintains a set of windows to display these statistics. It knows the order to expect data, so it parses the data as it arrives, updating the proper windows to show the user the needed run time statistics.

3. Adjusting parameters

The third purpose of an interface is to allow the user to modify the parameters of an RVM's algorithm or hardware at run time, in case conditions relevant to an algorithm cannot be determined until then. This can happen when the number of input samples required to determine a parameter is large, or when the parameter varies depending on where a machine is used. To allow the user to view and change parameter values, the MMI screen provides a widget for each initializing constant in an algorithm and each hardware module that is programmed by parameters rather than an executable. Examples of these widgets are shown in figure 7.1. Each module has a label to identify its function, and one or more values. Hardware parameter widgets also have labels for each field.

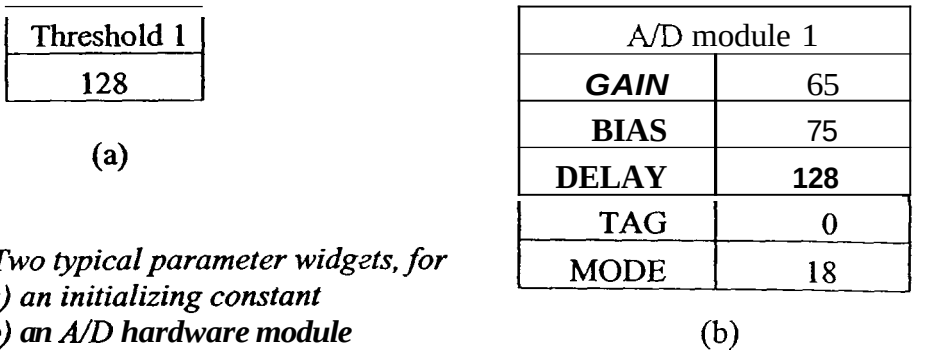


figure 7.1 - Two typical parameter widgets, for
(a) an initializing constant
(b) an A/D hardware module

From these widgets, the user can both see current values and set new values. Clicking the label instructs the MMI to send a command packet to pass the new values to the appropriate module. The MMI prepends the destination board and slot number, the amount of data and the command number, and sends the packet to the host interface. The host interface in turn passes the command number and data on to the appropriate module via the global bus. Dedicated hardware modules will recognize the command number and read in new parameters, while processors will read the data as the new values for the initializing constant specified by the command number.

To support the use of these widgets, the MMI provides three buttons. The first two allow the user to save current widget settings if they must be modified, and them if he renders a system unfunctional by playing with them. The third button toggles the MMI in and out of configuration mode. When not in configuration mode, only the set of widgets necessary to tweak regular operation of the machine are visible. Parameters such as camera capture mode, which should be set once and then left alone, are hidden so that they are not accidently changed. However, these parameters are still available in configuration mode, so that they can be set when the machine is commissioned, or if its environment changes drastically.

B. Building an MMI screen

The MMI lets a user control and monitor an RVM by communicating with the **RVM's** host interface module, to respond to and update a screen containing buttons, parameter widgets and statistic windows. By relying on the MMI to implement all of the logic involved in this communication, **RVMDES** reduces the task of designing an interface program to choosing what controls will be included, and customizing their attributes and placement. RVMDES then further simplifies the user's role by automatically choosing the number and some attributes of the controls needed in the interface, based on the user's algorithm and choice of hardware. It places these module in an editor and allows the user to reposition them, change a few attributes, and set initial values for parameter widgets, to make an ergonomic interface screen. This section considers how **RVMDES** initializes each set of controls, and the user customizes them. In each case, it refers to the example interface screen in figure 7.2, for the two object inspection RVM that has appeared throughout the thesis.

1. Parameter widgets

RVMDES generates one parameter widget for each initializing constant block in the user's algorithm and one for each hardware module that takes parameters instead of programs. RVMDES sets the name, data type, dimensions and initial values of widgets representing constant blocks to match the data for the block. In the example screen, the four widgets at the top center correspond to the algorithm's initializing constants. In general, only parameters expected to change at run time would be made initializing constants, but the designer has the option to give the operator access to any constant in this way.

For blocks representing hardware, RVMDES builds a label from the hardware module's type and ID number, as displayed in the hardware editor, and reads all other values from the hardware dictionary entry for the module's type. In addition, each field of a hardware parameter widget is labelled, because the fields have different meanings, and these labels are also read fkom the hardware dictionary. In the example, the two large widgets in the lower right corner correspond to the A/D and D/A modules. These widgets contain a large number of parameters, so a scroll bar is added, and the widget continues to use a reasonably sized window. The RVM's fork and merge modules are not represented, as the hardware dictionary indicates that they have no user tunable parameters.

Widget attributes are set to reasonable values by RVMDES, the user can edit most of them to be more meaningful or usable. Mainly, this involves changing labels and widget positions, and marking some widgets as visible only in configuration mode. In addition, the user can change initial values of the constants fiom the editor window, in case he did not bother to set them in the software editor, or if he knows hardware parameters better than the hardware dictionary does.

2. Statistic windows

RVMDES creates one statistic window for each algorithm output block partitioned onto the host interface module. Each uses the same name, data type and dimensions as its corresponding output block. Each takes the form of a matrix, like the parameter widgets, to show statistics. The user can edit the label, position and visibility of the widget, and can specify that it be replaced with a grey level display, treating the statistic data as an image with the proper dimensions. Before creating an image window however, the user should ensure that the data is of unsigned character or integer type, with values fi-om 0 to 255, if he hopes to be able to interpret the image. In the example screen, both outputs were correctly identified as statistics. The counter widget has also been renamed for readability. Recall that the name was abbreviated in the original block diagram, so that it could fit into the output block icon.

3. Buttons

In addition to the application dependent widgets and windows, RVMDES creates five buttons to allow the user to test and run an RVM, load and save parameter values, and toggle into and out of configuration mode. The user may only relocate these buttons and choose whether they are visible in configuration mode only. The single exception is the configure button, which must always be visible.

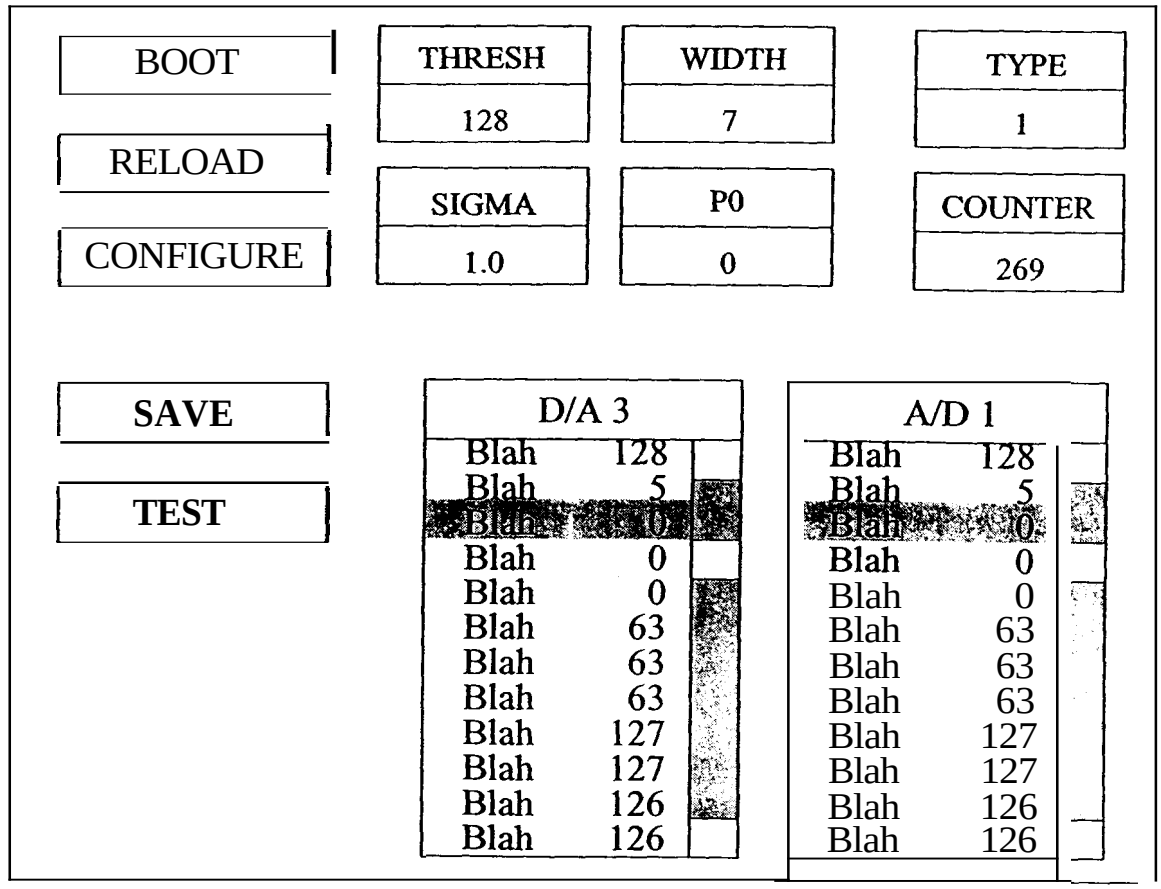


figure 7.2- interface screen

C. Configuration files

To simplify the process of building an interface program, RVMDES splits the interface code into two parts. One is the logic in the MMI program, which translates between widget commands and host interface commands. The other is the interface screen and other application specific data generated in RVMDES. To combine the two pieces, RVMDES writes its data to several files, and the MMI customizes itself to the particular application by reading the files. The data required to recreate a particular interface in the MMI is split over five files, each of which can be used and changed independently. This section describes what information is transferred by each file.

1. WGT file

The ***widget*** file lists the location and attributes of each parameter widget, statistic window and button in the interface. With this information, the MMI can recreate the interface screen. For parameter widgets, the file also includes the board and slot address of the module associated with the widget, and the command number for the parameter represented by the widget. Statistic windows do not have such information, but their order of appearance in the file is the order in which the MMI must fill them, using data returned

from the host interface. After all information for the interface, the WGT file contains the set of commands which must be sent to the host interface to remove modules from the RVM's global bus and start the RVM running. These will allow the MMI to start the RVM after it has booted all modules and sent any parameter initialization data.

2. INI file

The ***initialization*** file contains the parameter initialization data, for parameter widgets. The MMI reads the data to initialize widgets, immediately after creating them. It maintains the information in a separate file fiom the widget description so that the user has the option to reload or replace the file using commands in the MMI, without affecting widget information.

3. CFG file

The ***configuration*** file records information to reconstruct the **RVM** configuration generated by RVMDES. This is used during testing to draw the configuration diagram, and to determine whether it matches the actual **RVM** configuration. As such, it lists modules, overlays and cables that belong in each motherboard slot, jumper positions of each C44 module, and the location of each board in the system. It also lists the required connections between modules, which the MMI will test against messages returned by the host interface **as** test results.

4. GBI file

The *global* bus ***image*** tells the host interface module how to begin booting the RVM. It contains executables for processors, preceded by routing information including their length and the board and slot address of the modules that will receive. The file also records how much memory the host interface module should reserve for data that it will send to statistic windows after each frame. Unlike other files, the global bus image file is only read when the user commands the MMI to boot the RVM, at which time the MMI simply passes the file along to the host interface module.

5. TBI file

The ***test*** boot ***image*** file contains instructions to the instructs the host interface module, on how to boot the remaining **RVM** modules in test mode. Like the global bus image file, it is simply passed **to** the host interface at the user's command, where it is split and sent to the various modules. However, the format of the file is different. All hardware modules of the same type **run** the same test program, so the test boot file contains one program for each type of module in the **RVM.** At the moment, **this** means **an** executable for C44 modules, and the parameter lists needed to boot any custom hardware modules. After the test programs, the test boot file lists test information for each module, including the module's port and slot address and the type of module to expect there. **This** list presents modules in the same order **as** the configuration file, so that test programs will use the same module positions to record participants in connections **as** the MMI uses to decide whether those connections are correct.

D. Summary

RVMDES facilitates the design of an entire RVM by helping the user build interface software in addition to the RVM hardware. With the interface software, an operator can test and run an RVM, then tune it at run time by adjusting certain parameters. Because the tasks required of an interface program are consistent across image processing RVMs, RVMDES can rely on a precompiled executable containing an engine that performs these functions. Generating interface software then reduces to generating data files that customize the engine to a particular application. Most of this task is then handled by RVMDES, so the user only needs to arrange a set of controls on a simulated interface screen to produce an interface program.

VIII. A Complete Example

This chapter presents a complete example of how to design an **RVM** using RVMDES. It considers the problem of detecting whether text is present in an image. The contents of the potential text are unknown -- the algorithm must simply determine whether there its text present. The example **further** assumes that the application must operate at 30Hz, which is video frame rate.

The chapter follows the design cycle shown in figure 8.1. It begins with the software design cycle, which consists of writing an initial algorithm, then simulating and modifying it until it performs as desired on sample inputs. This is followed by the main hardware design cycle, which generally involves generating initial hardware, then iteratively finishing the design, finding the resulting run times, and modifying the design until the run times are fast enough.

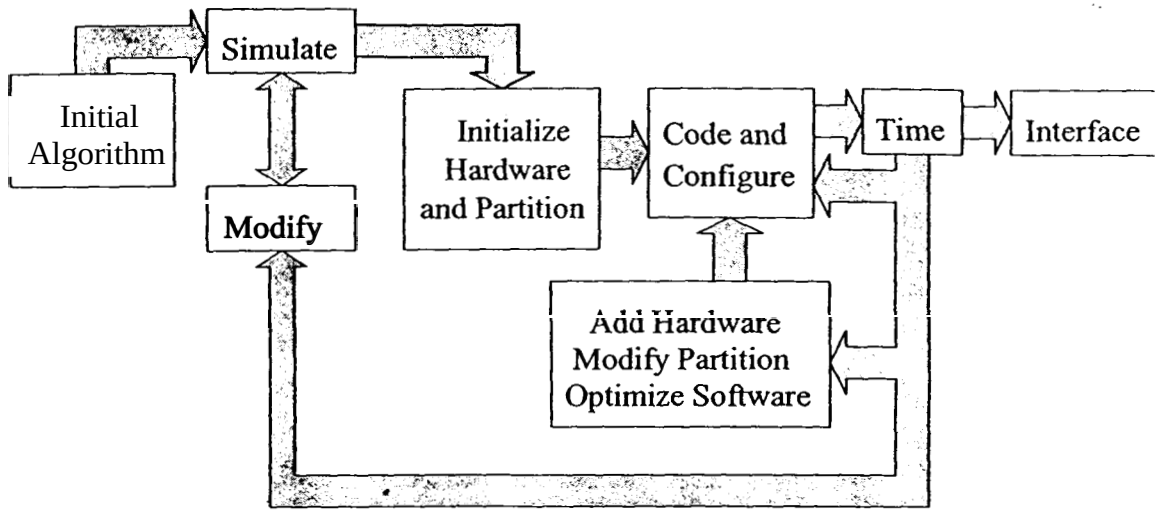


figure 8.1 - the RVM design cycle

A. Software Design

The first step toward designing the RVM is to build an algorithm that correctly identifies whether text is present in a set of sample inputs. This is a **two** stage processes of coding an initial algorithm, then simulating and modifying it until it works well.

1. Code initial algorithm

The problem statement calls for some sort of optical character recognition. One possible algorithm is **an** adaptation of the OCR agorithm described in [16], which detects the existance and location of text in video skims. It does not recognize individual characters, which is fine because the current problem does not specify which characters are involved. The only adaptation that must be made to the published algorithm is that rather than finding the location of text, the RVM algorithm should only recognize whether it exists. Figure 8.2 shows how the modified algorithm looks, coded in the RVMDES software editor.

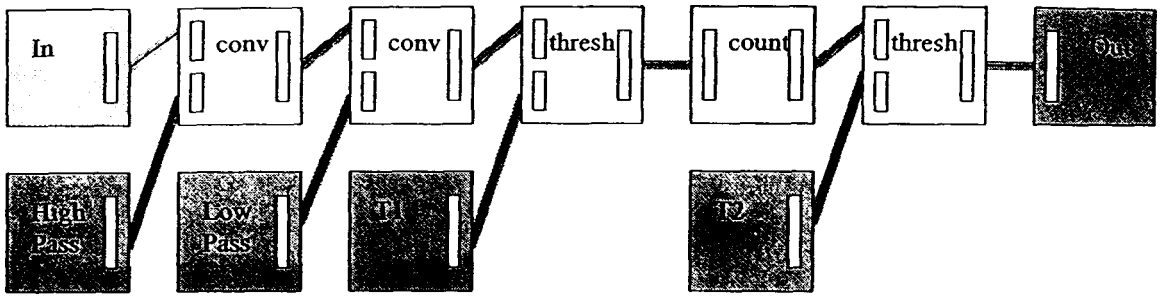


figure 8.2 - OCR algorithm

As shown, the algorithm convolves its input with a high pass filter to find the sharp edges associated with text, then with a low pass filter to spread news of the edge over several pixels. During this smoothing process, pixels inside text strings will reinforce eachother, while noise points will simply fade. The algorithm thresholds the smoothed image, to find text pixels, and counts the number of such pixels. If the number of text pixels exceeds a second threshold, the input image is determined to contain text.

In addition to creating and connecting blocks as shown in figure 8.2, the algorithm designer must set the parameters of several blocks. This is done fiom each block's specification window, which the user can raise by double clicking on the module. The left side of figure 8.3 shows the specification window for the low pass filter mask. Fields in the window allow the user to rename the block's icon and toggle its data type. These fields are common to most blocks. Because the block is a constant, the user can also choose the flavor of the constant, and fill in any supporting fields. The low pass filter mask will be a regular constant, the user can click the edit button to pull up a window like the one on the right of figure 8.3, in which he can edit the values. If the constant needed to be set at run time, like the emperically determined T1 and T2, the constant would be marked as *initialized* instead of *constant,* had been initializing. Then the user could set the initial value in the editor, or he could supply a filename fiom which to read the data. The remaining fields in the specification window hold the dimensions of the conFinally, fields common only to constants and inputs must be filled to show the inputs of the constant. Input and constant dimensions must be supplied before simulation or generating hardware. Also, the constant editor window will not appear until dimensions have been defined. Buttons next to the dimension fields can be toggled between constant and parameter. The latter will attach an input port to the block, fiom which can be attached to a dimension output port, to tie together dimensions of two blocks. This does not enable the editor.

In addition to setting the low pass filter, the user must key in the high pass filter mask, then two scalar thresholds. These thresholds are empirical, and may demand tweaking based on a lot of data, so lets make them initializing. We can still set an initial value based on sample inputs, but can then tweak it at run time.

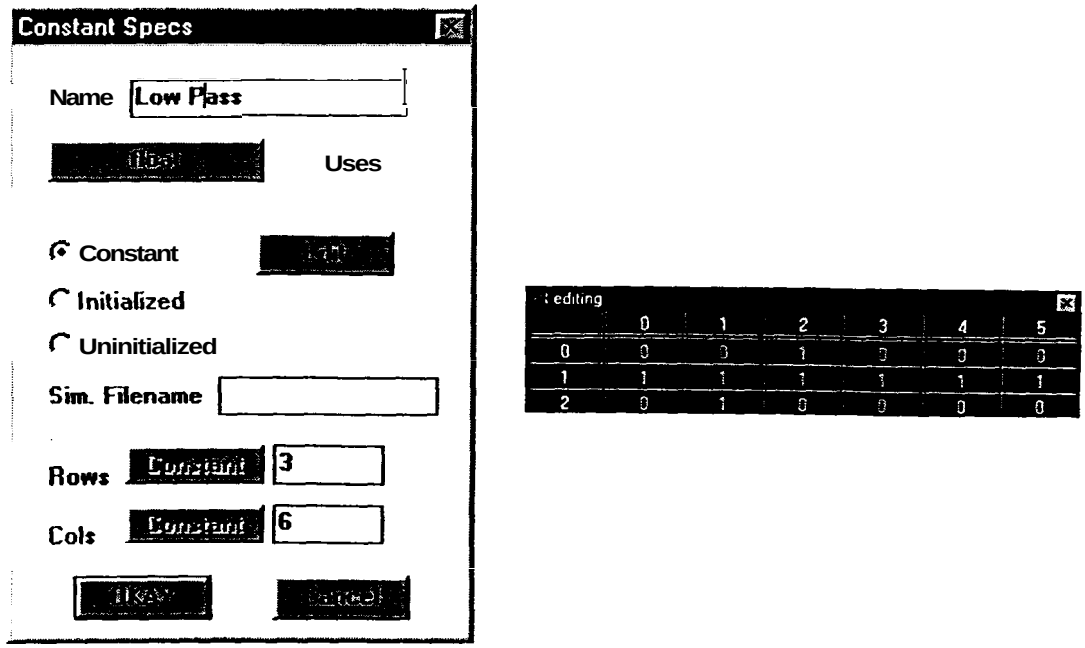


figure 8.3 - specification window and edit window for high pass filter

2. Simulate and modify

After coding an initial algorithm, the user should simulate and modify it as necessary so that it produces the correct inspection results for a set of sample inputs. To simulate an algorithm, the user simulates individual blocks by clicking on them while holding down the control key. If a block's inputs hold valid data and its outputs do not, simulating it will cause it to read or calculate its own outputs, and make them available on any connections from its output ports. Simulation must begin with inputs and constants, which have no inputs, then continue through functions and outputs as their inputs are given valid data. To show the user which connections contain valid data, RVMDES provides a simulation display mode, which will show connections with valid data in green, and those with invalid data in red.

Figure 8.4 shows the example algorithm being simulated. The simulation mode coloring shows that the user has simulated the first half of the algorithm, including the two filters and the first convolution. As the example algorithm diverges from the published one at this point, the user should look at the intermediate results of the algorithm here and in each later step, to see whether they match what he expects. If they do not, he should modify the algorithm. To view the data, he opens a viewing window by clicking on the output connection of the threshold module, while holding down the control key.

Suppose that the output of the thresholding operation is blank, indicating that nothing passed the thresholding, though the input to the algorithm contained text. This would

imply that one or both inputs to the threshold are bad, or that thresholding was not the optimum function to use. The user would first check the image input by clicking on it, to see that text areas have higher values than non-text areas. Assuming this is the case, the user would open the specification window of the constant input to the threshold, then open its editor window, to compare the threshold value with the values of the image being thresheld. Finding that the threshold is too high, the user would decrease it. This change will cause **RVMDES** to mark the output of the constant block and anything that depends on it **as** invalid. In figure **8.4,** this would be the outputs of the *T1* constant block and the output of the attached threshold block. The other connections maintain **their** data's validity, so the user only needs to resimulate the constant and threshold blocks to view the new results.

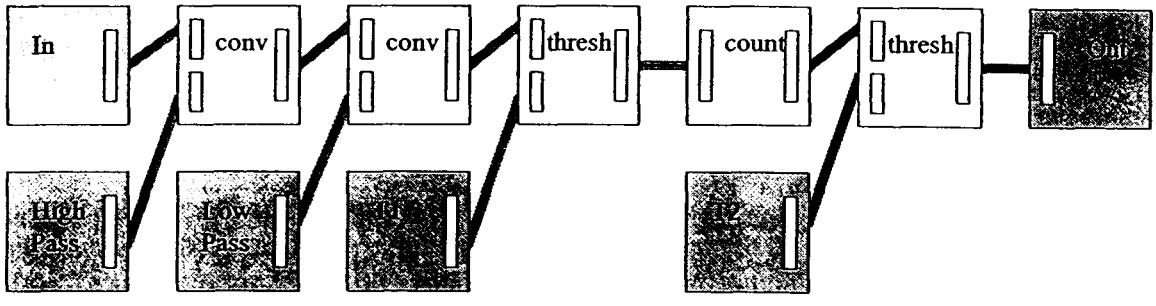


figure 8.4 - OCR algorithm being simulated

After several iterations, the output of the threshold should be binary 1 for text pixels and binary 0 for background pixels. If not, the user might consider replacing the threshold with a more complex operation, perhaps one that uses a threshold that adapts to the local intensities. The user would then delete the threshold modules, and connect on in its place. The inputs to the module would already be valid, being stored with the outputs of the convolution and constant blocks, so the user could immediately simulate the new block. This test-and-modifyprocedure should be repeated for the remainder of the algorithm, ensuring that each step performs **as** expected, up to the output.

B. Initial Hardware Design

Once **an** algorithm works, it must be converted into hardware. The user pushes a button in the RVMDES control panel to tell RVMDES to generate an initial set of hardware, write code for the processor module in this hardware and configure the hardware into **an** RVM design. Two of these steps require minimal help fiom the user, and RVMDES pops up windows to request that help.

1. Assigning output hardware

RVMDES knows that the initial design should include a host interface module to control the RVM, a C44 module to host all of the processing, an A/D module to provide the input, and something to provide the output. It generates a window like the one in figure 8.5 to ask how to assign that output. The left hand box contains a single entry, matching the sin-

gle output. The right hand side offers the types of hardware that could host it. Because the output is just a statistic, telling whether characters are present, the user would assign it to the host interface module, where it can be shown to the user and even trigger an alarm, without any additional hardware.

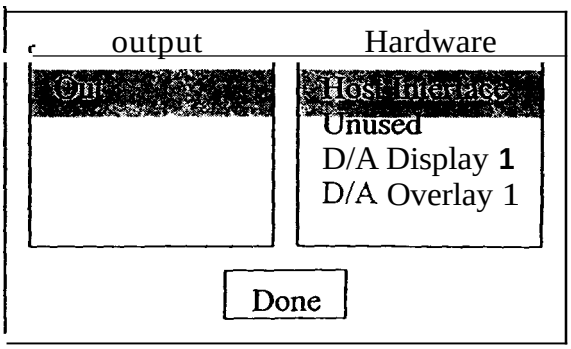


figure 8.5 - hardware wizard for OCR exmple

3. Configuring hardware

Once it has chosen hardwares, RVMDES will attempt to find the optimal way to assign it to circuit boards. As the design will only have three modules, **RVMDES** will quickly provide a diagram like that in figure 8.6, showing an optimal configuration. It will not realize that it has found the optimal solution, so it will continue looking until the user clicks the *stop* button. At that point, RVMDES will switch the stop button to a restart button, in case the user decides to return to look for a better configuration. It will also enable the ***timing*** button in the main control panel, so the user can continue with the design cycle. At any point while the configuration window is open, the user can click the print or save buttons to record the configuration, for use when assembling the **RVM** hardware.

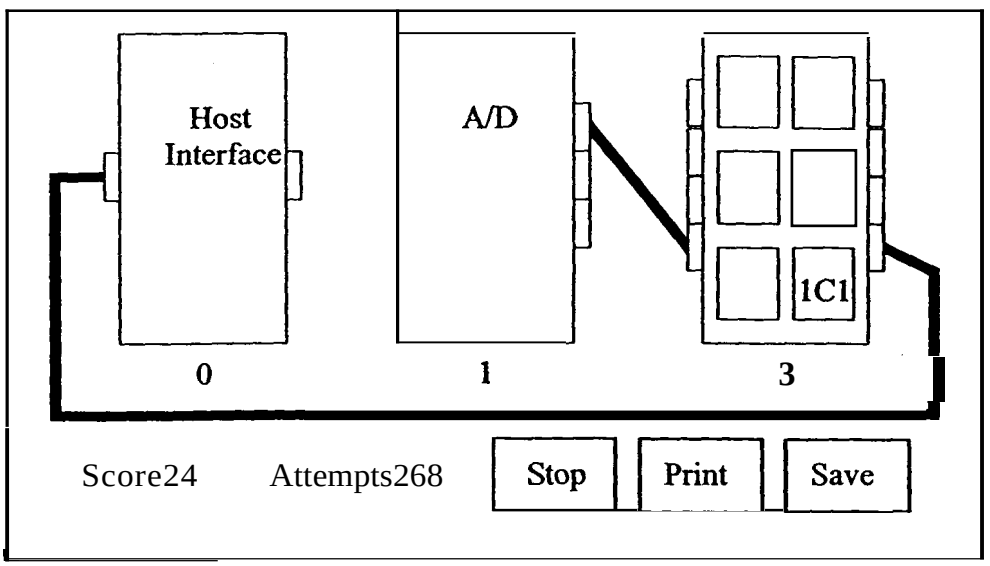


figure 8.6 - Configuration display for OCR example

C. Refining Hardware Design

The user continues the hardware cycle by iteratively finding the timing of the algorithm on the proposed hardware and modifying the hardware, software or partition of software onto hardware to improve the timing.

1. Timing

RVMDES's control panel includes a button labelled ***Timing.*** It will have been enabled when the user stopped the search for hardware configurations. Pushing the timing button will tell RVMDES to generate a timing diagram, fi-om which the user can check his RVM's speed and look for bottlenecks. Figure 8.7 shows the timing diagram RVMDES would generate for the algorithm, partitioned onto the initial hardware. The output on the host interface module is so fast that it does not even show up on the diagram.

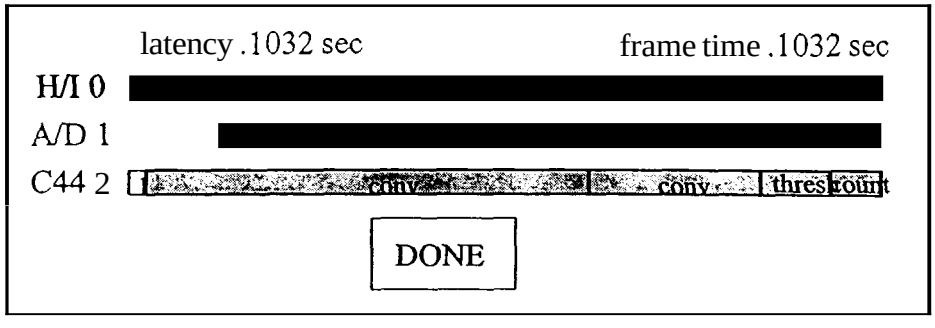


figure 8.7 - timing diagram for initial hardware in OCR example

2. Modifying hardware and partition

The timing diagram in figure 8.7 indicates that the RVM runs at about 10Hz, much slower than the specified 30Hz. It also shows that the single C44 is doing all of the work, and that the work could be split over multiple processors. The user would thus create a new C44 (module number 3) in the hardware editor, then repartition the first convolution onto it. The latter is done in a dialog window, called by clicking on the convolution block while holding down the *alt* key. The dialog asks for the identity of the module that should host the block, and the user would type 3, indicating the new module. The user would then dismiss the timing window and generate a new one, which will show slightly longer latency, and about one third less frame time.

The user will continue to iterate. He will duplicate the first hardware. Split rather than dealing, because you can. Then, instead of separating and doubling the second convolution, he will make a triple initial hardware and put the extra convolution on it. This will be almost fast enough, so optimize.

3. Optimize

show optimizer. show optimized and unoptimized programs. explain toggle. can combine thresh and count. can combine two convolutions, but why?

D. Finishing

build an interface, build an RVM, run the programs.

1. interface

asks for a file name, then builds those files.
will make two scalar constants, one a/d input, one output scalar.
you can rename and move around

2. assemble

3. test and run

IX. Conclusions

This thesis has presented RVMDES, a tool for quickly generating optimized machines for implementing image processing algorithms. In particular, it has covered how RVMDES takes advantage of the structure imposed by RVS hardware and image processing algorithms to provide tools for efficiently designing algorithms, hardware and interface programs, and for automatically optimizing programs.

A. Contributions

The principal contribution of the research described in this thesis is RVMDES itself. The structure of the tool not only puts it ahead of current design tools, but also anticipates future demands and provides a structure within which to meet them. While environments similar to RVMDES have been built to solve the limited problem of designing homogeneous, multiprocessor machines, RVMDES has several innovative aspects which allow a user to slash the design time for real time, image processing machines. In particular, the user can quickly optimize code, design RVM hardware and estimate system timing,

First, RVMDES provides semi-automatic optimization of image processing programs, and a function representation that facilitates it. Under normal circumstances, the only way to take advantage of inter-module optimizations would be to hand code a program, distracting the user fiom the task of quickly producing a machine. However, RVMDES uses a library of functions written the primitive definition format, allowing the user to quickly optimize his program but sparing him the digression of implementing and optimizing code, and the agony of trying to understand the optimized code later. While the user must still interactively decide which optimizations to apply, RVMDES determines which are available and translates them into code, so that in the optimization stage, the user can focus on the mapping between optimization and timing, not optimization and programming.the user only needs to be concerned with timing, not switching between timing and programming. This allows him to cycle faster.

Second, RVMDES automatically designs hardware layouts for RVM systems. Previous systems allowed users to build hardware nets, but left them the task of converting the hardware into a circuit board, or finding a (potentially nonexistent) configuration of a modular system that provided all of the required connections. By automating the layout process, RVMDES frees the user fiom both tasks. Further, because RVMDES can generate a minimal set of hardware as a starting point, the user's interaction with hardware is limited to deciding how adding hardware affects system timing and cost. Thanks to the reduced scope of the problem, design cycle time can be reduced.

Third, RVMDES provides a reasonably accurate method for instantaneously simulating the timing of an RVM system. Previous systems generally did not attempt this, and the ones that did required a development board, so that timing could be found empirically. With RVMDES, the user does not need to build the system to determine timing, and unlike emulators, RVMDES is able to model pipeline conflicts and other delays, because it records run times rather than trying to construct them. In addition, RVMDES can

include custom hardware, to provide timing for a heterogeneous system. By removing the burden of modeling timing interactions fiom the user, RVMDES increases the speed of the design cycle.

B. Future work

There are several possible directions for future work, all designed to increase the speed of the machines generated by RVMDES, or decrease the time required to design them.

First, the process of applying intermodule optimizations could be automated. The best choice of optimizations is input dependent, but the set of all possible optimized programs could be calculated, and timed on a fixed set of inputs, providing an indication of which set of optimizations was best on the average. The inputs could even be weighted, to match the frequency with which they would appear in th **RVM's** normal operation. Alternately, RVS hardware on a development **RVM** could provide a steady stream of real inputs. Timing a large set of possible optimized programs may be slow, but perhaps no slower than allowing the user to test a large number of them by hand. If optimization were automated, the user could focus solely on adding hardware, and could be guaranteed that the optimization would be at least as good as he could provide.

Second, following automatic optimization, the entire process of repartitioning and adding hardware to increase program speed could be automated. RVMDES already generates a starting configuration, including an initial hardware set and partitioning of the user's algorithm onto that hardware. The operators to change this configuration are also fixed, being limited to adding or duplicating hardware, and repartitioning or optimizing software. Heuristics already determine which of these operators should be applied, given the results of timing simulation, and could guide a search algorithm toward improved configurations. In general, the user has constraints on the latency, frame time and cost of the RVM, which would provide end criteria for a search. This search would not even need to be fast, because the user could simply write the algorithm, and leave RVMDES alone to generate the hardware. **As** an interesting twist, RVMDES might be combined with a program that automatically generates classification algorithms based on a set of inputs, so that the user could simply provide a set of classified images, and return once the **RVM** that classifies them has been designed.

Third, the remaining image processing optimizations should be implemented. This will probably require augmenting the primitive definition **to** record which functions can benefit from accumulators, what classes of functions can be reordered, and what circumstances warrant replacing a function with **a** lookup table. Shifted combination would also require some access to assembly code to use rotating buffers, if it is **to** be useful.

Along the same lines, RVMDES should be upgraded to fully optimize "bridge modules". These modules read and write data one pixel at a time, but do not provide **an** output pixel for each input pixel. Examples include projecting an image onto a column, and recording coordinates of sparse pixels marked binary **1. RVMDES** currently does not support serial merging across bridge module outputs, because the downstream module would need to

test whether and only operate if the bridge module produced a pixel, and the test would likely take up any time saved by the combination. However, it is probablypossible to reformat the function definition to allow RVMDES to determine where a pixel is assigned, and to insert the downstream module's kernel code there, obviating the test, and making the serial combination beneficial again.

Fourth, RVMDES should be expanded to include arbitrary dedicated hardware. For the moment, the only **RVM** dedicated hardware modules are for A/D, D/A and host interface, so hardware simulation is not well developed. Therefore, information about each (such **as** numbers of ports, allowable software assignments and relationship between software modules and hardware ports is hardcoded into RVMDES. Once there are enough modules fiom which to generalize behavior, RVMDES will need a way to input this information for new modules, and a way to time the new modules.

Fifth, DMA scheduling within each hardware module could be improved. Currently, inputs are randomly ordered and placed at the beginning of an executable, while outputs are similarly ordered and placed at the end. **This** could lead to a module waiting for an input that is not ready, yet while other inputs are ready and are ignored. It should be possible to reorder inputs and/or outputs or possibly even more inputs or outputs toward the middle of the program, allowing some processing before some inputs or after some outputs, if it would reduce idle time. This situation could also allow outputs to occur before inputs, so that some processing could be sent out to dedicated hardware and the results returned to the processor, which could operate on them.

Sixth, it would be nice to have intramodule optimizations. Particularly, we would like reducing trigonometry, and stripping out unused code, like the code to generate theta in a rho-theta Hough transform if it does not connect to anything. also internal reordering and sliding. **Also,** RVMDES code should use autoincrement addressing for its variables, which is much faster recalculating addreses in each iteration of the main loop. This autoincrementing could be built into definitions, but more likely would be built into the looping statements and addresses that are reduced fiom two dimensions to one.

Seventh, you could make an assembly hacking, intramodule optimizing, post processor. **A** simple search and replace strategy should allow RVMDES to exploit features of DSPs that cannot be called on fiom C, like parallel instructions, branch prediction, conditional loading, and a single cycle absolute value instruction. A more intelligent post processor might also be able to take advantage of situation dependent optimizations that the TI C4x compiler will not, like replacing a 5-cycle type cast (rounded to 0) with a single cycle truncation (rounded to negative infinity), or using rotating buffers **to** implement **shifted** combination in fast, on-board memory. Modifying assembly code produced by the TI C4x compiler to use these features can often double the speed of critical loops.

A related direction, which is probably a bad way to go, is to rewrite primitive definition body fields to use assembly language. Only the plus side, this would make both the assembly language post processor and a commercial compiler unnecessary, and should not require any changes to the basic structure of primitive definitions or concept behind code

generation. However, on the minus side, each definition would require assembly code for every processor available, for every function. It is unreasonable to expect a random user to write in assembly, much less in several assembly languages, and it is unreasonable to require that an entire new library of assembly be written when a new processor is made available. Therefore, a postprocessing intramodule optimizer, which can be written once for each new processor, seems to be a more promising direction.

Eighth, the timing simulator could be upgraded to use its current timing method by default, but to also be able to find exact timing using a development RVM attached to the PC running RVMDES. This method is not currently feasible since there is no such development board, but plans to build one are under consideration. With this dual timing method, many users could use the default timer to find ballpark timing during early hardware design on their own PCs, then move to a single PC with a development board to get exact timing for a final design.

Finally, a good direction to move would be look into optimization of parallel instructions for VLIW chips, like the TI C60, Philips trimedia, or Pentium with MMX technology, which will likely be the next processor added to the Egipt RVS. To optimize programs for these chips, loops must be split into four parts running parallel. This seems like a logical extension of pixelwiseness. A functionjust needs to record whether the order of processing is important. This would also allow **RVMDES** to write functions that run in orders other than reading order, like column major, or reverse reading. Could also expand to non-vision stuff, like MPG2 and DSP algorithms.

Appendix. Function Library

While RVMDES allows users to create their own image processing funtions, it also provides a library of 257 prewritten functions. The library is not intended to contain all image processing functions, just a representative set. This appendix lists the prewritten functions, divided into the categories RVMDES would display when the user chooses a new function module's function. The functions are listed With their name and a short description. Capitalized names mark functions derived from the SPIDER user's manual. Functions are listed once even if they apply to several variable data types, because RVMDES maintains a single definition for each, and modifies it during code generation to handle various data types and dimensions. For instance, alu_add can add two images, arrays or scalars, a scalar to each member of an image or array, or an array to each row or column of an image, and can do this with any combination of data types, but only constitutes one function.

14 ALU FUNCTIONS

alu_add, alu_sub, alu_mult, alu_and
alu_or, alu_xor, alu_not, abs_val
alu_reciprocal, abs_val, negate, sign
alu_divide_by_const, alu_divide_to_float

25 TRANSFORMS

RQNT - requantize grey levels
INVFA1, INVFA2, INVFA3 - Fourier deconvolution: divide images, attenuate outside a circle
scale - to specified grey level range
normalize2 - rescale grey levels fiom 0..oldmax-1 to 0..newmax-1
normalize3 - rescale grey levels to 0-255 range
normalize–4 - rescale input to maximum value 1.0
normalize–5 - rescale input to area 1.0
lookup_table - convert each pixel
HTBL2 - make lookup table for exact histogram transform
GTRN2 - apply lookup table for exact histogram transform
THST - apply image lookup table to image's histogram
alu_log, alu_ln, alu_sqrt, alu_exp - apply function to each image pixel
alu_log–fast, alu_ln_fast - make lookup table and apply to each pixel
Hough - rho-theta Hough transform, using a single theta per input pixel
zzhough - main loop of Hough
curvature - find H and K curvature of each image point
zzcurvature - main loop of curvature fmding
chain_to–coords, coords_to_chain - convert between the two representations

22 HISTOGRAM OPERATIONS

histogram - make a histogram
fill_histogram - main loop of histogram

histogram_masked - make histogram using only some image pixels
fill_histogram_masked - main loop of fill–histogram–masked
fill_hist_buckets - make histogram with other than one bin per pixel
LHST - histogram of laplacian of image
HIST2 - threshold reference image and use as mask for histogram)
FHST - apply laplacian, threshold at some %, use as histogram mask to insure bimodal.
DHST - histogram of sum of differences between pixel and lesser 8-neighbors
cuml_hist - convert regular histogram to cumulative histogram
combo_hist - histogram of points fiom list of coordinates
hist–stas3 - histogram of points fiom list of coordinates,
make hist–lookup–1 - fiom actual and desired mean and variance
make_hist–lookup–2 - similar, but moves mean
HTBL1 - make histogram transform table between 2 histograms
HTBL2H - main loop of HTBL2, to make lookup table for exact histogram transform
hist_make_xform_lookup - fiom two histograms
split–histogram–da - find threshold at middle of 2-gaussian fit
split–histogram–& - find threshold at maximum in difference histogram
split_histogram_percent - find threshold to make a fraction of pixels 1s
HGTR1 - histogram transform with no subdivision
HGTR2 - histogram transform with subdivision
HGHY1 - histogram hyperbolization with no subdivision

10 THRESHOLDINGS

thresh_clip - clip to 0 anything below threshold
thresh_clamp - clamp anything above threshold to that threshold
threshold min, threshold–max - output is minimum or maximum of two inputs
threshold– binarize at a threshold
threshold_general - out = (in1>in2) ? in3 :in4;
hilite - hilite any pixels above a threshold
thresh–2–inc, thresh_2_exc - zero pixels between or outside two threshold, inclusive or exclusive
thresh_eliminate - zero any pixels matching a threshold

21 DATA CREATORS

fill, copy - fill a variable with a scalar or a copy of another variable
table–init - fill with zeroes
make_hyperbole–filter - for histogram hyperbolization
make_hi_pass_rect - binary 1 inside the rectangle, binary 0 outside
make_lo_pass_rect
make_band_pass_rect
make–band_reject–rect
make_hi_pass_ring - binary 1 inside disk, binary 0 outside
make_lo_pass_ring
make–bandqass–ring
make_band_reject_ring
make_hamming - circular high-pass filter, weighted more towards center

make_hanning - similar to hamming
make_dx_filter, make_dy_filter, make_dxy_filter - for convolution in Fourier domain
make_laplacian, make_gaussian - for convolutions
make_log_lookup, make_ln_lookup - make lookup tables

28 STATISTIC GENERATORS

find_covariance
zzfind_covar - main loop of find_covariance, given means
cooccur_corr - find correllation from cooccurance matrix
correl_coeff - find correllation coefficient from covariance
img_moment - multiply each pixel by its row number
y_moment - find first moment of an array
moment - find any moment of a variable
barycenter-moment - find any moment about the center of gravity
inv_dif_moment - a weird moment weighted heaviest along x==y
entropy, cross_entropy - sum of P*log(P) or P*log(Q), over image or array
find_mean, find_variance
NORMx - find image's mean and stddev grey level
region_area_one, region_cog_one - find area and center of gravity of one labelled region
region_mom_all, region_area_all - find first x and y moments of all labelled regions
sum_squares, sum_pixels
find_min, find_max
find_max_index, find_min_and_index
count_nonzero - count nonzero pixels
dot_product - between two arrays
otsu_binarize - use moment information to find best split in a histogram
COF9_main - a strange feature, calculated from a coorccurance matrix and two projections

6 PER-WINDOW STATISTIC GENERATORS

variance, variance_structured - find variance under 3x3 window or arbitrary masked window
connect_4, connect_8 - find connectivity of binary image
crossing_4, _8 - find crossing number of binary image

13 BINARY MORPHING AND THINNING FUNCTIONS

binary erode, binary_dilate - morphology on 3x3 windows
binary_erode_structured, binary_dilate_structured - morphology on arbitrary masked windows
morph_bin_open, morph_bin_close - more morphology
skeleton_from_distance - extract skeleton from distance image
THNG3, thin_deutch_1, thin_deutch_2 - Deutcsh's binary thinning algorithm
wave_prop_thin - Shikano's wave propagation thinning algoritm
SRNK4 - *shrink* object by deleting points with connectivity 1
TEMX1 - thinner for texture edges found by TXEG2

SMOOTHING OPERATIONS

ASMT - Yokoya's edge preserving smoothing)
zzASMT - main loop of Yokoya's edge preserving smoothing
TEPA3 - texture bounary preserving smoothing, final of 3 steps
TEPA1 - moving average filter
TXDF21 - rotated moving average filter
EGMV_split - Merro-Vassy edge detection, preliminary smoothing
SRNK3_smooth - remove spurs
min_5n - replace pixel with minimum of itself and its 4-neighbors
nonmaximum suppression

37 EDGE FINDERS

edge_find_prewitt, edge–find–roberts, edge_find_sobel - convolutions
sobel_v, sobel_h - component loops of Sobel operator
edge_templateqrewitt - several convolutions, take highest result
edge–template–robinson
edge–template–kasvand
laplacian_1 , laplacian_2, laplacian_3 - convolutions
TXEG2 - update images of best line size, direction and strength
TXEG2H - main loop of TXEG2
canny
kts_bin - binarize, marking points and edges
kts_tmc_h_slow, kts_tmc_v_slow, kts_bin_h, kts_bin_v - component loops of kts_bin
ITEN1 - iterative edge and line enhancement
ITEN1A - one convolution for ITEN1
RXEP - initialize probabilities for relaxation based edge enforcement
EIKV1 - kasvand's iterative edge detection
ITEN2 - iterative contrast enhancement
ITEN2H - ITEN2 check with one mask
EGMV - Merro and Vassy's edge finding, magnitude only
EGMVplus - Merro and Vassy edge finding, magnitude and direction)
min_ddx, min_ddy - image derivitaves, or smaller side derivative if main one is too large
find–extrema–h, find–extrema–v - binarize, marking local maxima in either direction
RCRS1 - average absolute difference with 8-neighbors
TEPA2 - texture edge detector
TXDF1h, TXDF1v - horizontal and vertical edge detectors
TXDF22 - angle-specific edge finder
LHSTB - pseudolaplacian convolution

3 VECTOR MANIPULATORS

outer product
transpose
MTRX - matrix multiply

7 DATA MERGERS

ITEN2B - iterative contrast enhancement, merging step

RMS - pixelwise root-mean-square of two variables
sum_of–squares - faster than root-mean-square by not taking the root
max–of–4, _8 and _9 - pixelwise maximum value over several variables
atan2 - arctangent, for finding direction

9 SEGMENTATION OPERATIONS

RCRS2 - threshold adjustment for Yokoya's relative similarity based segmentation
region–label - standard region labelling of binary image
label_1, reconcile, move–to–top - component loops of region–label
RMRG2 - Brice and Fenema region merging across weak boundaries
RMRG2_list, RMRG2_phagocyte, RMRG2_weakness - component loops of RMRG2

17 IMAGE WARPINGS

affine_xform_n4, _n8, _nn, _nmin, _nmax - affinetransform, several interpolation schemes
nonlin_xform_n4, _n8, _nn, _nmin, _nmax - nonlinear transform, y=Axx+Bxy+Cyy+Dx+Ey+F
projective_xform_n4, _n8, _nn, _nmin, _nmax - y=(ax+by+c)/(dx+ey+f) etc
rect_to_polar - unroll circles to rectangles around the image center
chunky_ring_xform - unroll circles into rho and theta bins of specified size

6 RESAMPLING FUNCTIONS

downsample_min, downsample_max, downsample–avg - reduce resolution, 3 ways to interpolate
downsample - reduce resolution by an integer amount, no interpolation
upsample–integer - increase resolution by pixel replication
resample–by–center - arbitrary magnification, no interpolation

13 DISPLAY FUNCTIONS

pad - add zeroes to an image's bottom and right borders, to reach a certain image size
show–histogram - draw a binary image showing a histogram
show–threshold - overlayed on a histogram
overlay2 - draw lines representing nonzero points on a Hough plane
overlay3 - as overlay2, shifted and scaled to match the Hough plane
overlay–lines - fi-om a list of rho-theta pairs
hilite_borders_04, –14, –08, _18 - on binary image, mark edges as 2s
segment - color planes, cylinders and spheres based on curvature image
color–scale - for segmented regions, display using maximum variation of 24 bit color
overlay_points - highlight points fiom a list of coordinates

8 CONVOLUTION

convolve, convolve–horiz, convolve–vert - image processing convolution of windows, rows, cols
dconvolve, dconvolve_horiz, dconvolve_vert - electrical engineering convolution
SAD and SSD - image processing correllation, unnormalized

4 PROJECTIONS

project–col, project–row - project an image onto a row or column
fill_row_projection, fill_col_projection - main loops in projection operators

4 MISCELLANEOUS OPERATIONS

sum_pos–and–neg - split image into one image with the positives and one with negatives
div_pos–and–neg - pixelwise divide by either of two images, depending on main image pixel sign
list_nonzero - list coordinates of nonzero pixels in an image
DSTT 1H - pixelwise difference between an image and itself shifted down and right

[15] Webb, J.A., "Steps toward architecture-independent image processing," Computer, Feb. 1992, pp. 21-31.

[16] Smith, M. and T. Kanade, "Video skimming and characterization through the combination of image and language understanding techniques," CVPR '97, 1997, San Juan, Puerto Rico.