A Dataflow Environment for Real-Time Image Processing Applications

Terence Arden and Joseph Poon
Logical Vision Ltd., Vancouver, B.C., Canada

Abstract

A dataflow environment based on a client/server approach, called WiT, enables dataflow graphs to be executed efficiently with little overhead. Data tokens are managed by reference and reside on servers until either data is requested for viewing or required by another server. An enhanced fire-on-any behaviour greatly simplifies the design of many simple graph constucts such as multiplexors or crossbars which are overly complicated when implemented with classical dataflow contraints. *Sync* tokens are used to accommodate the need for synchronizing data, especially useful when controlling hardware. A hierarchical scheduler maintains execution sequence in a logical progression across multiple sub-graphs to provide a server an opportunity to generate well structured standalone code suitable for real-time target hosts. An example of WiT using hundreds of nodes and links to model Datacube devices for a realistic application is presented. The use of hierarchical operators serves to reduce such a complex application to a manageable level.

Introduction

The design and construction of real-time image processing applications can be a time-consuming and costly process. Ideally, software tools should support the development of algorithms on a variety of platforms at relatively low cost and allow for the migration to higher performance hardware when needed.

WiT is a visual dataflow programming environment for image processing algorithm development. Its CAD-like environment offers considerable leverage over C-based programming and text-based libraries. The client/server structure of WiT allows the user to migrate to higher performance hardware without changes to the conceptual design represented in a dataflow graph. Dataflow graphs in WiT can employ conditional flow controls, looping, sequencing through a list of data objects, unlimited arc branching, and unrestricted flow direction. Data anywhere on the graph can be inspected without structural modification to the graph. There are other visual programming packages available[1,2,3,4,5,6], some are commercial products and others are research projects. A comparison between WiT and these packages have been described elsewhere[7]. WitFlow, a subset of WiT, contains a server for Datacube's ImageFlow and allows the development of applications on Datacube's MaxVideo 200 pipeline image processor. Models for the MaxVideo 200 and Digicolor are currently included. Programmers can design pipeline processing tasks with a mouse and create ImageFlow code that runs without performance penalty.

In this paper, we briefly review the advantages and drawbacks of the dataflow concept in general, then we discuss the enhancements that WiT made to classical dataflow which enables it to model and control hardware processes, with emphasis on execution efficiency, scheduling behaviour, and synchronization. Finally, an example of WiT using hundreds of nodes and links to model Datacube devices for a realistic application is presented.

Review of the Dataflow Concept

Dataflow has a number of advantages over common control flow (procedural) languages such as C or FORTRAN. It is visual and intuitive, which lends itself well to graphical, visual programming. It is inherently parallel in its description of algorithms, which makes it suitable for programming parallel computational systems.

Unfortunately, the dataflow paradigm suffers from a number of significant drawbacks which has prevented its widespread acceptance. The main weaknesses of dataflow systems are related to speed of execution, node scheduling semantics, and data synchronization. The success of delivering on the promise of using dataflow for rapid design and execution flexibility relies greatly on how dataflow systems are implemented.

In the classical dataflow paradigm, data tokens flow along arcs into computation elements (nodes), which may accept any number of input tokens and produce any number of output tokens. Copies of the same token can be sent to multiple nodes. A node is ready to fire when all its inputs are available. After firing, all the inputs are consumed and all the outputs are sent. Figure 1 shows an example of a dataflow implementation of the equation

$$x = \frac{-b - \sqrt{b^2 - 4 \times a \times c}}{2 \times a}.$$

Execution Efficiency

The most serious drawback in dataflow is execution efficiency, since there is considerable data movement as

indicated by the number of arcs in a graph. For example, there are 16 arcs in Figure 1. If token travel on each arc is implemented by actual physical transfer of data, then the overhead can be prohibitive.

The number of physically available processing elements is typically much less than the number of nodes in a dataflow graph (e.g. even the simple graph in Figure 1 has 15 nodes). This means that there must be some form of scheduler or arbitrator which can decide which processing element will be assigned to an operation (node). Unless the scheduler is efficient, it may cause unbalanced work distribution among the processing elements, causing them to sit idle unnecessarily, or the scheduler itself may become the bottleneck of the entire computation system. Unless great care is taken to ensure execution efficiency, overhead costs make dataflow suitable only for coarse grain parallelism.

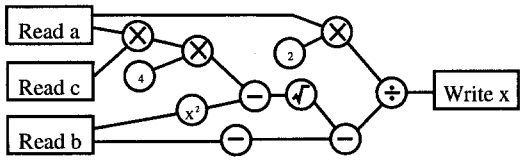

Figure 1. Dataflow graph example

Firing Requirement

The requirement that a node must have all its inputs before it can fire and to produce all outputs when it is finished ensures correct firing sequence. However, this requirement can sometimes cause unnecessary difficulties in realizing simple applications. Consider an example of a resettable counter shown in Figure 2. The *T* and *F* operators are necessary to satisfy the requirement that *counter* can only fire if all its inputs are available, although sometimes we want the counter to count, while other times we want it to reset. Figure 2 may seem like an elegant solution, but consider the behaviour of the box outlined in the dashed line. We can consider the dashed line box to be a new operator which encapsulates the behaviour of the sub-graph inside it. Then to the outside world, the dashed operator behaves as though it can fire even when only one input is available. In other words, the complexity caused by introducing the *T* and *F* operators is unnecessary if an operator can be optionally defined such that they can fire when *any* of its inputs is available.

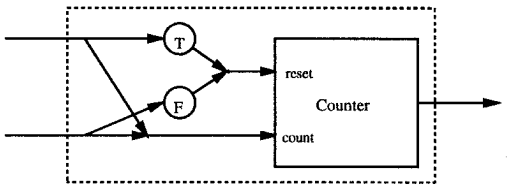

Figure 2. Classical dataflow implementation of a reset-table counter

The scenario illustrated in Figure 2 is much more commonplace than one might initially think. As an analogy to functions or subroutines in procedural languages, it is common for a visual programming environment to allow the user to group a collection of operators together and make it into a new operator. Since the graph represented by the new, hierarchical operator can start processing when any of its inputs is ready, it is only logical that the operator that encapsulates the graph can fire if any of its inputs is ready also.

Synchronizing With Variable Number of Inputs

Suppose we wish to use a dataflow language to control hardware. We want to set some of the registers in the hardware to some particular values, and activate the hardware when that is done. This may be represented as in Figure 3.

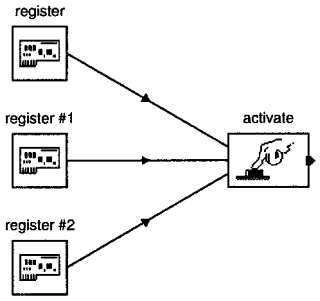

Figure 3. Synchronizing hardware control

As the *activate* operator will only fire when all its inputs are available, we are guaranteed that the hardware will only be enabled when all the registers are properly programmed. However, *activate* must have as many input ports as the number of registers which we want to program. In situations where there are a huge number of registers that need to be programmed, or when some registers can be left at default values so that the total number of inputs to *activate* is variable, classical dataflow becomes inadequate.

Enhanced Dataflow in WiT

In order to overcome the limitations imposed by traditional dataflow systems, WiT employs an enhanced dataflow scheme which allows it to tackle real-time applications without compromising the advantages of dataflow.

A client/server approach is used to model the separation between how graphs are designed from the way in which operations are carried out. The client implements the design layer through a CAD-like interface (GUI) where graphs are interactively created to form algorithms and automatically scheduled to run. Servers, on the other hand, run as separate processes to perform the computations in a graph by communicating with the GUI using a suitable inter-process communication method. WiT supports both TCP/IP and DDE protocols.

The servers can be run in parallel across numerous machines which exploit special hardware devices or simply perform everything in software.

The division of work in a client/server scheme offers two advantages. First, the algorithm designer is given the freedom to rapidly express how problems are solved regardless of how underlying computation is actually performed. Second, the mapping of graphs to an appropriate technology allows many possibly different execution platforms (e.g. software or real-time hardware) without forcing changes to the conceptual represented in a graph. This gives the developer a path to migrate slow software-based graphs to high performance hardware without significant re-engineering.

Execution Efficiency

Token travel in WiT is achieved by simple manipulation of a handle to the data object, not the actual data itself. This reduces the overhead associated with data movement to a negligible amount. When multiple servers are available, actual data transfer has to take place. The GUI component still handles each data object by its handle, but the actual data itself may reside on any of the servers, or the GUI itself. The GUI keeps track of which server or servers has a copy of the actual data, and the GUI scheduler will schedule operations so that data transfer is minimized. This will be discussed in more detail in Section 4.

Simple operations such as adding two numbers or concatenating strings are handled by built-in operators which always run on the GUI, thus reducing the overhead of fine-grain operations. Although profuse use of such small operations should be avoided since there is still significant overhead compared to compiled code, occasional use of such operators can greatly reduce the number of special operators required and improve readability and ease of maintenance of algorithms.

Firing Requirement

WiT supports both fire-on-all and fire-on-any operators. Special treatment is given to hierarchical operators, see Section 4 for details. In addition, operators can control whether all, some, or none of their outputs are to be produced. Outputs may or may not be connected in a graph.

Synchronizing With Variable Number of Inputs

A special token type, *sync*, is introduced in WiT to simplify the handling of synchronization requirements. Normally, with regular data tokens, if multiple links are joined in a junction, each token that arrives from any of the incoming links is transferred to the outgoing link. For example, if there are *N* input links to a junction and one output link from it, then *N* tokens will be sent to the output link. However, if the token traveling on a link is a *sync* token, the link junction behaves differently. When a *sync* token arrives at a junction, instead of being transferred immediately to the output link, the junction waits till a *sync* token arrives from *each* of the input links before sending a *single sync* token to the output link. Because there is no limit to the number of links connected to a junction, the *sync* token can be used to ensure proper execution synchronization even when the number of inputs to an operator is variable.

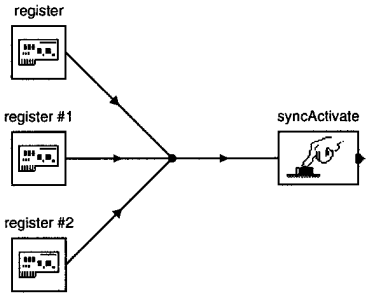

Figure 4. Synchronizing hardware control with sync tokens

To illustrate, if we consider the situation in Figure 3 again, we can replace it with the graph shown in Figure 4, where each operator which programs a register produces a *sync* token. Now we can change the number of registers we wish to program without changing the number of input ports of *syncActivate*.

Scheduling Considerations

The scheduler plays a vital role in WiT, since scheduler efficiency has a direct impact on overall execution speed. Also, it provides a hierarchical mode which is useful when generating program code on a server.

Data Transport

Token data generally are not transported between the UI and servers or among the servers themselves unless it is necessary. Often an entire algorithm, including input/output operations, can be executed on a single server without *any* data transfer at all. Data transfer is necessary when the object needs to be displayed, or when the server that has data does not support a required operation.

When multiple servers are involved, the WiT scheduler maintains information about where each data object resides. When an operator needs to be scheduled, the scheduler assigns it to the available (not busy) server which requires the least amount of data transfer.

When an arc branches into parallel branches, copies of the token data will be physically copied to multiple servers. But after this initial copying, the servers will be able to execute in parallel.

Server Specific Operators

Servers in WiT can support different sets of operators. For example, it is possible to have a general purpose server (call it server A) which supports all operators, and another (server B) which utilizes special hardware for filtering operators only. So there are some operators that run only on server A, some that run only on server B, and some that can run on either. The WiT scheduler maintains all this information and schedules an operator

only on a server which supports it. When an operator is supported by multiple servers, the server name that appears first in the definition of the operator is given priority.

Flat and Hierarchical Scheduling

When scheduling hierarchical operators, WiT provides a choice of flat or hierarchical scheduling. With flat scheduling, the hierarchical operator behaves as if it is a graph. When inputs arrive at its inputs, they will be sent directly to the underlying operators within the hierarchical graph. When any output token is ready, they are sent immediately downstream. This allows the scheduler to maximize parallel execution, which in turn should result in faster overall speed.

With hierarchical scheduling, the hierarchical operator behaves like a fire-on-any primitive operator (one that is implemented directly in C). The WiT scheduler finishes all schedulable operators within a graph level before descending into a hierarchical graph. The effect is that input tokens tend to collect at the inputs ports of the hierarchical operator. Similarly, the WiT scheduler only leaves a hierarchical graph when all schedulable operators are scheduled. The effect is that output tokens tend to collect at the output ports of the hierarchical operator before they are sent downstream. Hierarchical scheduling is useful for producing better structured program code where code fragments correspond to the operations within each hierarchical node. Flat scheduling tends to create 'spaghetti' code as nodes from unrelated parts of a graph can fire in an interleaved fashion.

A Real-Time Application

A special server called WitFlow was developed to manage simultaneous pipeline execution of dataflow graphs on Digicolor and MV-200 hardware from Datacube[8]. WitFlow presents each hardware device as a set of hierarchically nested graphs which offer a high-level interface to programming a massive set of low-level registers. Each node in a WitFlow graph represents a basic building block used in defining an imaging pipe. There are multiplexors and crossbars for controlling dynamic data pathways and convolvers and lookup tables to provide image processing primitives. Several models, say a Digicolor and MV-200, may be connected in a graph to pass real-time RGB channels from one board to the next. The results of hardware pipes can be uploaded into WitFlow for continued processing by the host CPU to finish an application. For example, other servers may be exploited to interface with programmable logic or robot controllers.

As an example of a real-time application programmed with WitFlow, consider the problem of tracking a sailboat as it moves over a lake (Figure 5). The field of view may contain clouds, trees, and small houses at the lakeside. We wish to track the boat using real-time hardware, Digicolor and MV-200, for color acquisition and area parameter computation. The result will be a simple graphic used for overlay on periodic video frames to mark the sailboat position.

Figure 5. A sailboat image

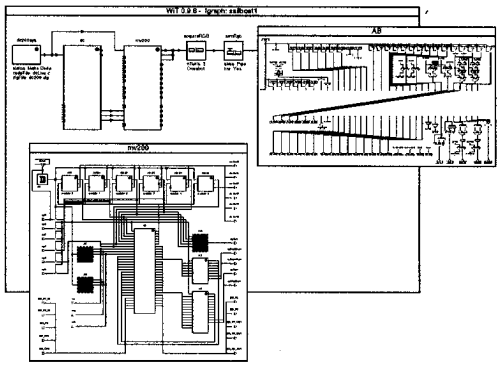

Figure 6. Acquisition graph for real-time tracking of sailboat

This problem is readily solved by instantiating a Digicolor and MV-200 model, represented as hierarchical icons, in the WitFlow workspace. The icons are connected by links to feed live RGB data into the MV-200 from the Digicolor (Figure 6). The MV-200 routes the incoming signals through a crossbar switch into three image memories. The same signals are transmitted, in parallel, from the memories through a LUT to a network of ALUs which apply the distance function $d = \sqrt{(r-rt)^2 + (g-gt)^2 + (b-bt)^2}$. The d term is compared to an acceptable threshold representative of a sphere of allowable color vectors. The rt, gt, and bt terms represent the RGB threshold, i.e. the center of a sphere with radius d. The subtraction and squaring operations are performed by a LUT in each of the memory devices. The addition and final square root is implemented with an ALU network and 12-bit LUT. The entire equation requires a single pipe. This threshold method is used twice to isolate bright white and dark reddish objects corresponding to the sail and hull colors of the sailboat. As a result, two 8-bit binary images are generated which are used as the input to a 3-stage blob parameter pipe. The blob pipe produces a bounding box and pixel area measurement for each region in the binary images. The measurement or feature vectors are uploaded to WitFlow from the hardware and used to select a pair of related blobs from both color thresholds which satisfy the defi-

nition of a sailboat, e.g. triangular white sail above wide, narrow reddish hull.

The process is completed by displaying a symbol on an uploaded snapshot from the live video feed. This cycle is repeated continuously where the effective match frequency is $t = 5 \times 512 \times 485 \times \frac{1}{P_r} + T_m$ where 5 pipes are required to operate on a 512 × 485 image at a rate denoted by P_r, the pipe rate the case of the MV-200, P_r is 20 Mhz. T_m is the time the host CPU consumes in cycling through a feature vector searching for a sailboat match.

Acknowledgements

Partial funding for this work was provided by the National Research Council of Canada.

References

1. *AVS Technical Overview*, Advanced Visual Systems Inc., 300 Fifth Ave., Waltham, MA 02154, 5.0 edition, 1992.
2. *IRIS Explorer User's Guide*, Silicon Graphics Inc., 1.0 edition, 1991.
3. *A Visual Programming Language for Visualization of Sci entific Data*, PhD thesis, University of Illinois at Urbana-Champaign, Department of Computer Science, July 1992.
4. Neil Hunt, "Graphical Programming Techniques for Image Processing on Single and Multiple Processor RISC Systems," *Proc. of Electronic Imaging West*, Mar. 1992.
5. *Knowledge-Based Vision Systems: Visual Programming Environment*, Amerinex Artificial Intelligence, Inc., 409 Main St., Amherst, MA 01002, 3.0 edition, 1993.
6. J. Rasure and C. S. Williams, "An integrated dataflow language and software development environment," *Journal of Visual Languages and Computing*, **2**, 1991.
7. M. S. Atkins, T. Zuk and B. Johnston, "Role of Visual Languages in Developing Image Analysis Algorithms," School of Computing, Simon Fraser Univ., Burnaby, B.C., Canada.
8. *Datacube Image Processing Manual*, Document No. SM101, Datacube Inc., Dec. 1991.