A Functional Data-flow Architecture dedicated to Real-time Image Processing

J.SÉROT, G. QUÉNOT and B. ZAVIDOVIQUE

Laboratoire Système de Perception, DGA/ETCA, 16 bis Av. Prieur de la Côte d'Or, 94114 Arcueil Cedex, France

Abstract

This paper presents a data-flow computer developed at ETCA and dedicated to real-time image processing. Two types of data-driven processing elements, dedicated respectively to low and mid-level processings are integrated in a regular 3D array. Its design relies on a close integration of the data-flow architecture principles and the functional programming concept.

Image processing data-flow graphs, first expressed using a functional syntax are directly mapped onto the processor array.

The programming environment includes a complete FP-specification to network configuration compilation stream along with a global operator database.

An experimental system, including 1024 low-level custom data-flow processors (6 $\times$ 25 MBytes/s, 50 million operations per second) and 12 T800 transputers , was built and several image processing algorithms were run in real time at digital video speed.

Keyword codes: C.1.3; D.1.1; I.4.0

Keywords: Data Flow Architectures; Functional Programming; Real Time Image Processing

1 Introduction

Real-time image processing, requiring a computing power in the range of billions of operations per second, is still far beyond the capacity of current general purpose computers.

Most of the computationaly demanding tasks involved in such processing exhibit an inherent parallelism. This parallelism may be exploited by an architecture providing a high degree of pipelining.

On the other hand, previous work in the field of image understanding [7] showed that many image processing algorithms are based on the distinction between low, intermediate and high level of processing, and that distinct processors are required to handle each of these levels under real-time constraints.

This served as a basis for the development of a high performance computer architecture able to integrate different types of processors under an unifying programming model and dedicated mainly to real-time image processing.

The approach adopted for the design of this computer is similar to the one proposed by Koren and Silberman [9] where the algorithm is first represented as a data-flow graph and then mapped onto a network of data-driven processing elements.

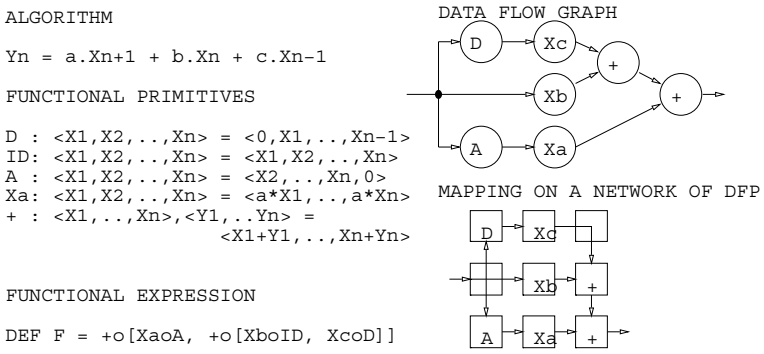


Figure 1: Functional programming and Data-flow implementation

Processing elements execute basic operations involved in a data-flow graph, while regular interconnections of these elements serve to build paths implementing the edges of this graph.

Our approach, however, is slightly different since it relies on a tight integration of the data-flow hardware computing model described in [9] and of the FP functional programming syntax introduced by Backus [3].

Our model, illustrated on a very simple example fig. 1, appears to be slightly more restrictive that the original one. It is very well suited, however, for iterative processing of huge data structures like digital images at video frame rates.

2 Data-flow functional model

An algorithm to be implemented is first represented as a directed data-flow graph (DFG). Such a DFG may be viewed formally as a directed graph whose nodes are basic operators of the algorithm and whose arcs represent data dependencies.

Execution of this DFG occurs as tokens flow along the arcs, into and out of nodes, according to a set of firing rule. These firing rule specify that a node may fire whenever tokens are present on its input arcs and tokens may be produced on its output arcs. Nodes may fire concurrently. When a node fires, input tokens are consumed, function result are computed and produced on the output arc(s).

Advantages provided by this representation have already been discussed [5][8].

First, it enables the exploitation of concurrency inherent in the algorithm without requiring the programmer to indicate it explicitly. As a result, all parallel operations may be executed concurrently.

Second, the lack of hidden dependencies between operators yields to a cleaner algorithm specification.

Third, execution pipelining may be increased by assigning a FIFO buffer to each arc. This scheme allows a node to produce one output token before the previous one has been effectively consumed.

Previous work [2] have shown that there exists a natural duality between directed acyclic data-flow graphs and functional expressions introduced by Backus in its FP programming language. Both are ways of representing an algorithm. The first is relative to an hardware execution model, the second is its natural software expression. From a more formal point of view, it can be shown that the explicit representation of data dependencies in the former corresponds to the lack of side-effect in the latter.

3 Data-Flow Computer Architecture

The core of our data-flow functional computer is a network of data-driven processing elements designed to operate on data-flows. We call *data-flow* a structured data set moving serially along a physical link ("data-stream" would be more accurate in this context). Practically this will be a sequence of numeric values and parenthesis. The goal is to achieve computations on these data-flows on the fly, by coupling each operation defined in the algorithm DFG with a physical processor.

Two types of processors were integrated within the DFFC architecture. They are dedicated respectively to low and mid-to-high level processing.

Low-level image processing, deals mainly with sensory data, in which the 2D image topology still prevails. This class of computations, requires high-throughputs but involve simple and repetitive types of calculations and a fixed number of operations per pixel (A 5×5 convolution, for example, requires 25 multiplications and 24 additions per pixel). It has to be handled by a fast specialized processor.

Mid-to-high level processing is characterized by the manipulation of symbolic extracted feature, such as contours, polygons, sets of points or heterogeneous data structures. This requires less raw computing power but involves much more algorithmic complexity. It must be handled by a more general purpose processor.

The integration of different kinds of processors reflects at the hardware level the distinction between levels of processing at the application level. It prevents application from inefficient implementations that would result if all atomic operations should have the same complexity.

3.1 Low level processing elements

For the execution of low-level functions, we developed a custom data-flow processor (DFP) [11]. Two coupled DFPs have been included in a single chip. Each DFP has 6 input-output ports and has been designed to be mesh-connected in 3D networks. The core of the data-flow processor is interfaced to the outside world through 3 input stacks and 3 output stacks. Routing of data-flows between stacks and I/O ports is done via a configuration register independent from the operator function. A 3-stages pipelined datapath, including a 8-bit multiplier and a 16-bit ALU is inserted between the input and output stacks. The control part has been designed on the basis of a a programmable state-machine, for which a 64 32-bit word program RAM has been included in the DFP. Each DFP also includes a 256 9-bit words local data memory.

According to usual data-flow principles, execution of DFP operators is controlled by a dynamic firing rule.

A wide variety of data-flow operators can be implemented using the state machine. Complex operators (convolution for example) can be defined as macro-functions using combinations of basic operators mapped on groups of processors.

DFPs may be used simultaneously as data-flow operators or as cross-bar routing elements.

3.2 Mid-to-high level processing elements

For the execution of mid-level functions, the T800 transputer was chosen because it is a general purpose processor which directly incorporates communication links. It is therefore very well suited for the implementation of complex functions into data-flow operators.

Transputer modules including 1 Mbyte of memory are the basic elements of high-level processing networks.

Any kind of mid to high-level operator may be implemented on such processing elements as C programs, provided the two following constraints are met : First is to conform to the data-flow principle, i.e. output flows must be only functions of input flows, independently of what occurs in other processors and second is to satisfy real-time execution timing constraints.

The first problem is solved by implementing the operator as a single parallel transputer process reading input tokens and writing output tokens on logical input/output "slots" within an infinite run loop. In order to be make the code of such an operator independent of its placement within the transputer network, the mapping between the logical slots of the operator and the physical links of the transputer module is done dynamically at load-time by a local boot-process.

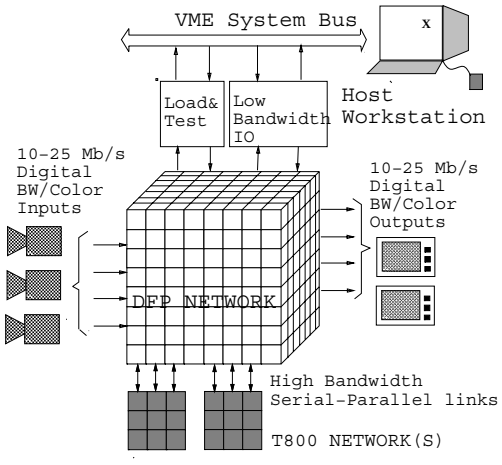
The second problem, which raises from the cooperation of heterogeneous and asynchronous processing elements under real time execution constraints, may be solved by using a dual-port buffering mechanism.

Simultaneous routing path(s) are implemented within each transputer node as parallel processes, allocated at load-time by the local boot-process.

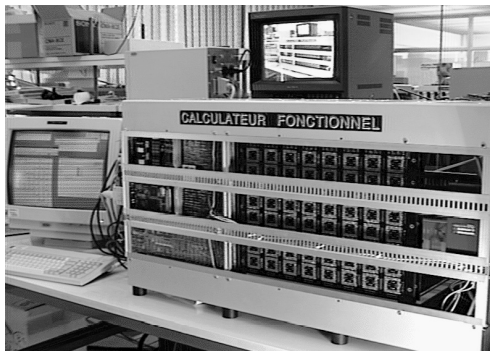
PE	USE	BANDWITH	PROGRAM	CROSSBAR
DFP 8-bit	Low-level Simple operators	6×25 Mb/s 50 Mops	State machine	Hardware configuration register
T800 32-bit	Mid level Complex operators	4×2 Mb/s 3 Mflops	C process	Concurrent routing processes

3.3 Global architecture

The computer architecture is illustrated fig.2.



(a) Hardware architecture



(b) Computer

Figure 2: The Data Flow Functional Computer

The core of this architecture is a large 3D inter-connected network of DFPs. This network is physically built of 3D-interconnected boards, each including 128 DFPs in a 8 $\times$ 8 $\times$ 2 network. Up to 8 of these boards have been "stacked" onto each other to provide a total amount of 1024 DFPs.

Transputer networks are arranged in 3 $\times$ 3 2D arrays "plugged in" the DFP network via a common backplane. DFP and Transputer modules communicate using bidirectional parallel/serial link interfaces.

Our computer also incorporates digital B/W and color video input/output subsystems, low-bandwidth data-flow interfaces and a global controller coupled with an SPARC^{tm} host workstation. Apart from running the whole software programming environment, the host workstation may act as a final high-level processing layer. This can be done via the data-flow low I/O ports of the low-bandwidth interface.

4 Functional Programming

Advantages of using a functional, side-effect-free language to describe data-flow graphs have already been detailed [6][8]. Koren and al [9], for example, used Ackerman's VAL programming language [1]. In our context, previous works have shown that the FP syntax introduced by Backus have nice properties for directed acyclic data-flow graph descriptions [2]. In order to efficiently couple this syntax both with the semantic of the image processing data flow graphs and with the processing elements capabilities, we will use a restricted dialect $< \mathcal{O}, \mathcal{P}, \mathcal{F} >$ of Backus' original FP, in which:

- $\mathcal{O}$ is the set of *objects* : There will be only two basic types of objects, referred as Atoms : Pixel (numeric value) and Control (StartOfList, noted "<" and EndOfList, noted ">").

  In order to efficiently represent data flows, several predefined composite types are provided:

  - Line : Structured sequence of Pixels ; Example : <1,2,3>
  - Frame : Structured sequence of Lines ; Example : <<1,2>,<3,4>>
  - N-tuple : Fixed-length sequence of Pixels ; Example : 1,2,3

  Line objects provide an implementation of the list concept. Frame objects will typically correspond to images (and in fact, video input and output are sequences of frames). N-tuples are provided as an implementation of fixed-sized vectors of values. Such a classification is not strict and in fact any combination of atoms can be used, but it simplifies function typing and semantic checking of programs.

- $\mathcal{P}$ will be the set of predefined functions or *primitives*. Each primitive function of the language corresponds to an operator implementable on at least one type of processor (or group of processors) of the data-flow computer.

- $\mathcal{F}$ is the set of *functional forms*. Only three functional forms are provided in order to construct new functions from existing ones:

  - The composition form defined by: $(f \circ g) : x = f : (g : x)$
  - The construction form defined by: $[f_1, ..., f_n] : x = (f_1 : x, ..., f_n : x)$

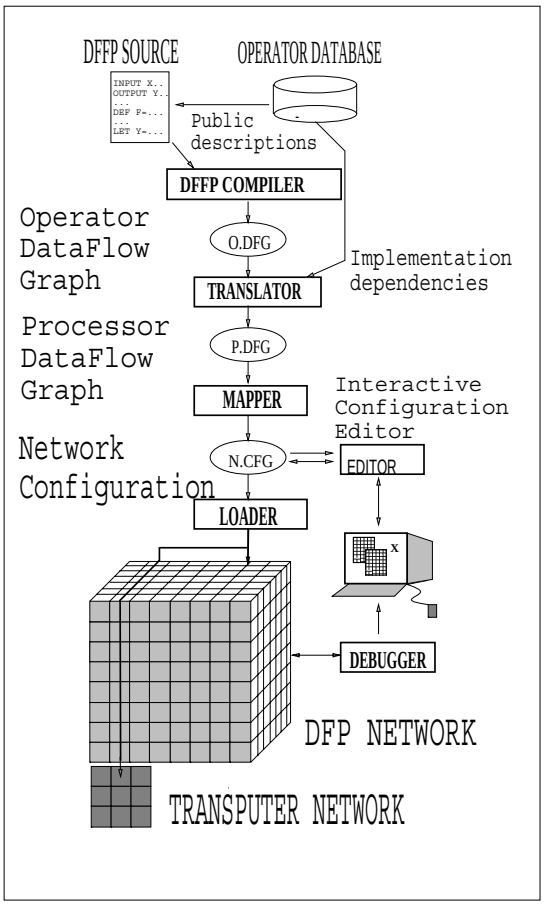


Figure 3: Programming environment overview

– The selection form defined, for any integer k by: $k \circ [f_1, ..., f_n] = f_k$

As illustrated fig.1, composition functional form corresponds to a serial placement of operators, while construction form corresponds to a parallel placement. Selection form is only provided as a means of selecting one specific result of a multi-output function.

It can be noted that we make a "pure" use of the functional programming concept since we completely remove variables from algorithm expressions. This appears to be the software counterpart of eliminating address concept at the hardware level.

5 Programming Environment

It is illustrated Fig.3.

5.1 Operator libraries

Since goal is to couple each primitive of the language with a data-flow operator, a library has to be designed for each type of processor. These libraries will mainly save end users

from writing the most commonly used primitives in the field of image processing. This is specially true for the low-level operators implemented on the DFPs, since writing state machine source code is a task quite similar to writing assembly language code on a classical computer. The current version of the low-level library, including both single processor operators and encapsulated macro-operators contains about 150 primitives. This can be viewed as a near optimal size, since the low-level operators complexity is essentially limited by the DFP hardware computing resources.

By contrast, transputer-based operators complexity is only restricted by software (as long as execution time is not a constraint). Therefore, aside from a basic mid-level library, it was necessary to allow users to write their own specific data-flow operators. For this, a C source template is provided. A common specific module, merged at compile-time, and initiating a local boot process at load-time, makes the definition of such an operator independent of the actual network configuration and of any parallel routing process. This module also provides a set of pre-defined data-flow I/O routines to communicate with the DFP processors. The result is that, at the application level, DFP-based and Transputer-based operators look exactly the same, thus greatly increasing the coherency of the application design.

Programmers view of the libraries is limited to a global operator database gathering public descriptions of primitives. Each entry in this database consists of a prototype, typing operator input and output sets, a list of its external parameters and their default value along with a brief description of the operator functionality.

The libraries may be easily extended to suit user's specific needs at any stage of application development.

This approach has the advantage of introducing a software hierarchy that maximizes code reusability and therefore greatly reduces the programming complexity. It also provides an homogeneous software interface for an heterogeneous architecture.

5.2 DFFP Compiler

An algorithm to be implemented is first expressed as a combination of primitives using functional forms in a FP-like syntax. The resulting functional expression is then converted into an equivalent operator data-flow graph by means of a DFFP compiler built upon a LR(1) parser like YACC. Here DFFP stands for Data Flow Functional Program, recalling that our dialect of FP is merely a data-flow graph description language.

Fig. 4 illustrates this stage as handled by the interactive DFFP to graph compiler.

Rightmost is the DFFP source window containing the algorithm functional description. Once compiled, the equivalent data-flow graph is visualized, thanks to smart graph drawing algorithm, based on a simulated annealing version of the STT algorithm [14]. This feature provides a useful bridge between a textual, formal description of an algorithm and its more "intuitive" graph representation.

The DFFP source code may be divided in four main parts: Global symbols declaration, global input/output specification, functional macro definitions and main instanciation part.

The global symbols, declared in the first part are global parameters whose scope extends to the whole module.

The specification part allocates and types physical input/ouput devices. VIDEO devices correspond to digital video sub-systems while ASYNC devices are associated to controller data-flow low-bandwidth ports.

The main instanciation part contains the highest functional level description of algorithm. It combines primitive functions (noted here in upper-case letters, for the sake

of clarity) or user-defined macros (lower-case) to build outputs from inputs. The DEF definition statement enables a function result to be used more than once. It can also be used to split a complex function into subsequent expressions. The LET statement binds output identifiers with function results. The 1 (2, ..) constructors are function output selectors.

Macro definitions, delimited by keywords BEGIN and END, consist of a local parameter declaration, a local input/output specification and a main-like functional body. From a classical programming language point of view, a macro is a function declaring inputs, outputs and making use of local declarations (parameters) and function symbols (DEFS). From a data-flow graph point of view, a macro is a stand-alone subgraph, which can be freely replicated. Macro definitions may be nested with an arbitrary depth. In this case, the scope of the local parameters extends recursively to the macro body and its nested components. A include statement provides a means of sharing macro definitions among source files. It might even be possible to pre-compile useful macro definitions in order to insert them in the primitives library (In this case the programmer will have to write its data-base external specification).

5.3 Translation

A translator handles the conversion of the operator data-flow graph output by the compiler into a loadable processor graph, that is the conversion of public instances of the primitives into their actual implementation on the processor(s). This may involve, for example, translation from external parameters to processor registers and/or macro-operators expansion.

5.4 Mapping

The resulting processor graph, where each node corresponds to an operator implementable on a physical processor has then to be mapped onto the network. This mapping stage consists of three steps: placement, path construction and path balancing.

Placement assigns each basic operator to a physical processor of the network.

Path construction determines how the arcs of the operator DFG are represented by paths within the processor network.

Path balancing may involve insertion of additional FIFO buffers in order to compensate for token delays induced by different path lengths or by explicitly delayed operators.

The problem is complicated by the fact that these three steps interact with each other. It is simplified by the fact that each processor may act simultaneously as an operating, routing or buffering element.

It can be handled by an automatic place and route algorithm, largely inspired by previous works on graph mapping [4], specially on hex-connected arrays of asynchronous cells [10][12]. A first version of the algorithm, dedicated to application prototyping, allows a fast sub-optimal mapping. A optimized version, still under development will seek to minimize the graph hosting volume by using pre-mapped macro cells.

5.5 Loading

The network configuration file generated by the mapper is finally downloaded from the host into the computer through the global network controller. Programming of the DFPs id performed via a global bidirectional built-in scanpath. The transputers bootable code

is conveyed through the DFP network via the DFP network data paths. The complete loading time for a 1024 DFP + 12 T800 configuration is about 30 seconds.

Once loaded the application may be executed in real-time operating on video flows coming from CCD cameras and sent to video monitors.

Low-bandwidth data-flow interfaces may also be used to run the algorithm in a step-by-step manner, data being read from (written to) the host. This, along with the possibility to access any processor register from the network controller allows a fine-grain, data-flow oriented debugging of algorithm.

All of the programming tools are integrated within an interactive, user-friendly graphic interface running under the X window system.

6 Image Processing Applications

Several significant real-time image processing algorithms have been formulated using our data-flow functional programming model and successfully implemented. These range from low-level processings like histogram equalization or edge detection to mid-level algorithms based on geometrical or color features extraction.

Following is an overview of a real-time vision algorithm which is a good example of cooperation between low and mid-level operators.

The goal of this algorithm is the extraction of main edges along predominant directions. It is based upon the computation of the histogram of gradient direction and a partial Binary Hough Transform.

The functional description of the algorithm is given fig. 4.

From the system point of view, several noticeable points should be pointed out:

First is the use of optional parameters to parametrize primitives or user-defined macros.

Second, is the structured design provided by the *dirSegs* macro definition. From this remark, it is clear that extending the algorithm in order to take into account more directions would be straightforward.

Last is the use of *polymorphic* operators. The result of such operators may depends on input data types or effective parameter value. The goal here is to facilitate programmers access to the library by reducing the number of primitive symbols they have to deal with. For transputer-based primitive, this polymorphism is handled at load-time, by local instanciation of external parameters. For DFP-based operators, it is handled by the operator state machine itself, thus providing a true run-time polymorphism.

From an algorithmical point of view one interesting feature of this implementation is the use of a programmable gray level ramp both to compute the projection and to locate the position of the detected lines. This method introduced in [13] has proven to be very well suited for highly pipelined architecture like the DFFC. Here a 10-bit-wide θ-oriented, screen centered, gray level ramp R_θ is generated using a single macro involving one transputer and a few DFPs.

Another interesting feature is the buffering of the ASYNC input implemented by the \$ operator. This buffering, which uses a dual port memory to store the last threshold value, prevents the THR thresholding operator from blocking and therefore allows the user to interact with the running algorithm in a completely asynchronous manner.

This algorithm has been executed in real time on 25 Hz sequences of 800×572 pixel images (16 MHz pixel frequency). Results of execution on a indoor scene are illustrated Fig. 5.

This application can be easily extented in order to take into account more directions. As described here, it uses nearly 200 DFPs and 3 transputers, involves about 2 billions of 8- or 16-bits operations per second, and has been achieved with 1/4th of the complete configuration of our computer.

7 Performance evaluation

By nature, this Data Flow Functional Computer operates at a fixed speed (in this case at digital video speed, up to a 25 Mhz pixel frequency). Therefore, its performance cannot be evaluated by the time it takes to perform a given task. The performance measure should rather be the number of processors (or the fraction of the machine) needed to perform a given task.

More precisely, the performance index is the number of operations per pixel performed at digital video speed. Each DFP being able to perform 1 or 2 elementary operations per pixel, the maximum theoretical power of the computer is about 2000 operations per pixel. Depending upon the regularity of the algorithms and the degree of mapping optimization, the practically available power ranges from 200 to 800 operations per pixel. The algorithm illustrated in the previous section involves about 120 operations per pixel. Up to now, the most complex algorithm implemented is a low-level filter computing pixel rank order within a 16×16 window and involving 766 operations per pixel.

This computer is not limited by I/O problems. Using the implemented I/O channels it can input, process on the fly and output simultaneously two black and white and two color digital video streams.

8 Conclusion

This paper has described the concept and design of a data-flow functional computer (DFFC) developed at ETCA and its application to real-time image processing. The very last version of this computer embeds 1024 DFPs in a $8 \times 8 \times 16$ three-dimensional network and 12 T800 based transputer modules. It will be soon extended with four 3×3 two-dimensional transputer networks.

Hardware and software design were based on a close integration of the data-flow computing model and the functional programming concept. This lead to a simple highly regular hardware and a homogeneous, user-friendly software programming environment. It is hoped that this should allow image processing specialists, who may not be aware of machine architecture, to design, implement and test a wide range of image processing applications.

Several significant low to mid-level image processing algorithms were implemented and executed at video frame rates.

Investigations are now carried on to implement larger image processing applications based upon more complex pattern recognition features.

Apart from being a research tool for real-time image processing, the DFFC may also be used as a design platform for autonomous vision automata. In this concern, work is under development to extract an on-board autonomous sub-system from a DFFC configuration. This process may be handled at the system level, by packing a mapped network of processors into a single machine, or at the circuit level, using waferscale integration techniques. Wafer stacking technologies involving three-dimensional routing should also be worth considering.

The data-flow functional computer is mainly dedicated to image processing. Its concept, however, is much more general and can be maintained within a system that includes processors of different granularity. Basic processors may be adapted to address many other problems. For example, a 64-bit floating point data-flow processor could similarly be developed and used for the design of a data-flow functional supercomputer dedicated to the resolution of partial derivative equations by finite difference methods.

References

[1] W.B. Ackerman and J.B. Dennis. *VAl-A Value-Oriented Algorithmic Language: Preliminary Reference Manual.* MIT/LCS/TR-218, Jun 1979.

[2] E. Allart and B. Zavidovique. Functionnal image processing through implementation of regular data flow graphs. In *21st Annual Asilomar Conf. on Signals,Systems on Computers, Pacific Grove,Ca*, Nov 1987.

[3] J. Backus. Can programming be liberated from von neumann style ? a functional style and its algebra of programs. *Communications of the ACM*, 21(8), Aug 1978.

[4] S. Bokhary. On the mapping problem. *IEEE Transactions on Computers*, Mar 1981.

[5] A.L. Davis and R.M Keller. Data flow program graphs. *Computer*, 16, Feb 1982.

[6] J.B. Dennis. Data flow supercomputers. *Computer*, 13, Nov 1980.

[7] C.C. Weems et al. Parallel processing in the darpa strategic computing vision program. *IEEE Expert*, 6(5), Oct 1991.

[8] K.M. Kavi, B.P. Buckles, and U. Narayan Bhat. A formal definition of data flow graph models. *IEEE Transactions on Computers*, C-35(1), Nov 1986.

[9] I. Koren, B. Mendelson, I. Peled, and G.B. Silberman. A data-diven vlsi array for arbitrary algorithms. *IEEE Computer*, Oct 1988.

[10] B. Mendelson and G.B. Silberman. Mapping dataflow programs on a vlsi array of processors. In *Processinof 1987 Intl Conf on Computer Architecture*, Jun 1987.

[11] G.M. Quenot and B. Zavidovique. A data-flow processor for real-time low-level image processing. In *IEEE Custom Integrated Circuits Conference, San-Diego, CA*, may 1991.

[12] B. Robic, P. Kolbezen, and J. Silc. Area optimization of dataflow-graph mappings. *Parallel Computing*, 18, 1992.

[13] J.L. Sanz and I.H. Dinstein. Projection-based geometrical feature extraction for computer vision: Algorithms in pipeline architectures. *IEEE Trans Pattern Analysis Machine Intelligence*, PAMI-9(1), Jan 1987.

[14] K. Sugiyama, S. Tagawa, and M. Toda. Methods for visual understanding of hierarchical systems structures. *IEEE Transactions on Systems, Man and Cybernetics*, SMC-11, Feb 1981.

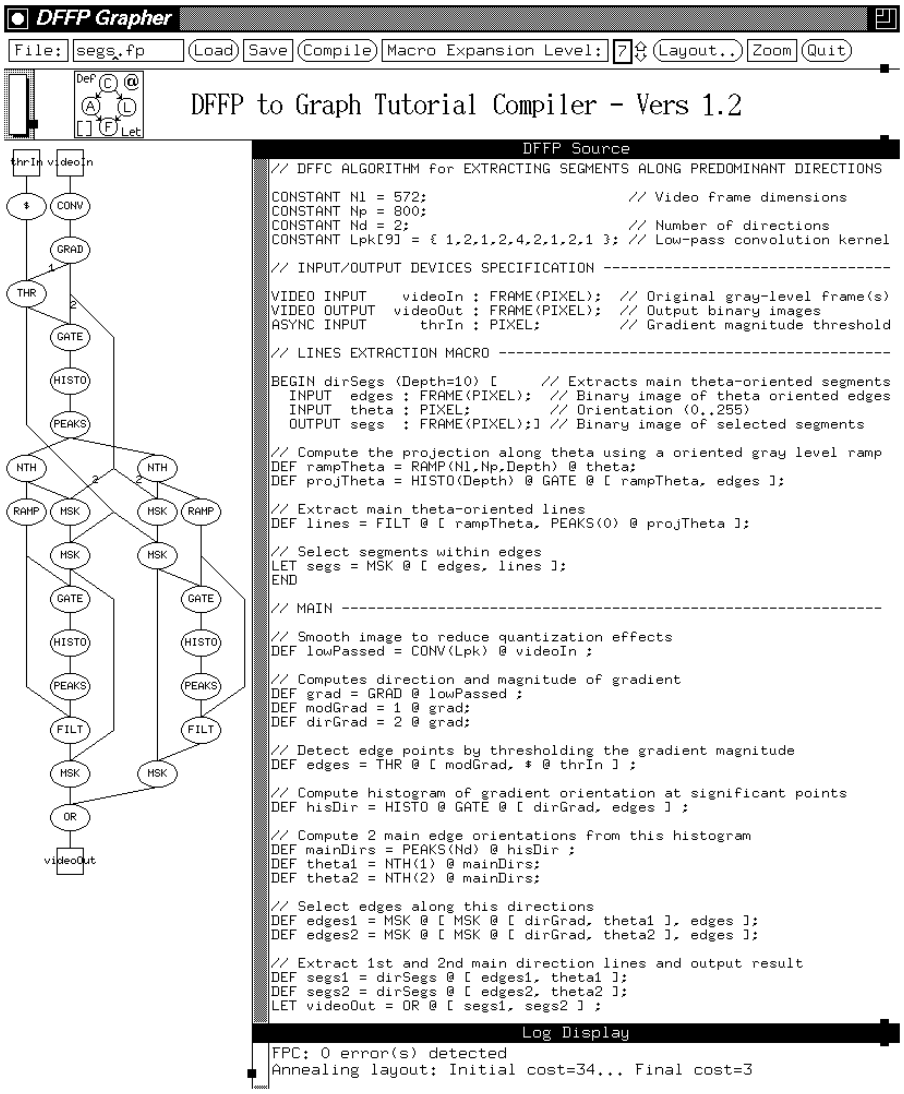
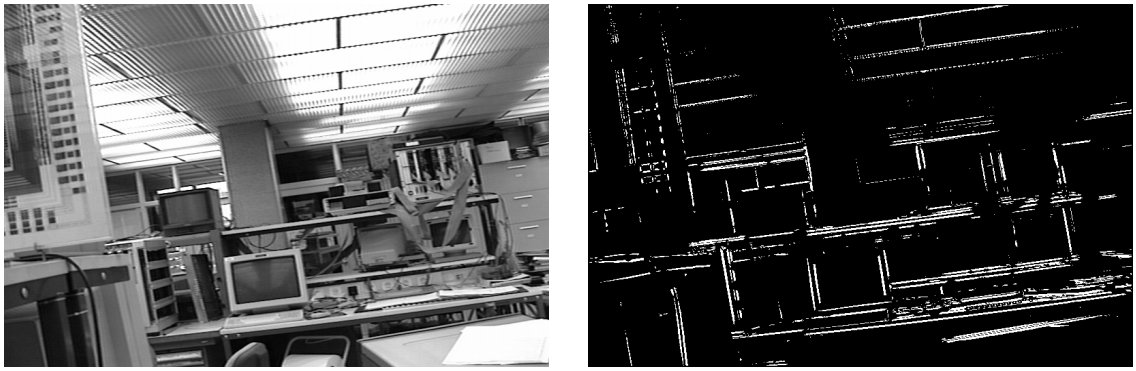


Figure 4: DFFC Compiler sample session



(a) Original 800x572 image (b) Extraction of main segments

Figure 5: Real-time extraction of edges along predominant directions