

Integrating Transputer Arrays within a Data-Flow Architecture: Applications in Real-time Image Processing

Olivier ECKLÉ, Jocelyn SÉROT, Georges QUÉNOT

Laboratoire Système de Perception

DGA/Etablissement Technique Central de l'Armement,

16 bis Av. Prieur de la Côte d'Or, 94114 Arcueil CEDEX (FRANCE)

Tel: (33 1) 42 31 93 48 Fax: (33 1) 42 31 99 64 Email: eckle@etca.fr

Abstract.

This paper presents a Data Flow Functional Computer (DFFC) developed at ETCA and dedicated to real-time image processing. One original feature of this computer lies in the integration both at the hardware and software level of two types of data-driven processing elements: 1024 custom Data Flow Processor (DFP) – embedded in a 3D interconnected network and dedicated to low level processing and 36 T800 Transputers – embedded in a 2D interconnected networks and dedicated to mid to high level processing. A unifying programming model is provided, based on a close integration of the data-flow architecture principles and the functional programming concepts.

An image processing algorithm, expressed using an FP-like functional syntax is first converted into a Data-flow Graph (DFG). The nodes of this graph are real time operators implementable on the physical processors of the data-flow machine. This DFG is then physically mapped onto the network of processors.

The programming environment includes a complete compilation stream from FP-specification to hardware implementation, along with a global operator database. An original programming technique has been developed for the transputers to ensure a full compatibility with data-flow model.

Several image processing algorithms were implemented on this system and run in real time at digital video speed.

1. Introduction

Real time image processing involving a computing power in the range of billions of operations/s still benefits from specialized architectures. The design of such architectures can be based on the two following points:

First, most of the computationally demanding tasks involved in the processing exhibit an inherent parallelism and this parallelism can be exploited by an architecture providing a high degree of pipelining.

Second, these tasks may be classified according to low, intermediate, and high level computational tasks, and distinct processors are required to handle each of these levels under real-time constraints [1].

These assumptions served as the basis for the design of our Data-Flow Functional Computer.

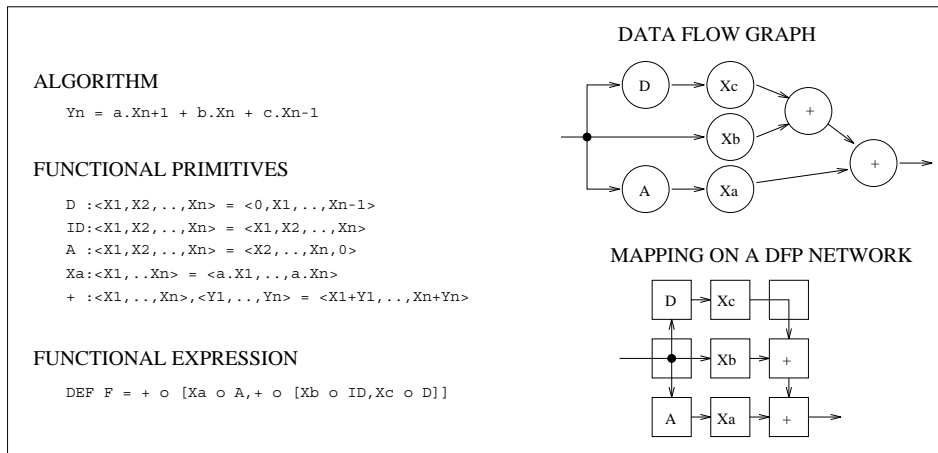


Figure 1: Functional programming and Data-flow implementation

The approach that we adopted [2] is similar to the one proposed by Koren [3] where the algorithm is first represented as a data-flow graph and then mapped onto a network of data-driven processing elements. Processing elements execute basic operations embedded along 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 [3] and of the FP functional programming syntax introduced by Backus [4].

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

It allowed us to design and build a massively parallel computer mixing fine grain and coarse grain processors within a highly regular architecture, along with an homogeneous and easy-to-use software programming environment. Several real-time image processing algorithms were reformulated according to this programming model and run successfully at video frame rates.

In the next section, the basic principles of the data-flow functional computing model are briefly outlined. Section 3 describes the processing elements and the global hardware architecture of the computer. Section 4 is a presentation of a FP dialect used to program the DFFC. The software programming environment is presented in section 5. Section 6 describes the transputer arrays integration and section 8 concludes this paper.

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: Nodes are basic operators of the algorithm and 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 rules. These firing rules specify that a node becomes active whenever tokens are present on its input arcs and tokens may be produced on its output arcs. When this occurs, input tokens are consumed, function result(s) are computed and produced on the output arc(s).

Advantages provided by this representation have already been discussed [5][6].

First, it enables exploiting the inherent concurrency in the algorithm without requiring the programmer to make it explicit. 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 actually consumed.

Previous work [2] has shown that a natural duality exists 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 one refers to a 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.

In order to efficiently couple the FP syntax with the semantics of the DFG we use a FP subset. This language subset, detailed in section 4, may be viewed as an efficient formalism for describing and manipulating directed acyclic data-flow graphs.

3. The E.T.C.A. Data-Flow Computer Architecture

The core of the E.T.C.A. 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 brackets. The goal is to achieve on the fly computations on these data-flows 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-throughput, but involves 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 more symbolic extracted features, such as edges, 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, like a transputer.

It is clear that this integration of processors with distinct granularity reflects, at the hardware level, the distinction between levels of processing at the application level. It may be generalized to any kind of processing elements as long as all these processing elements share the same data-flow communication protocol.

The main goal is to prevent operators from being inefficiently implemented, as this occurs for instance, if all atomic operations must have the same complexity. Such an effect will also be limited, practically, since several simple operators can be associated to a single processor and complex operators may be encapsulated as macro-operators distributed among small groups of processors.

3.1. Low level processing elements

For the execution of low-level functions, we developed a custom data-flow processor (DFP) [7]. 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. Each stack is a 8 9-bit word, 25 MBytes/second bandwidth, synchronous FIFO, and acts as a built-in token balancing buffer. Each I/O port contains a receiving part that can be connected to any input stack, and a sending part that can be connected to any output stack. The actual configuration of each I/O port is completely independent from the operator function.

A 3-stage pipelined datapath is inserted between the input and output stacks. The first stage decodes the input data type and generates commands for the following stages. 8-bit operations (input shifts and multiplications) are performed during the second stage, and 16-bit operations (general purpose ALU, absolute value, minimum or maximum, output shift) are performed at the third stage. This datapath is able to perform up to 50 million 8- or 16-bit operations per second.

According to usual data-flow principles, execution of DFP operators is controlled by a dynamic firing rule. The control part has been designed on the basis of a programmable state-machine.

A wide variety of data-flow operators can be implemented using the state machine. Among them are arithmetic and logical operators, comparators, counters, line and pixel shifts, derivatives, summations and histograms. More 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. In fact each processing node can be seen as a cross-bar including a processing unit or vice versa.

3.2. Mid-to-high level processing elements

Two reasons account for the integration of a general purpose processor within computer architecture: First, is ensuring an efficient implementation of complex operators without using a prohibitive number of DFPs; second, is allowing higher processing levels.

The choice of the InMos T800 Transputer was motivated by the following features:

- it directly incorporates communications links and can be easily integrated within a data-driven architecture.
- data-flow operators can be easily programmed as C processes.
- it can execute at the frame rate complex manipulations of symbolic extracted features (such as histograms, projections, ...).
- a reasonable cost and good availability when the objectives were defined.

Transputer modules including a T800 CPU + FPU, 4 serial communication links and 1 Mbytes of external RAM are the basic elements of the 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. This will be detailed in section 6.1.

The second problem raises from the cooperation of heterogeneous and asynchronous processing elements under real time execution constraints. It will be traited in section 6.2.

3.3. Global architecture

The computer architecture is illustrated Figure 2. The core of this architecture is a large 3D inter-connected network of DFPs [8]. This network is physically made of 3D-interconnected boards, each including 128 DFPs in a $8 \times 8 \times 2$ network (Figure 3). 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 3×3 2D arrays "plugged in" the DFP network via a common backplane. DFP and Transputer modules communicate using bidirectional parallel/serial link interfaces.

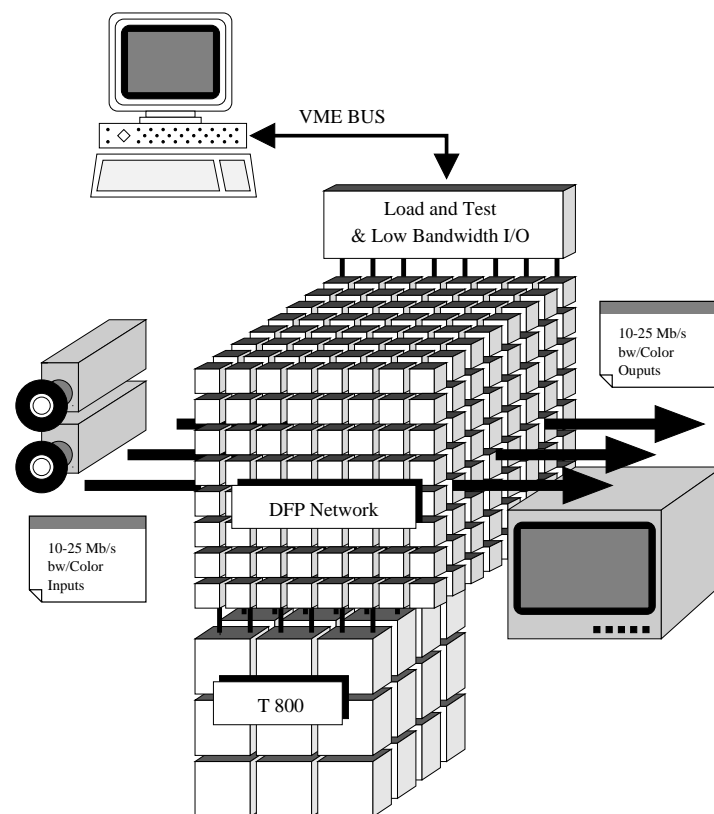


Figure 2: Data-flow Functional Computer Architecture

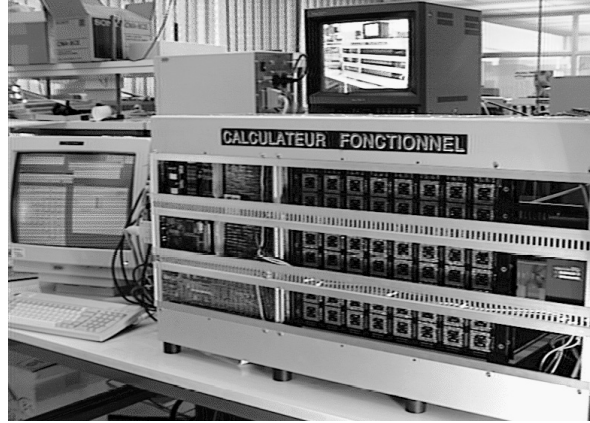


Figure 3: The Data Flow Functional Computer

Our computer also incorporates digital B/W and color video input/output subsystems, low-bandwidth data-flow interfaces and a global controller coupled with a SPARCtm 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. It is quite difficult to evaluate the performances of the Data-Flow Functional Computer by comparing them with those of other similar machines, because:

- As far as we can know, the Data-Flow Functional Computer is the only one in its category. The other projects of Data-Flow machines which exist in the world are so different that compare its with our Data-Flow Functional Computer has no real sense.
- The most significant performances measure of the Data-Flow Functional Computer is the number of operation by pixel, and not the usual MIPS, because, in ours case, the instruction execution rate is completely controlled by the video I/O data frequency.

4. Functional Programming

Advantages of using a functional, side-effect-free language to describe data-flow graphs have already been detailed [9][6]. Koren and al [3], for example, used Ackerman's VAL programming language [10]. In our context, previous works have shown that the FP syntax introduced by Backus has 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 $\langle \mathcal{O}, \mathcal{P}, \mathcal{F} \rangle$ of Backus' original FP, in which:

- \mathcal{O} is the set of *objects* : There will be only two basic types of objects, referred to as Atoms : Pixel (numeric value) and Control (Start_Of_List, noted "<" and End_Of_List, 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 : $\langle\langle 1,2\rangle,\langle 3,4\rangle\rangle$
- 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-size 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 program checking.

- \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: $[f_1, \dots, f_n] : x = (f_1 : x, \dots, f_n : x)$
 - The selection form defined, for any integer k by: $k \circ [f_1, \dots, f_n] = f_k$

As illustrated Figure 1, the composition functional form corresponds to a serial placement of operators, while the construction form corresponds to a parallel placement. The selection form is only provided as a means of selecting one specific result from 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 addresses at the hardware level. Moreover, it reflects the strong relationship between functional programming and data-driven architectures.

5. Programming Environment

Figure 4 gives an overview of the programming environment.

5.1. Operator libraries

Since our goal 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 application programmers from writing the most commonly used primitives in the field of image processing [11].

This is specially true for the low-level operators implemented on the DFPs, since writing state machine source code is 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 operator complexity is only restricted by software (as long as the 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 (see section 6.1). This module also provides a set of pre-defined data-flow I/O routines to communicate with the DFP processors. At the application level, this makes DFP-based and Transputer-based operators look exactly the same, thus greatly increasing the coherency of the application design.

Appendix 1 gives the implementation of an operator for the extraction of line window coordinates.

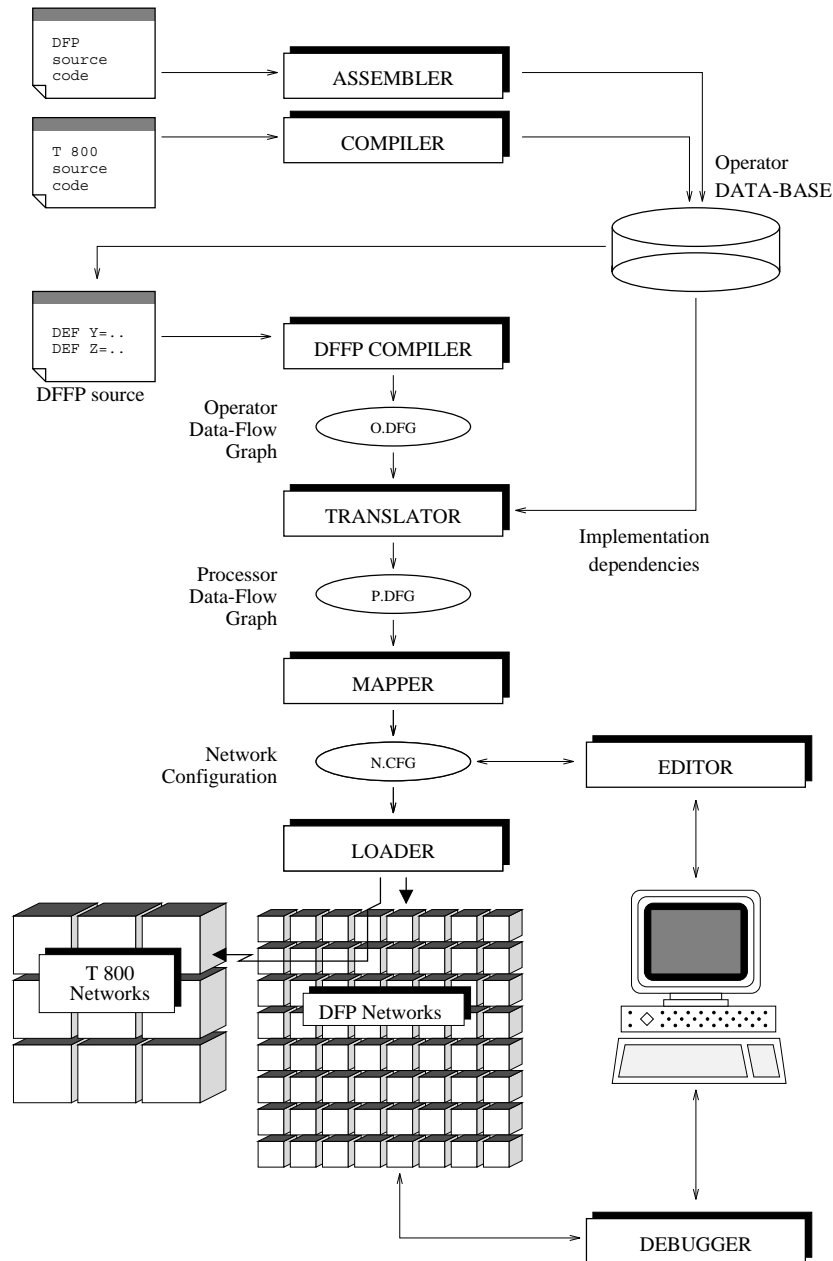


Figure 4: Programming environment overview

Apart from the algorithmic simplicity of this —rather tutorial— example, one noteworthy point is the use of “external” parameter(s) to adjust the operator functionality. The default values of such parameters, supplied in the operator source code, may be overridden at load-time by user-specific values, provided at the application level (see section 6.1). This feature, handled by the local boot-module, allows the design of very flexible data-flow operators and save application specific code recompilation.

Programmers’ view of 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 helps introducing a software hierarchy to maximize code reusability. Therefore it 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 compiler.

Figure 5 illustrates this stage as handled by an interactive version of this DFFP compiler. Rightmost is the source window containing the algorithm functional description. Once compiled, the equivalent data-flow graph is visualized in the left most window. This feature provides a useful bridge between a textual, formal description of an algorithm and its more “intuitive” graph representation.

5.3. *Translation*

A translator handles the conversion of the operator data-flow graph generated by the compiler into a loadable processor graph, that is the conversion of public instances of primitives into their actual implementation on the processor(s). This involves:

- choice of the target processor (DFP or Transputer).
- translation from external parameters to processor registers and/or macro-operators expansion.
- macro operators expansion.

This phase is handled by a pattern matching mechanism based on a translation table.

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(s). This mapping stage consists of three steps: placement, path construction and path balancing.

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

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

DFFP Grapher

File: segs.fp Load Save Compile Macro Expansion Level: 7 Layout.. Zoom Quit

DFFP to Graph Tutorial Compiler - Vers 1.2

```

graph TD
    thrIn[thrIn] --> conv[CONV]
    videoIn[videoIn] --> conv
    conv --> grad[GRAD]
    grad --> thr[THR]
    thr --> gate[GATE]
    gate --> histo[HISTO]
    histo --> peaks[PEAKS]
    peaks --> nth1[NTH]
    peaks --> nth2[NTH]
    nth1 --> ramp1[RAMP]
    nth2 --> ramp2[RAMP]
    ramp1 --> msk1[MSK]
    ramp2 --> msk2[MSK]
    msk1 --> msk3[MSK]
    msk2 --> msk4[MSK]
    msk3 --> gate2[GATE]
    msk4 --> gate3[GATE]
    gate2 --> histo2[HISTO]
    gate3 --> histo3[HISTO]
    histo2 --> peaks2[PEAKS]
    histo3 --> peaks3[PEAKS]
    peaks2 --> filt1[FILT]
    peaks3 --> filt2[FILT]
    filt1 --> msk5[MSK]
    filt2 --> msk6[MSK]
    msk5 --> or[OR]
    msk6 --> or
    or --> videoOut[videoOut]
  
```

DFFP Source

```

// DFFC ALGORITHM for EXTRACTING SEGMENTS ALONG PREDOMINANT DIRECTIONS
CONSTANT Nl = 572; // Video frame dimensions
CONSTANT Np = 800; // Number of directions
CONSTANT Nd = 2; // Low-pass convolution kernel
CONSTANT Lpk[9] = { 1,2,1,2,4,2,1,2,1 }; // INPUT/OUTPUT DEVICES SPECIFICATION -----
VIDEO INPUT videoIn : FRAME(PIXEL); // Original gray-level frame(s)
VIDEO OUTPUT videoOut : FRAME(PIXEL); // Output binary images
ASYNC INPUT thrIn : PIXEL; // Gradient magnitude threshold
// LINES EXTRACTION MACRO -----
BEGIN dirSegs (Depth=10) [ // Extracts main theta-oriented segments
  INPUT edges : FRAME(PIXEL); // Binary image of the theta oriented edges
  INPUT theta : PIXEL; // Orientation (0..255)
  OUTPUT segs : FRAME(PIXEL);] // Binary image of selected segments
// Compute the projection along theta using a oriented gray level ramp
DEF rampTheta = RAMP(Nl,Np,Depth) @ theta;
DEF projTheta = HISTO(Depth) @ GATE @ [ rampTheta, edges ];
// Extract main theta-oriented lines
DEF lines = FILT @ [ rampTheta, PEAKS(0) @ projTheta ];
// Select segments within edges
LET segs = MSK @ [ edges, lines ];
END
// MAIN -----
// Smooth image to reduce quantization effects
DEF lowPassed = CONV(Lpk) @ videoIn ;
// Computes direction and magnitude of gradient
DEF grad = GRAD @ lowPassed ;
DEF modGrad = 1 @ grad;
DEF dirGrad = 2 @ grad;
// Detect edge points by thresholding the gradient magnitude
DEF edges = THR @ [ modGrad, # @ thrIn ];
// Compute histogram of gradient orientation at significant points
DEF hisDir = HISTO @ GATE @ [ dirGrad, edges ];
// Compute 2 main edge orientations from this histogram
DEF mainDirs = PEAKS(Nd) @ hisDir ;
DEF theta1 = NTH(1) @ mainDirs;
DEF theta2 = NTH(2) @ mainDirs;
// Select edges along this directions
DEF edges1 = MSK @ [ MSK @ [ dirGrad, theta1 ], edges ];
DEF edges2 = MSK @ [ MSK @ [ dirGrad, theta2 ], edges ];
// Extract 1st and 2nd main direction lines and output result
DEF segs1 = dirSegs @ [ edges1, theta1 ];
DEF segs2 = dirSegs @ [ edges2, theta2 ];
LET videoOut = OR @ [ segs1, segs2 ];
  
```

Log Display

```

FPC: 0 error(s) detected
Annealing layout: Initial cost=34... Final cost=3
  
```

Figure 5: DFFP Compiler sample session

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 gets complicated as these three steps interact with each other. It is simplified in so far as each processor may act simultaneously as an operating, routing or buffering element.

Mapping may also be performed manually thanks to an interactive graphic tool illustrated Figure 6.

5.5. Loading

Network configuration files generated by the mapper are finally downloaded from the host into the computer through the global network controller. DFP's are programmed via a global bidirectional built-in scanpath. Transputer bootable code is conveyed through the DFP network via the DFP network data paths. Complete loading for a 1024 DFP + 12 T800 configuration takes about 15 seconds.

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

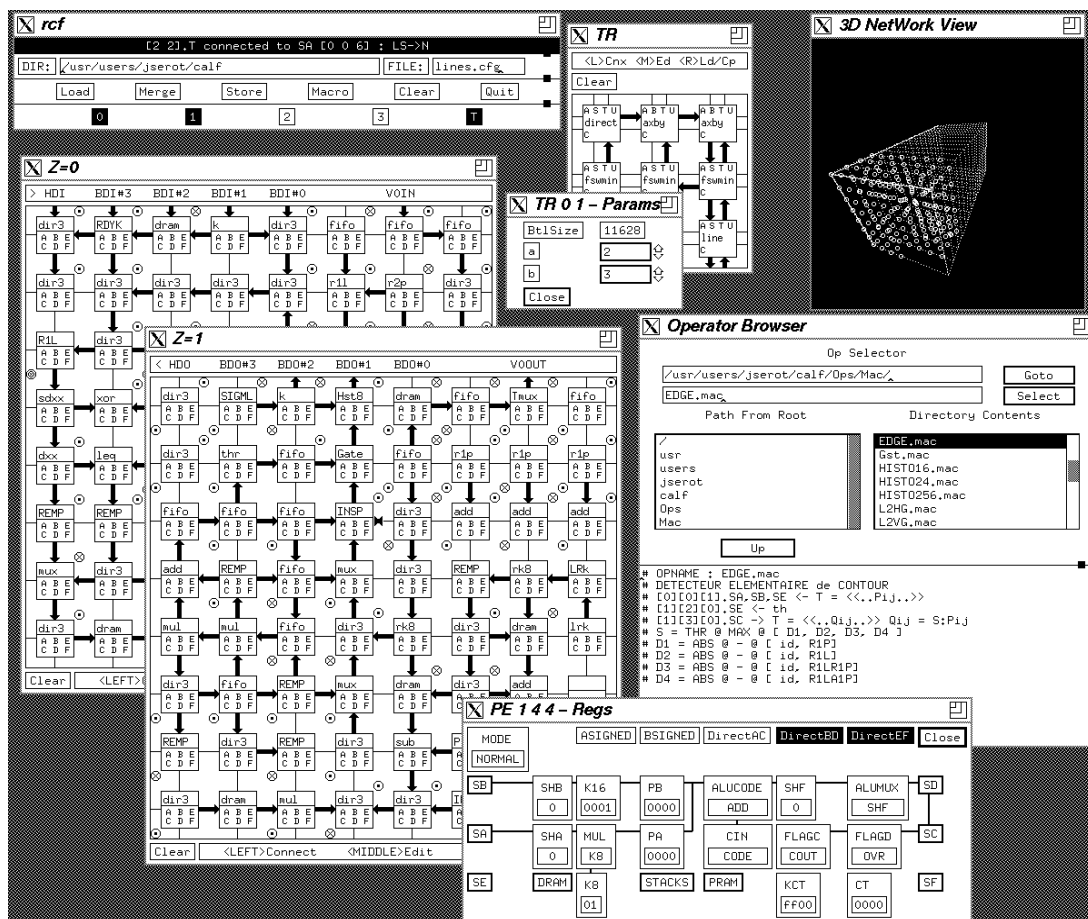


Figure 6: Interactive graphic tool overview

Using low-bandwidth data-flow interfaces algorithms can be run step-by-step, data being read from (written to) the host. This, along with the possibility to access any processor register from the network controller (via a graphical tool very similar to the one shown Figure 6), allows a fine-grain, data-flow oriented debug.

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

6. Transputer arrays integration

6.1. Loading transputer programs and configurations

The classical method for program development on transputers networks relies upon the so-called "network configuration file". This feature is clearly not compatible with our data-flow programming paradigm since it tightly binds together the executable code of each operating process and its input/output link configuration. By contrast, our goal was:

- to develop a library of flexible transputer-based operators.
- to homogenize the programming of DFP-based and transputer-based networks.
- to dynamically adjust the functionality of operators by means of a set of user-specific parameters.
- incidentally, to isolate our programming environment from the INMOS specific tools.

For this we had to:

- allow each transputer of the network to be programmed via those which are connected to the DFP network.
- allow the input/output links of an operator process to be configured at load time
- be able to adjust parameter values at load-time, without having to re-compile the process source code.
- allow execution of parallel routing processes (as DFP-based operators).

This led us to develop an original programming protocol for transputer-based networks. Our solution relies upon the definition of a generic initialization process automatically executed by each transputer at the boot-time. This process dynamically handles: first, the allocation and initialization of the operating process and of any parallel routing processes; second, the mapping of the operator logicals channels to the transputer physical links; finally, the initialization of the operator local parameters. The code of this initialization process is transparently linked with the code of each user-defined operator.

Each transputer network is logically divided into n columns of m transputers. The executable code associated with each column is conveyed from the host workstation (where it has been compiled) to the first transputer in the column through the DFP network (Figure 7).

Within each column, transputers are initially configured in "Boot from link" mode. The first transputer in the column – connected to the DFP network via a parallel/serial

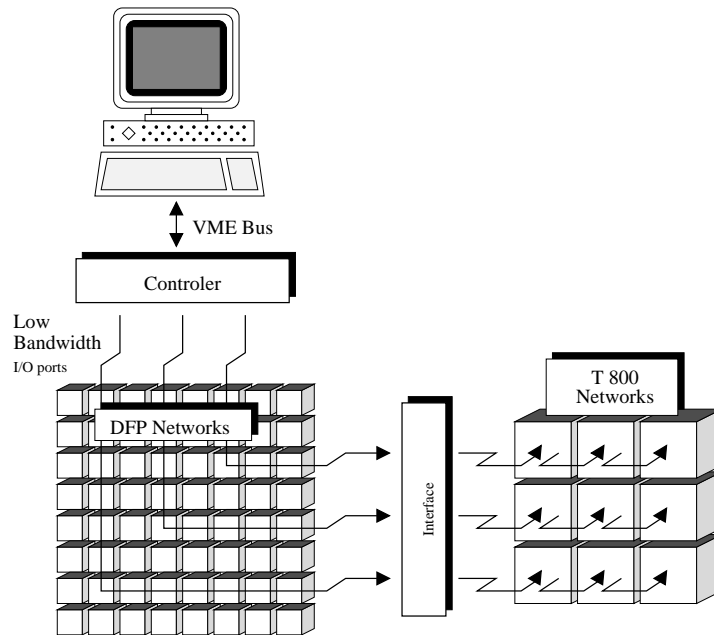


Figure 7: Transputer bootable code dispatching

link interface – dispatch the bootable code for the whole column. The received block of data contains four fields (Figure 8).

- first, the executable code of the local operating (and routing if any) process(es).
- second, two vectors giving the input/output configuration, i.e. the mapping between the logical channels of the operator and the physical links of the transputer.
- then, the values of the operator parameters (these values may be the default ones, set at compile-time, or may have been overridden by the application programmer during the compilation-translation scheme).
- finally, an integer giving the total number of bytes that must be transmitted to the next transputer in the column.

This programming technique may of course be generalized to handle arrays of any size. Moreover, it can be adapted to suit to any network topology, especially those in which users must have a full dynamic control over process allocation and link configuration.

6.2. Handling of real-time execution timing constraints

As stated in section 3.2, the cooperation of heterogeneous and asynchronous processing elements under real time constraints raises specific problems. This is illustrated Figure 9a, where:

- L_1 and L_2 are low-level operators implemented on DFPs.
- H is a high-level operator, implemented on a T800 node.
- L_1 input is a video frame sequence.

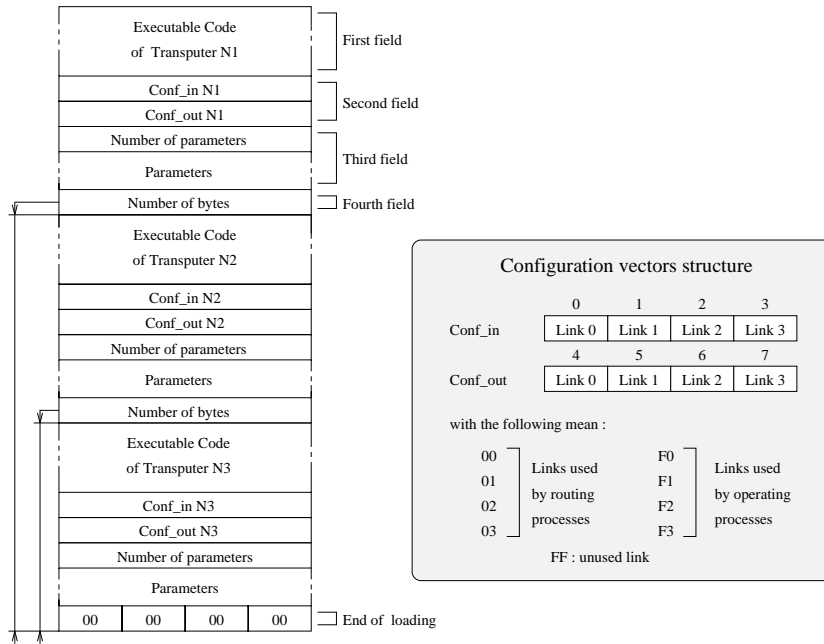


Figure 8: Data structure loaded on the transputers

- L_1 output is a line sequence (histograms of input frames, for example).
- H dataflow operator maps a line into a single pixel value (an histogram into a binarization threshold, for instance).

According to the functional syntax detailed in section 4, the program may be written:

$$\text{DEF program} = L_2 \circ [H, ID] \circ L_1 \text{ where ID is identity operator}$$

This "static" functional description, however, does not take into account timing constraints. In particular, if H is "slow" compared with L_1 , tokens accumulate on arc i . Since arc buffering capacity is limited, a dead-lock may occur.

The first way to solve this problem is to insert a FIFO buffer between L_1 and L_2 . This straightforward solution has two main drawbacks: First, the FIFO capacity depends on the delay introduced by the H operator, and this delay may be data-dependent, even unknown *a priori*. Second, in case of a fairly slow operator the required capacity may be prohibitive.

This led us to introduce a specific coupling mechanism, based on dual port buffering, for coupling asynchronous data-flow operators. This mechanism, that we called S/M coupling allows a DFP operator running at pixel-frequency to have access to the last updated output(s) of a slower high-level T800 node, instead of waiting for the next one(s), thus preventing dead-locks. It is illustrated Figure 9b, in which:

- H is the high level operator to be coupled.
- S is an upstream synchronization operator. It unconditionally accepts a sequence of objects X_1, X_2, \dots for input. When receiving a RDY control token it extracts the next X_i object from the input sequence and sends it to H . This operator acts as a synchronized input buffer for the following H operator.

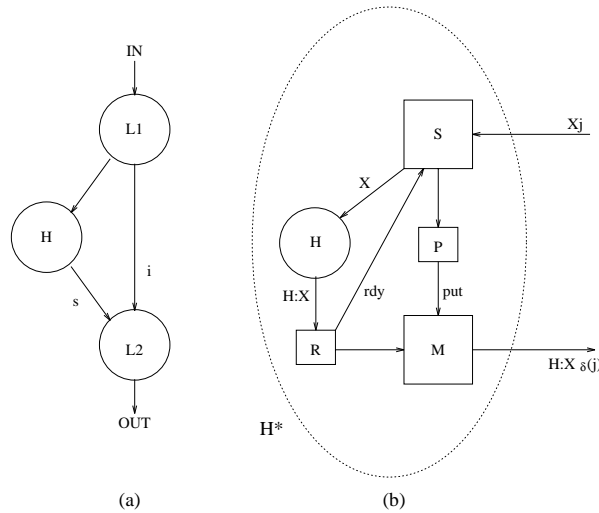


Figure 9: S/M coupling

- M is a downstream synchronization operator. It memorizes the last output of H and outputs it when receiving a PUT control token.
- R and P are auxiliary operators generating respectively the RDY and PUT control tokens from S input and H output (the PUT token is first self generated in order to start the process).

All of these synchronizing operators are easily implemented on DFPs. None of them is a "pure" data-flow operator, but the *encapsulation* $H\star$ may now be viewed as a *non-deterministic* data-flow operator:

$$H^*(X_j) = H(X_{\delta(j)})$$

where $\delta(j) = IntegralValueOf(j \div \tau) \times \tau$ and τ features the delay introduced by the operator.

This outward non-determinism simply shows that a real-time data-flow driven machine cannot be described in a purely static formalism. It should be pointed out, however, that would the operator delay be estimated, the τ constant can be specified for the operator to show a strictly deterministic behavior.

This coupling mechanism can be generalized both at the hardware and software level to provide access to any combination of previous outputs (instead of the very last one). It can be handled automatically at compile time in order to be fully transparent to programmers.

On the other hand, it can be explicitly by-passed, if the high-level throughput is compatible with the pixel frequency.

Simultaneous routing path(s) are implemented within each transputer node as parallel processes, allocated at load-time by the local boot-process. This scheme has been proven acceptable since the time allocation strategy between concurrent processes running on the same transputer is based on I/O link activity.

7. 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 36 T800 based transputer modules.

Hardware and software design were based on a close integration of the data-flow computing model and the functional programming concept. This led to a simple highly regular hardware and an 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 such as extraction of main edges along predominant directions, labeling in connected component, color objects tracking, nagao filter, and more classical applications like convolutions, motion detection, edges extraction [12].

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 granularities. 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 super-computer dedicated to the resolution of partial derivative equations by finite difference methods.

8. Acknowledgements

The authors would like to thank Eric MERLET and Thierry GRELAUD for their work on transputer board integration.

APPENDIX 1: Example of C Data-Flow Process Source Code

```

1 #include "operator.h"
2
3 /* OPERATOR EXTERNAL DESCRIPTION
4 *
5 * FSWMIN(A:LINE(PIXEL)) = C:PIXEL,D:PIXEL
6 *
7 * Fixed width window extraction based upon global minimum
8 * Input: A = <a0,...,aN-1>
9 * Assume: w = param[0]
10 * Compute: j = Arg Min ( aj ) , j0=max(0,j-w/2) , j1=min(N,j+w/2)
11 * Output: C = j0,j1 */
12
13 #define N_IN 1          /* # of operator input links */
14 #define N_OUT 1        /* # of perator output links */
15
16 Parameter param[] = {  /* External parameters */
17     "width", 4 };     /* Name, default value */
18
19 /* OPERATOR IMPLEMENTATION */
20
21 void operator (Process *p,          /* Operator process */
22               Channel **in ,       /* Input links */
23               Channel **out,       /* Output links */
24               int *userParam ,     /* User supplied operator */
25               /* external PARAMETERS */
26               int nbUserParam)     /* # of user supplied operator */
27               /* external PARAMETERS */
28 {
29     int line[1024];                /* Local variables */
30     int width, linesize, min, at, from, to, i;
31
32     SetParams(param, userParam,
33               nbUserParam);        /* Get effective parameters */
34     width = param[0].value ;
35
36     for (;;) {                    /* Never ending loop */
37         linesize = ChanInLine7(in[_A],line) ; /* Read input line */
38         for ( i=1, at=0, min=line[0] ; i<linesize ; i++ )
39             if ( line[i] < min ) {
40                 at=i;
41                 min=line[i];
42             }
43         from = at-width/2 ; if ( from < 0 ) from=0 ;
44         to = at+width/2 ; if ( to > n-1 ) to=linesize-1 ;
45         ChanOutPixel15(out[_C],from); /* Write output indexes */
46         ChanOutPixel15(out[_C],to);
47     }
48 }

```

References

- [1] C.C. Weems *et al* (1991), *Parallel Processing in the DARPA Trategic Computing Vision Program*, IEEE Expert, Oct 1991.
- [2] E. Allart and B. Zavidovique, *Functional Image Processing though Implementation of regular Data Flow Graphs*, 21st Annual Asilomar Conf. on Signals, Systems on Computers Pacific Grove CA, USA, 2-4 Nov 1987.
- [3] I. Koren, B. Mendelson, I. Peled, G.B. Silberman, *A Data-Driven VLSI Array for Arbitrary Algorithms*, IEEE Computer, Oct 1988.
- [4] J. Backus, *Can programming be liberated from Von Neumann Style? A functional style and its algebra of programs*, Comm. of ACM, Vol. 21, No 8, Aug 1978.
- [5] A.L. Davis, S.A. Keller, *Data-Flow Program Graph*, IEEE Computer, Feb 1982, pp 26-40.
- [6] 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.
- [7] G. Quénot, B. Zavidovique, *A Data-Flow processor for Real-Time Low-Level Image Processing* IEEE Custom Integrated Circuits Conference, San-Diego, CA, USA, 13-16 May 1991.
- [8] G. Quénot, B. Zavidovique, *The E.T.C.A. massively Parallel Data-Flow Computer for Real-Time Image Processing*, IEEE Int Conf on Computer Design, Cambridge MA, USA, 14-16 Oct 1992.
- [9] J.B. Dennis, *Data Flow Supercomputers*, Computer, Vol 13, Nov 1980, pp 48-56.
- [10] W.B. Ackermann and J.B. Dennis, *VAL - A Value-Oriented Algorithmic Language: Preliminary Reference Manual*, MIT/LCS/TR 218, Jun 1979.
- [11] J. Sérot, G. Quénot, B. Zavidovique, *A functional Data-Flow Architecture dedicated to Real-Time Image Processing*, Proc IFIP WG10.3, Working Conference on Architectures and Compilation techniques for fine and medium grain parallelism, Orlando, FL, USA, 20-22 Jan 1993.
- [12] G. Quénot, C. Coutelle, J. Sérot, B. Zavidovique, *Implementing image processing applications on a real-time architecture*, Computer Architecture for Machine Perception Workshop (CAMP'93), New Orleans, Lousiana, USA, Dec 1993.