A Modular Software Architecture for Real-Time Video Processing[1]

Alexandre R.J. François and Gérard G. Medioni

Integrated Media Systems Center / Institute for Robotics and Intelligent Systems
University of Southern California, Los Angeles, CA 90089-0273
{afrancoi,medioni}@iris.usc.edu

Abstract. An increasing number of computer vision applications require on-line processing of data streams, preferably in real-time. This trend is fueled by the mainstream availability of low cost imaging devices, and the steady increase in computing power. To meet these requirements, applications should manipulate data streams in concurrent processing environments, taking into consideration scheduling, planning and synchronization issues. Those can be solved in specialized systems using ad hoc designs and implementations, that sacrifice flexibility and generality for performance. Instead, we propose a generic, extensible, modular software architecture. The cornerstone of this architecture is the Flow Scheduling Framework (FSF), an extensible set of classes that provide basic synchronization functionality and control mechanisms to develop data-stream processing components. Applications are built in a data-flow programming model, as the specification of data streams flowing through processing nodes, where they can undergo various manipulations. We describe the details of the FSF data and processing model that supports stream synchronization in a concurrent processing framework. We demonstrate the power of our architecture for video processing with a real-time video stream segmentation application. We also show dramatic throughput improvement over sequential execution models with a port of the pyramidal Lukas-Kanade feature tracker demonstration application from the Intel Open Computer Vision library.

1 Introduction

The past few years have seen a dramatic intensification of research activity in the field of video analysis, especially for motion analysis, stabilization, segmentation and tracking (see e.g. [4][1][13][3]). This was made possible in part by the mainstream proliferation of imaging devices and the steady increase in computing power. However, most video processing algorithms still operate off-line. Only a few real-time systems have been demonstrated (see e.g. [2][6][12]). Furthermore, in many cases, a (sometimes extrapolated) throughput of 5-10 frames per second (fps) is considered “real-time”, with the assumption that increasing computing power will eventually allow the same algorithms to perform at 30 fps. For direct scalability to be at least possible, the supporting systems must rely on multi-threading to ensure that all the available computing power is utilized efficiently. Consequently, such difficult issues as scheduling, planning and synchronization, must be specifically and carefully addressed. In traditional real-time vision systems, those are handled with clever *ad hoc* designs and implementation techniques. Flexibility, generality and scalability may be sacrificed for performance. Temporally sensitive data handling is a major requirement in multimedia applications, although the emphasis is put on the capture/storage/transmission/display string. Examples are BMRC's Continuous Media Toolkit [11] and MIT's VuSystem [9], that both implement modular dataflow architecture concepts. Their control mechanisms, however, are more concerned with the circulation of data between processing modules (possibly distributed), and they do not provide a complete uniform processing model. We propose the *generic*, *extensible*, *modular* software architecture presented in figure 1. A *middleware layer* provides an abstraction level between the low-level services and the applications, in the form of software components. An *application layer* can host one or several data-stream processing applications built from instances of the software components in a data-flow based programming model.

This paper is organized as follows: in section 2 we describe the Flow Scheduling Framework (FSF), cornerstone of our architecture, that defines a *common generic data and processing model* designed to support concurrent data stream processing. Applications are built in a data-flow programming model, as the specification of data streams flowing through processing centers, where they can undergo various manipulations. In section 3, we demonstrate the

[1] This research has been funded by the Integrated Media Systems Center, a National Science Foundation Engineering Research Center, Cooperative Agreement No. EEC-9529152, with additional support from the Annenberg Center for Communication at the University of Southern California and the California Trade and Commerce Agency.

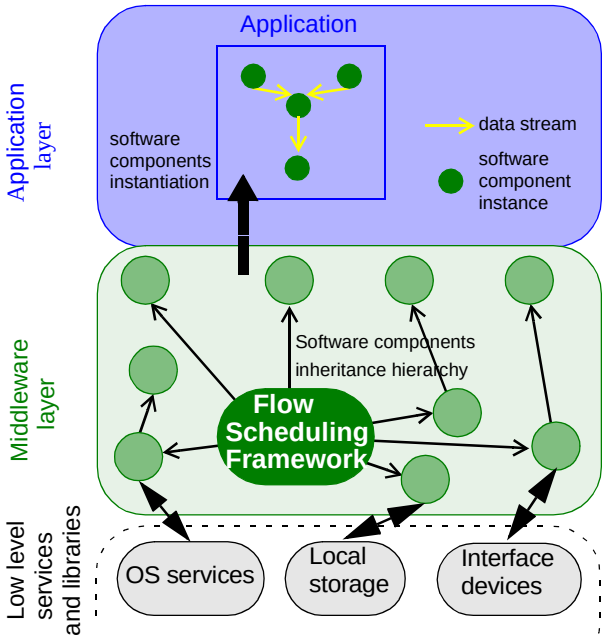

Fig. 1. A modular software architecture.

power of our architecture for vision systems with a real-time video stream processing application and with comparative performance tests conducted with the pyramidal Lukas-Kanade feature tracker demonstration application from the Intel Open Computer Vision library. We conclude the paper in section 4 with a summary of our contribution.

2 The Flow Scheduling Framework (FSF)

The FSF is an extensible set of classes that provide basic synchronization functionality and composition mechanisms to develop data-stream processing components. The FSF specifies and implements a *common generic data and processing model* designed to support stream synchronization in a concurrent processing framework. Applications are built in a data-flow programming model, as the specification of data streams flowing through processing centers, where they can undergo various manipulations. This extensible model allows to encapsulate existing data formats and standards as well as low-level service protocols and libraries, and make them available in a system where they can inter-operate.

Dataflow processing models are particularly well suited to data-stream processing application specification. However, traditional dataflow models were not designed to handle on-line data streams, and thus are not directly applicable to our design. An application is specified as a dependency graph of processing nodes. In the traditional approach (see figure 2a), each processing node has a fixed number of inputs and outputs, each of a given data type. When all the inputs are available for a given node, they are processed to produce the node's outputs. This processing can be triggered either manually or automatically. The production of the outputs in one node triggers or allows the processing in other nodes, and the procedure is propagated down the graph until the final outputs are produced. This approach is adequate in the case of deterministic processing of static data. Input type checking can be done at application design time. A first problem arises when the data type of an input is known at design type, but not the number of objects of that type. This situation can be handled at the price of more complex structures. A deeper deficiency is the static model on which, ironically, dataflow programming is based: the "dataflow" is regulated off-line, implicitly. There is no support for on-line processing, time consistency or synchronization, and process parameters are not integrated in the data model which makes any type of feed-back impossible. Extending the static dataflow model to a dynamic, data stream model is not trivial: process inputs are no longer static, unique objects (e.g. a single image), but time samples entering the system at a given rate (e.g. the frames of a video stream). Consequently, as process completion order is not deterministic, data must be kept available in the system until all dependent processes have been com-

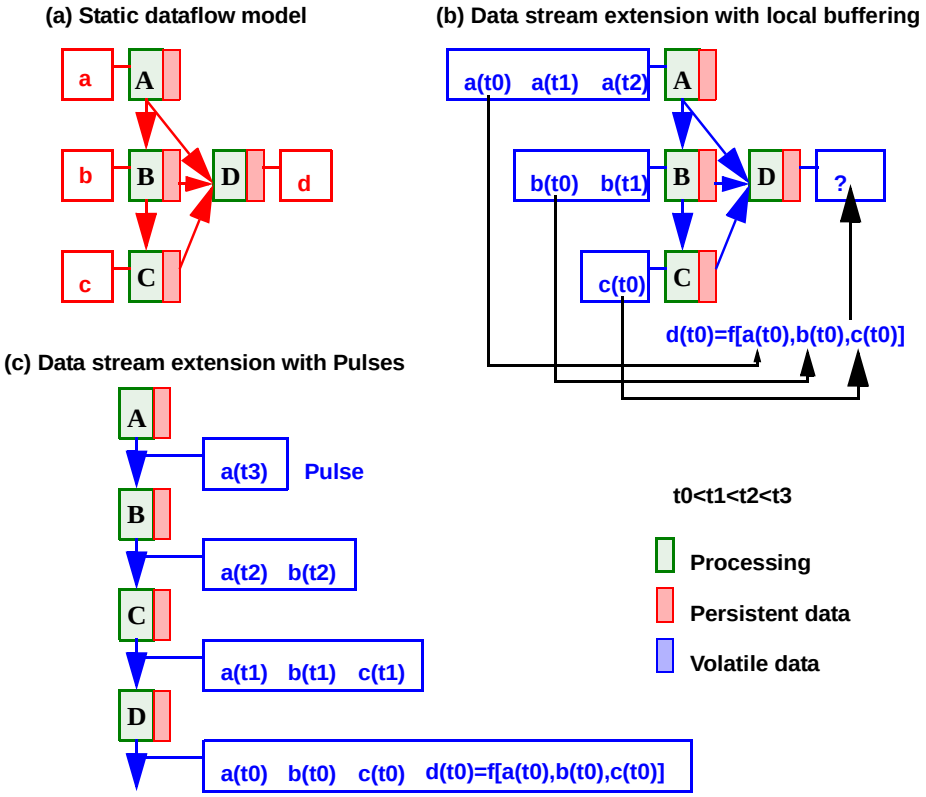

Fig. 2. Extension of the static dataflow model to handle data streams.

pleted. The straightforward extension to the static model, using local buffers in processing centers as described in figure 2b, although functional in simple cases, introduces several fundamental problems, such as the persistence of computed samples in the system and the collection of samples stored in independent buffers for data or process synchronization: if a process depends on data produced by several other processes, it should perform on synchronous inputs (i.e. data produced from the same sample), which requires, if buffers are used, to search all the buffers of all the input processes for samples with a particular time stamp. In order to avoid those problems, we introduce volatile carriers for synchronous data, called *pulses*, that are flowing down the stream paths (figure 2c). The color coding defined in figure 2 (green for processing, red for persistent data and blue for volatile data) is used consistently in the remainder of this paper.

2.1 Overview of FSF Concepts

Information is modeled as data *streams*. A stream represents one or several synchronized objects, i.e. expressed in the same time referential. An application is specified by a number of streams of various origins and the manipulations they undergo as they pass through processing nodes called *cells*. *Intra-stream processes* include formatting, alteration and presentation. A stream can originate from local sources such as local storage or input devices, or can be received from a network. It can also be created internally as the result of the manipulation of one or several other streams. As a stream goes through different cells, it can be altered. After processing, a stream can be stored locally, sent to output devices or sent on a network. *Inter-stream operations* are necessary for synchronization, as time referential transformations require the exchange or re-organization of data across streams. Streams occur in an application as time samples, which can represent instantaneous data, such as samples describing the state of a sensor at a given time, or integrated data describing, at a given time, the state of one or several objects over a certain period of time. We make a distinction between *active streams*, that carry volatile information, and *passive streams*, that hold persistent data in the application. For example, the frames captured by a video camera do not necessarily need to remain in the applica-

tion space after they have been processed. Process parameters however must be persistent during the execution of the application.

2.2 Data Model

We define a structure that we call *pulse*, as a carrier for all the data corresponding to a given time stamp in a stream. In an application, streams can be considered as pipes in which pulses can flow (in one direction only). The time stamp characterizes the pulse in the stream's time referential, and it cannot be altered inside the stream. Time referential transforms require inter-stream operations.

As a stream can represent multiple synchronized objects, a pulse can carry data describing time samples of multiple individual objects as well as their relationships. In order to facilitate access to the pulse's data, it is organized as a mono-rooted composition hierarchy, referred to as the pulse structure (see figure 5). Each node in the structure is an instance of a *node type* and has a *name*. The node name is a character string that can be used to identify instances of a given node type. The framework only defines the base node type that supports all operations on nodes needed in the processing model, and a few derived types for internal use. Specific node types can be derived from the base type to suit specific needs, such as encapsulating standard data models. Node attributes can be *local*, in which case they have a regular data type, or *shared*, in which case they are subnodes, with a specific node type. Node types form an inheritance hierarchy to allow extensibility while preserving reusability of existing processes.

2.3 Processing Model

We designed a processing model to manage generic computations on data streams. We define a generic cell type, called *Xcell*, that supports basic stream control for generic processing. Custom cell types implementing specific processes are derived from the Xcell type to extend the software component base in the system. In particular, specialized cell types can encapsulate existing standards and protocols to interface with lower-level services provided by the operating system, devices and network to handle for example distributed processing, low-level parallelism, stream presentation to a device, etc.

In an Xcell, the information carried by an active stream (*active pulses*) and a passive stream (*passive pulses*) is used in a process that may result in the augmentation of the active stream and/or update of the passive stream (see figure 3). Each Xcell thus has two independent directional stream channels with each exactly one input and one output. Part of the definition of the Xcell type and its derived types is the process that defines how the streams are affected in the cell. The base Xcell only defines a place-holder process that does not affect the streams.

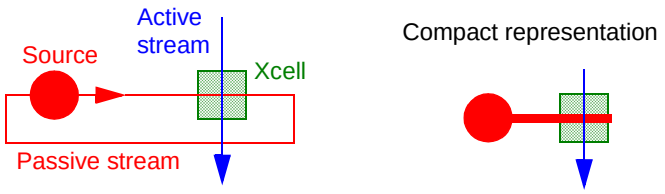

Fig. 3. Generic processing unit

Flow Control. In an Xcell, each incoming active pulse triggers the instantiation of the cell process, to which it is the input data. The process can only read the active pulse data and add new data to the active pulse (i.e. augment its structure). *Existing data cannot be modified.*

Process parameters are carried by the passive stream. A passive stream must form a loop anchored by exactly one source that produces a continuous flow of pulses reproducing its incoming (passive) pulses. The passive pulse arriving at the cell at the same time as the active pulse is used as parameters for the corresponding process instance. The process can access and may update both the structure and the values of the passive pulse.

When the process is completed, the active pulse is sent to the active output. The passive pulse is transmitted on the passive output, down the passive loop, to ultimately reach the source where it becomes the template for generated pulses. Note that the continuous pulsing on passive loops is only a conceptual representation used to provide a unified

model. Furthermore, in order to make application graphs more readable, passive loops are represented in compact form as self terminating bidirectional links (see figure 3).

In this model, the processing of different active pulses of the same active stream can occur in parallel, as pulses are received by the cell. However, a given active pulse being processed in a cell is not output until the process is complete. This multithreading capability is necessary if real-time performance is to be met, as it allows to reduce the system latency and maximize the throughput (see figure 4). Concurrent execution, however, introduces the major burden

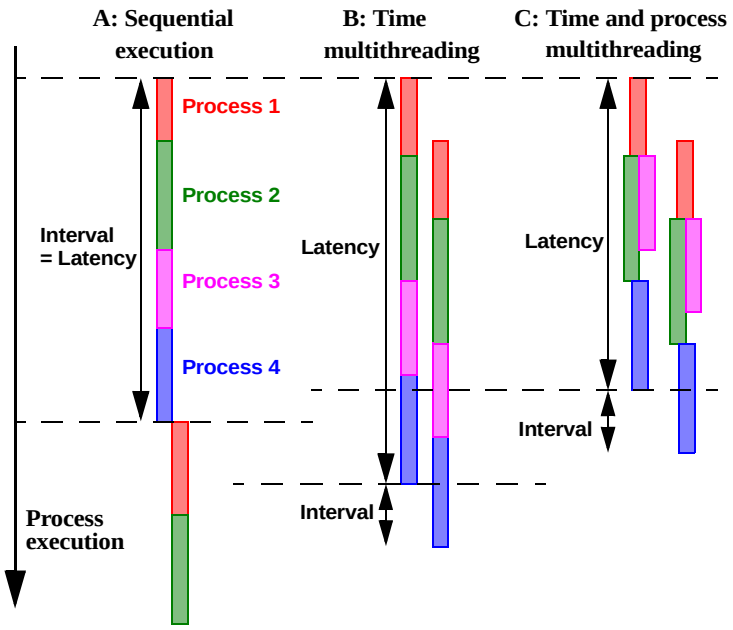

Fig. 4. Advantage of multithreading for temporally sensitive applications. Processes 2 and 3 are independent; Process 4 depends on both 2 and 3. With a sequential execution model, the processing lag also constrains the achievable rate (inversely proportional to the interval between the completion of the processes of two consecutive time samples). Multithreading separates the lag from the processing rate.

of having to keep track of data relationships, in particular dependency and synchronization information. In our model, a cell cannot alter the time stamp of an active pulse: time referential transformations, such as buffering or pulse integration, require to make the active pulse data persistent *via* a passive loop, as described later.

Filtering. In an Xcell derived cell, the process is defined as a function that takes as input data an active pulse and a passive pulse, and may augment the active pulse and/or modify the passive pulse. The process description includes the specification of the associated input and parameter types, i.e. substructures to look for in the active and passive pulses structures respectively. When designing custom cells, corresponding local data types and node types must be defined if not already available. A partial structure type is specified as a *filter* or a composition hierarchy of filters (see figure 5). A filter is an object that specifies a node type, a node name and eventual subfilters corresponding to subnodes. The filter composition hierarchy is isomorphic to its target node structure. When an active pulse is received, the cell process input structure must be identified in the active pulse, and similarly the parameter structure must be identified in the passive pulse before the process can be started. Finding substructures in pulses is called filtering, and is a subtree matching operation. In order to provide efficient access to relevant data in the pulses, filtering operations return *handles*, that are used in the process for direct access to the relevant structure nodes. A handle is a pointer to a pulse node. Handles can be organized in composition hierarchies isomorphic to filter and node structures. A handle or composition hierarchy of handles is formed as the result of the filtering of a pulse's structure.

The Node names specified in filters to identify instances can be exact strings or string patterns to allow multiple matches. The filtering process thus returns a list of handles, one for each match. The interpretation of multiple

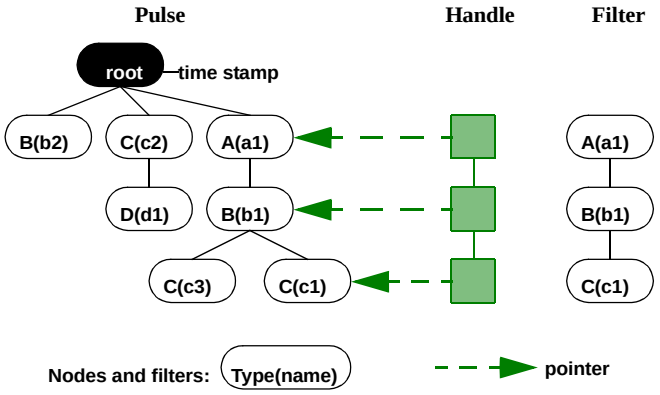

Fig. 5. Pulse filtering. To each process are associated input and parameter data types, in the form of substructures called filters, for which the incoming active and passive pulses structures (respectively) are searched. These filtering operations return handles, used during processing for direct access to relevant nodes.

matches in the active pulse is part of a specific process implementation. For example, the process can be applied to each match, thus allowing to define operations on objects whose count is unknown at design time.

Filtering efficiency is a major issue especially on active streams, since it must be performed independently on each new active pulse. In traditional dataflow models, data resulting from processes is not accumulated in a structure, but simply output from a processing node. Type checking can therefore occur at application design time, and there is no need to look for the relevant data in the incoming stream. After experimenting with this approach, we found that in applications where synchronization is a central issue, keeping data synchronized in our pulse structure is more efficient than separating data elements for processing and later having to put back together synchronous samples distributed in independent storage nodes. The hierarchical structuration of the pulse data is suitable for efficient implementation of filtering. Furthermore, the filtering approach allows to apply a same process to an undetermined number of objects in an active pulse while preserving the temporal relationship of the object samples. It also makes it possible for several cells on a passive loop to share part or totality of their parameters.

2.4 From Components to Applications

Stream Routing. An application is specified by a graph of cells with two independent sets of links: active and passive. Active and passive connections are different in nature, and cannot be interchanged.

There is no limitation on the number of cells on a same passive loop. Feed-back can thus be implemented either with cells that directly update their parameters in their process, or with separate cells for processing and update, if the update depends on some subsequent processes, as shown in figure 6. Updated parameters are used in processes as they become available on the corresponding passive loop, thus feed-back loops cannot create any interlock.

The set of active connections between the cells is a *dependency graph*. A cell using as input the result of another cell must be traversed after the cell on which it depends in order for its process to be executed (otherwise the filtering fails and pulses are transmitted without any process). Independent cells can be traversed in arbitrary order, and their processes can occur in parallel. To take advantage of this parallelism, we define the *Scell* type, which is a stream splitter (see figure 7). An Scell has one active input and two active outputs. Scells transmit incoming input active pulses on both active outputs simultaneously.

The only local synchronization enforced in the cells is that an active pulse does not reach a given cell before it has been completely processed by the previous cell on the active path. In order to enforce this constraint when a stream follows several parallel paths, we introduce the *Wcell* (see figure 7), which has two active inputs and one active output. A Wcell is needed when a cell depends on several independent cells traversed in parallel: the Wcell waits until it receives the same pulse on both its active inputs before sending it to its active output.

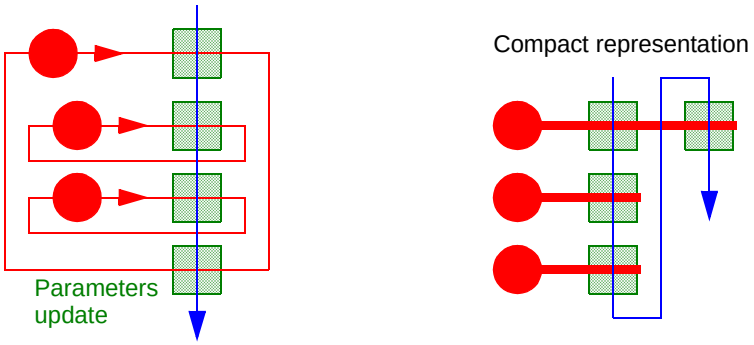

Fig. 6. Parameter feed-back on a passive loop

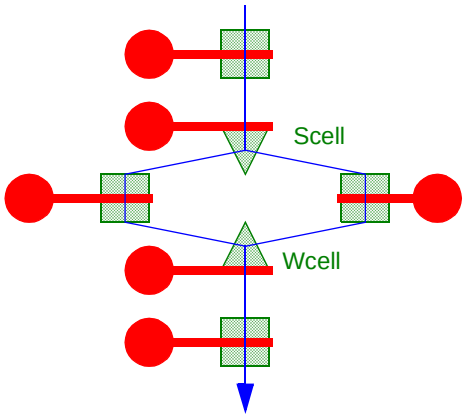

Fig. 7. Stream routing for parallel execution

There is no *a priori* assumption on the time taken by cell processes. If the processing time in the upstream cells varies from pulse to pulse, active pulses might reach a given cell out of order. If simple time consistency can be enforced in cell processes, more elaborate synchronization mechanisms are usually required.

Synchronization. In our framework, synchronization reduces to time referential transformations. As already pointed out, each active stream has its own time referential, to which the pulses' time stamps relate. All passive streams share the same system time referential. The time referential in a stream cannot be altered. Time referential transformations thus require the transfer of pulses across different streams. To do so, active pulses must be stored in a passive loop before they can be reintroduced in a different active stream. Intuitively, if any time reorganization of the active pulses is to occur, they must be buffered. The minimal setup required is presented in figure 8. Active pulses are buffered, by a specialized cell, to a passive loop in which the pulses describe the content of the buffer at a given time. Buffered pulses may undergo a time transform. Another specialized cell on the same passive loop can retrieve the pulses from the buffer and restitute them in an active stream at the desired rate, determined either by a dedicated timer or by incoming active pulses. This operation may involve resampling. More generally, if two active streams are to be combined in one single active stream (see figure 8), the pulses from both streams must be buffered in a passive loop structure after undergoing a time transform to place them in a common time referential. Synchronous pulses can be retrieved from the buffer and placed in a same active pulse, after an eventual time transformation. This pattern easily generalizes to several active streams being recombined into several different active streams. Although the time transformation, buffering and eventual resampling when de-buffering are generic concepts, the actual operations to carry are data (node) type and cell process dependent, and thus are not part of the framework definition.

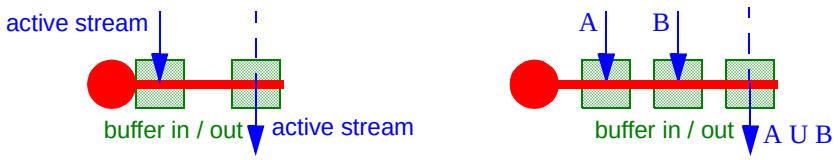

Fig. 8. Pulse buffering (left) and streams synchronization (right)

3 Implementation and Evaluation

We have implemented a prototype FSF in the form of a C++ library, made of a set of C++ classes implementing the basic cell and node types, pulses, filters and handles. All our developments are currently targeted at the Windows NT/ 2000 operating system, but the library can easily be ported to other multitasking operating systems. Note that this implementation does not feature any resource exhaustion control mechanism, which would require using Quality of Service functionalities from a real-time Operating System.

Using our prototype FSF library, we have developed two applications to illustrate the power of our design both in terms of application development and run-time performance. Building these application required to define a number of custom cells, together with the node types required for their parameters, and a few other node types for the basic data types manipulated in the applications. Many image processing operations are carried out using the Intel Image Processing Library [7], which we encapsulated in our system. For all reported experiments we used a dual Pentium III 550 MHz machine running Windows 2000.

3.1 Real-Time Video-Stream Segmentation

Our first demonstration is an adaptive color background model-based video-stream segmentation application, as described in [5]. The corresponding application graph is presented in figure 9. On frames 240x180 pixels, the application, which involves such expensive operations as the conversion of each RGB frame to the HSV color system, runs at 30 fps, a frame rate far superior to the one achieved in our previous stand alone implementations. The multi-threaded processing model makes real-time processing possible by utilizing all the available processing power.

Implementing the different components needed to realize this application was simpler than in a stand-alone system, because of the well-defined structuration imposed by the FSF model. Consequently, the overall application was easier to put together. In particular, in a single thread implementation, the feed-back loop is very inefficient in terms of system throughput. Furthermore, if all the described node types and cells mentioned actually had to be developed, most of them are generic and can easily be re-used in other applications. As more modules are developed, less generic components will have to be implemented to realize a given application. Reversely, as new components are developed, they can be used in existing applications by substituting functionally equivalent cells.

3.2 Lukas-Kanade Tracker

Our second demonstration aims at emphasizing the importance of multithreading and synchronization issues in on-line and real-time processing. The recently released Intel Open Computer Vision Library (OpenCV) [8] contains the source code for a large number of useful vision algorithms. As a library, it does not offer any application support, although simple demonstration applications are distributed with it. A representative video processing demonstration is the pyramidal Lukas-Kanade feature tracker [10]. For our purpose, we do not consider the algorithm itself, nor its implementation, but rather the way in which it is operated. The OpenCV demonstration is a simple Windows application built with the Microsoft Fundation Classes (MFC). The dynamic aspect of the video processing is not multi-threaded, and is driven by the MFC's display callback message mechanism. As a result, frame capture, processing and display cycles are executed sequentially, and thus the throughput is limited to a few fps while only a fraction of the available processing power is used (not to mention the inability to take advantage of an eventual second processor).

Using the exact same processing code as provided in the OpenCV LKTracker demo, we have built a tracker component and tested it in the simple application whose graph is presented in figure 10. Encapsulating the tracking code

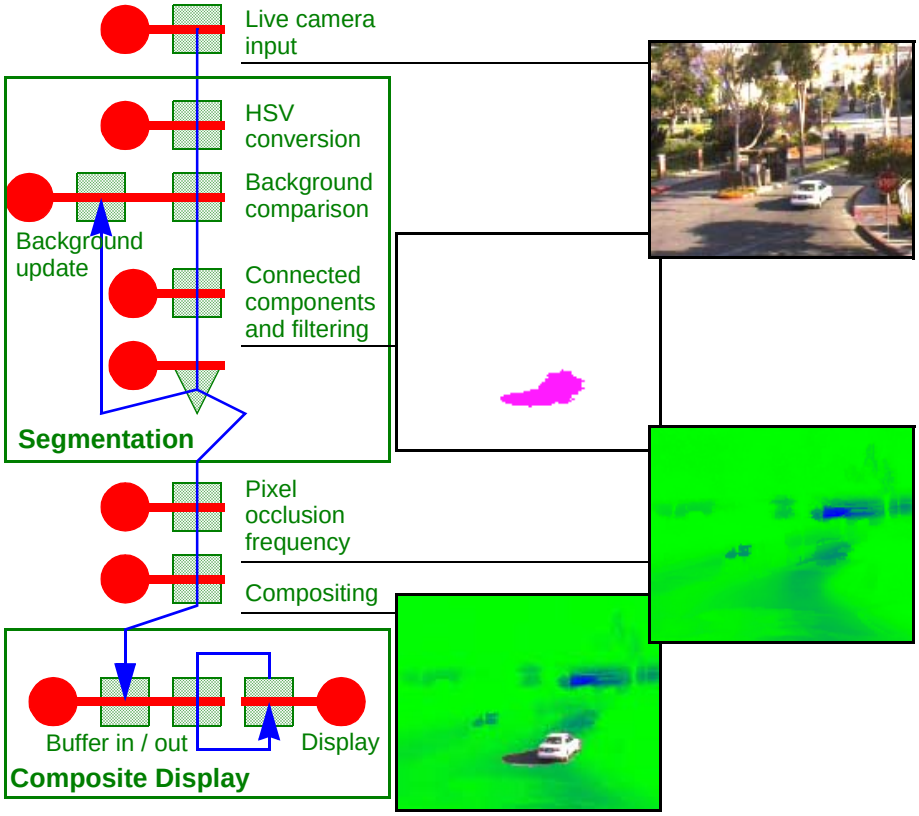

Fig. 9. Real-time video stream segmentation application

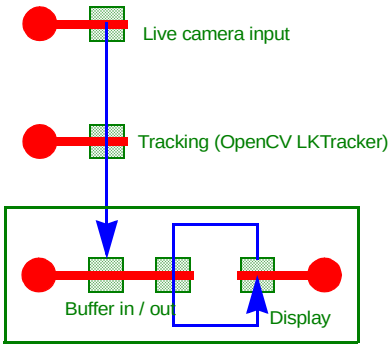

Fig. 10. OpenCV Lukas-Kanade tracker port to our multithreaded framework

in our framework and building the application took only a few hours. The major difference with the MFC version is that capture, processing and display are now carried out by independent threads for each frame. We have conducted the tests for several input frame rates and number of tracked points. A plot of the processor load against the frame rate is presented in figure 11. The frame rate in the MFC demonstration is limited by the performance of subsystems of the machine on which the application is running (mainly capture card, main processor and display card), and impossible to control. On the other hand, our architecture allows to control the frame rate, is clearly scalable and allows to better take advantage of all the available processing power.

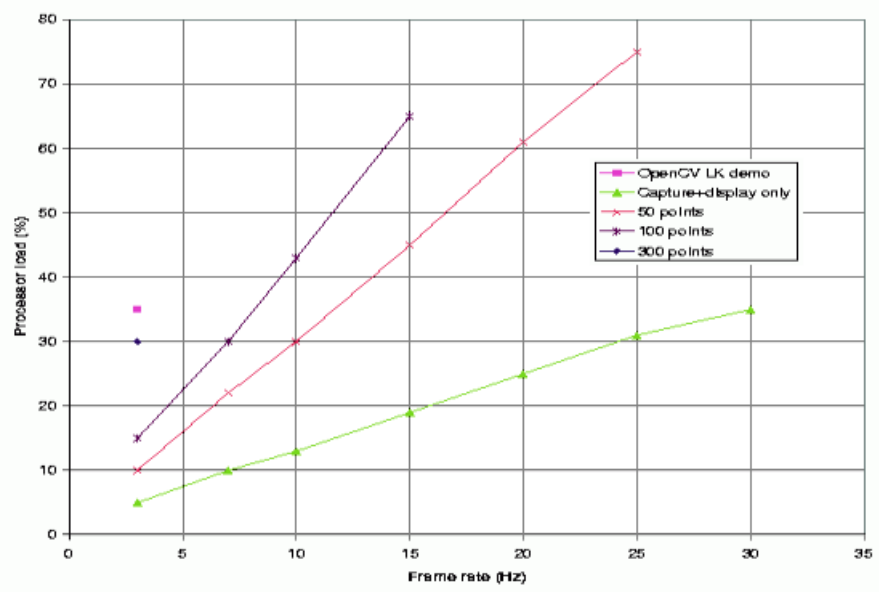

Fig. 11. Processor load against frame rate comparative results obtained with the OpenCV LKTracker demonstration for various frame rates and numbers of tracked points.

Note that this simple demonstration, although useful in making our point for a concurrent processing model, is also an example of bad transition from a sequential processing model. Inappropriate design of the tracking component is the reason why the throughput is not limited by processing power, but instead by exclusive concurrent access to tracking data.

4 Conclusion

We have presented a generic, extensible, modular software architecture to support real-time vision applications. The cornerstone of this architecture is the Flow Scheduling Framework (FSF), an extensible set of classes that provide basic synchronization functionality and composition mechanisms to develop data stream processing components. The FSF specifies and implements a *common generic data and processing model* designed to support stream synchronization in a concurrent processing framework. Applications are built in a data-flow programming model, as the specification of data streams flowing through processing centers, where they can undergo various manipulations. This extensible model allows to encapsulate existing data formats and standards as well as low-level service protocols and libraries, and make them available in a system where they can inter-operate.

We have demonstrated the importance of multithreading and synchronization issues for on-line, real-time processing, and thus the relevance of our architecture design. The demonstration applications also prove the practicality and efficiency of its implementation, both in terms of development and execution. Our experiment results also confirm that concurrent processing of data streams not only requires a software architecture supporting synchronization and flow control mechanism, but also careful application design, as a straightforward port of the static (or batch), sequential version will usually prove inadapted to the dynamic model.

Future developments include the incorporation of relevant existing CV algorithms, for example through the encapsulation of the Intel OpenCV Library. We also intend to use this system as a development platform for computer vision research, especially for the design of dynamic data structures and algorithms for real-time processing of video streams.

References

1. M. Ben-Ezra, S. Peleg, M. Werman, “Real-Time Motion Analysis with Linear Programming,” Computer Vision and Image Understanding, 78(1), April 2000, pp. 32-52.

2. T. E. Boult, R. Micheals, X. Gao, P. Lewis, C. Power, W. Yin and A. Erkan, “Frame-Rate Omnidirectional Surveillance and Tracking of Camouflaged and Occluded Targets,” in Proc. Workshop on Visual Surveillance, Fort Collins, CO, June 1999.
3. I. Cohen and G. Medioni, “Detecting and Tracking Moving Objects for Video Surveillance,” in *Proc. IEEE Conf. on Computer Vision and Pattern Recognition*, Fort Collins, CO, June 1999, vol. 2, pp. 319-325.
4. R. T. Collins, A. J. Lipton and T. Kanade, Special Section on Video Surveillance, IEEE Trans. on Pattern Analysis and Machine Intelligence, 22(8), August 2000.
5. A. R.J. François and G .G. Medioni, “Adaptive Color Background Modeling for Real-Time Segmentation of Video Streams,” in *Proc. Int. Conf. on Imaging Science, Systems, and Technology*, Las Vegas, NA, June 1999, pp. 227-232.
6. I. Haritaoglu, D. Harwood and L. S. Davis, “W4: Real-Time Surveillance of People and Their Activities,” IEEE Trans. on Pattern Analysis and Machine Intelligence, 22(8), August 2000, pp. 809-830.
7. Intel Image Processing Library, http://developer.intel.com/vtune/perflibst/ipl/
8. Intel Open Computer Vision Library, http://www.intel.com/research/mrl/research/cvlib/
9. C. J. Lindblad and D. L. Tennenhouse, “The VuSystem: A Programming System for Compute-Intensive Multimedia,” IEEE Jour. Selected Areas in Communications, 14(7), September 1996, pp. 1298-1313.
10. B. D. Lucas and T. Kanade, “Optical Navigation by the Method of Differences,” in Proc. Int. Joint Conf. on Artificial Intelligence, Los Angeles, CA, August 1985, pp. 981-984.
11. K. Mayer-Patel and L. A. Rowe, “Design and Performance of the Berkeley Continuous Media Toolkit,” in Multimedia Computing and Networking 1997, pp. 194-206, Martin Freeman, Paul Jardetzky, Harrick M. Vin, Editors, Proc. SPIE 3020, 1997.
12. G. Medioni, G. Guy, H. Rom and A. François, “Real-Time Billboard Substitution in a Video Stream,” *Multimedia Communications*, Francesco De Natale and Silvano Pupolin (Eds.), Springer-Verlag, London, 1999, pp. 71-84.
13. K. Toyama, J. Krumm, B. Brumitt and B. Meyers, “Wallflower: Principles and Practice of Background Maintenance,” in *Proc. IEEE Int. Conf. on Computer Vision*, Corfu, Greece, September 1999, pp. 255-261.