VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

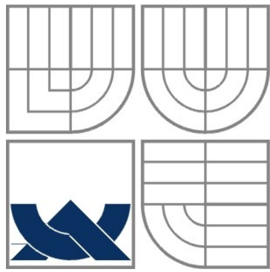
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

REAL-TIME IMAGE PROCESSING ALGORITHM IMPLEMENTATION

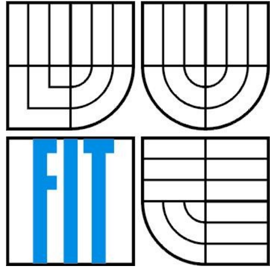
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE Andrej Rypák
AUTHOR

BRNO 2009



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

IMPLEMENTÁCIA ALGORITMU NA SPRACOVANIE OBRAZU PRACUJÚCEHO V REÁLNOM ČASE

REAL-TIME IMAGE PROCESSING ALGORITHM IMPLEMENTATION

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — Andrej Rypák
AUTHOR

VEDOUCÍ PRÁCE — Ing. Thierry Grandpierre, Ph.D.
SUPERVISOR

BRNO 2009

Abstrakt

V tomto projektu se zabývám rozšířením softwaru využívaného k rychlému vývoji a optimalizaci aplikací běžících v reálném čase pro architektury, které jsou složené z více obecně různorodých výpočetních prvků. Software se nazývá SynDEx. Po krátkém úvodu prezentuji techniky, které jsem použil pro implementaci dvou různých specifických algoritmů pro segmentaci a značkování obrazu a komponenty pro zobrazování výsledků.

Abstract

In this project I deal with designing program components for a rapid prototyping software called SynDEx that is used for automatic code generation, real-time application development and optimization for multicomponent mixed architectures. I briefly introduce this software. I present techniques I used to implement two different specific image processing chains and a displaying component. The chains perform image segmentation and labelling.

Klíčová slova

aplikace pracující v reálném čase, rychlé prototypování, automatické generování kódu, SynDEx, zpracování obrazu, dělení obrazu, detekce hran, uzavírání křivek, filtr „kostra“, prahování, značkování, zobrazující komponenta na bázi OpenGL

Keywords

real-time applications, rapid prototyping, automatic code generation, SynDEx, image processing, image segmentation, edge detection, contour closing, skeleton filter, labelling, thresholding, OpenGL display component

Citace

Andrej Rypák: Real-time Image Processing Algorithm Implementation, bakalářská práce, Brno, FIT VUT v Brně, 2009

Real-time Image Processing Algorithm Implementation

Prehlásenie

Prehlasujem, že som túto bakalársku prácu vypracoval samostatne pod vedením Ing. Thierryho Grandpierra, Ph.D..
Ďalšie informácie mi poskytli Ing. Eva Dokladalová, Ph.D. a Prof. Michel Couprie.
Uviedol som všetky literárne pramene a publikácie, z ktorých som čerpal.

……………………
Andrej Rypák
25.5.2009

Poďakovanie

I would like to thank my project supervisor, teacher and scientist at ESIEE, Thierry Grandpierre for great approach, friendly atmosphere, valuable information and correct guidance. I would also like to thank ESIEE and especially the Computer Department for providing me with up-to-date equipment and an appropriate laboratory and scientist Michel Couprie for helpful information about certain algorithms. My further thanks belong to both Eva Dokladalova for assistance with the project and Jan Černocký for further project supervision at FIT VUT and administration help.

©Andrej Rypák, 2009
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Contents

Contents....1
1 Introduction....3
2 Theoretical Background....4
2.1 AAA – Algorithm-Architecture Adequation....4
2.1.1 Multicomponent Architectures and Real-time Application Rapid Prototyping....4
2.2 The Algorithm Chain....5
2.2.1 Edge Detection Using Canny/Deriche Filter....6
2.2.2 Smoothing and Edge Detection – Garcia Lorca....7
2.2.3 Thresholding....8
2.2.4 Contour Closing – Image Segmentation....9
2.2.5 Labelling....10
2.2.6 Thinning and Crest Reconstruction....10
2.3 IBM Cell Processor....11
3 Tools Description....12
3.1 SynDEx....12
3.1.1 Short Description....12
3.1.2 Usage....12
3.1.3 Modular Programming....13
3.2 M4 Macro Processor....14
3.3 GNU C and C++ Compilers, GNU Make....14
3.4 Extendibility, Modifiability and Modularity....14
4 Implementation Methods....15
4.1 File and Stream Handling....15
4.1.1 Endianness....15
4.1.2 IBM Cell versus Intel Based Machines....15
4.1.3 Word and Double-word Splitting and Merging....16
4.2 The Interface....17
4.2.1 Memory Allocation....17
4.2.2 Structures....18
4.2.3 The Interface – Final Decision....18
4.3 SynDEx Modules – The Definitions....19

4.4 Encapsulation of Algorithms into Modules....19
4.5 Macro Definitions....19
4.5.1 Macro Example....20
4.5.2 Macros – Part 2....20
5 Program Components....21
5.1 The Display Component....21
5.1.1 How It Works....21
5.1.2 What It Offers....21
5.2 Two Algorithm Chains....22
5.3 File Handler....22
5.4 IR Camera Stream Reader....22
5.5 Dilatation, Erosion....23
6 Conclusion....24
7 Bibliography....25
8 Attachments....26

1 Introduction

The world around us is filled with multimedia. We do not only use them for amusement, but also for many other purposes, such as surveillance, security and safety. Good examples are IR cameras able to recognize a person present on the road in any weather and light conditions, cameras that monitor driver's face to recognize a micro-sleep, voice-recognition locks and many others. This is where real-time applications are well utilized.

Real-time applications for multi-component architectures are usually very specific and need whole teams to be developed. But what if the users were provided by easy-to-use software and they could easily build their own processing chains and applications?

SynDEx is such a software. And the main idea of this project was to extend its functionality by developing image-processing modules for a library it uses. Real-time applications have to guarantee a certain maximum response time and a certain level of precision. For this reason, lightweight solutions still producing good results are needed.

The first part of my work consisted in implementation of a specific image processing algorithm chain composed of several independent algorithms. Each of them had to be encapsulated in a separate and autonomous module. Detailed description of the algorithm chain is in section 2.2.

The tools I used, including SynDEX software, are briefly introduced in chapter 3.

During the second stage I had to deal with endianness and IBM Cell processor based architecture described in section 2.3 and my solution in section 4.1.

Then I had to modify and optimize my previously implemented algorithm to make it ready for both homogeneous and heterogeneous architectures containing more processors, processor cores or more DSP (digital signal processing) units. To meet these requirements I had to create an interface described in section 4.2.

Finally, I tested several applications built on my modules. These are fairly presented in chapter 5.

2 Theoretical Background

2.1 AAA – Algorithm-Architecture Adequation

To better understand real-time applications and problems linked with their development we need to look deeper into the background behind the applications as well as the purpose why we call them real-time.

Algorithm-Architecture Adequation is a prototyping methodology which allows us to rapidly develop and optimize the implementation of real-time algorithms. The word *adequation* is a French expression, meaning an efficient match. The goal of the AAA methodology is to find out the best implementation of an algorithm specifying the functions (operations) the application has to perform onto a multicomponent architecture, while satisfying real-time and embedding constraints [1]. It uses special optimization heuristics to evaluate not only the time needed to process a specific operation on a certain set of data and by one of the operators specified (a processing unit), but also the time needed for the communication between processing units, that can be of different kind and interconnected with various media type. This is how AAA can predict and guarantee a certain application response time, thus meeting the real-time constraint, which along with a specified level of precision is the basic requirement for real-time applications.

As with all well-written applications we want them to be reusable. Rapid prototyping helps us easily reuse, modify and extend application components and this way develop whole applications in much shorter time increasing the effectiveness of developers. Also, once well tested modules are accepted as appropriate and correct, there is no need to further debug them when an error occurs in an application further reducing the development time.

2.1.1 Multicomponent Architectures and Real-time Application Rapid Prototyping

The problem with rapid prototyping when used for real-time applications originates in neglecting some aspects that do not seem to alter the processing time much in most of cases, but sometimes can drastically reduce effectiveness of an algorithm and in some cases even cause failure to fit into the time limit for the real-time algorithm chain. This problem usually appears when the prototyping software does not take into account the inter-processor communication.

The AAA introduced above uses characterization and performance prediction to determine maximum process time for each chain, and therefore it is suitable even for rendering of code for distributed real-time embedded data-flow applications supposed to run on multicomponent mixed (heterogeneous) architectures. It uses distribution and scheduling heuristics to perform optimizations.

2.2 The Algorithm Chain

A serial connection of several independent functions, each encapsulating a special image-processing algorithm, can be called an algorithm chain. In such a chain the output of one element is passed on to the following function as its input. The chain starts with the first element receiving an input and ends when the final element produces an output.

Let me introduce the main image-processing chain I worked with during my research. It accepts an uncompressed black&white image stored in an array of pixels as its input and produces a colour-labelled segmented image as an output, stored in an equivalent structure. The data-flow of the chain is shown in the Figure 1 below.

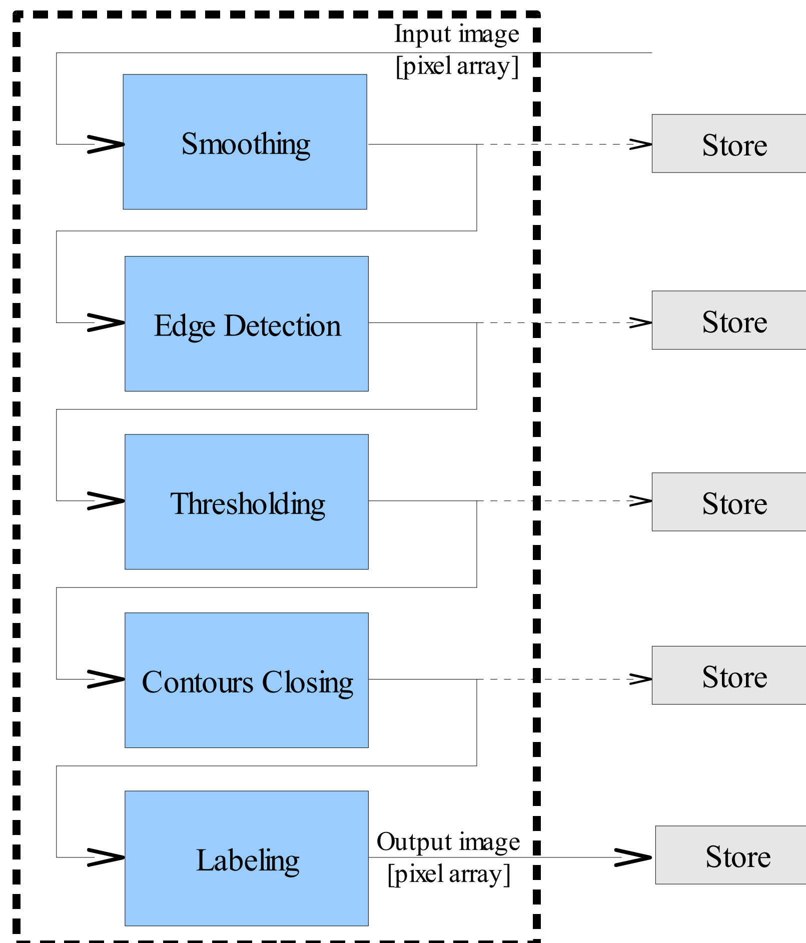


Figure 1: The data-flow of segmentation and labelling image processing chain. Dashed arrows show partially processed data output

2.2.1 Edge Detection Using Canny/Deriche Filter

The well known Sobel filter (and Robert, used below) and some other filters based on convolution of the image and a simple matrix may seem to be the best choice for real-time image processing because they use cheap operations and are easy to be computed. However, to prepare a good quality image for further processing, Sobel's solution is not sufficient. Its performance deteriorates rapidly when it is used for blurred and noisy images [3]. To meet our requirements, we need to reach out for more complex solutions that are not sensitive to image noise and detect real edges only.

One of these is Canny's optimal edge detector [4]. He defined a model of a step in white Gaussian noise and assumed that the detection was performed by convolving the model with a spatial antisymmetric function which should give a good detection, good localization and one response to one edge. Canny found an approximation of the optimal filter, the first derivative of a Gaussian, and he used it as a F.I.R filter (Finite Impulse Response).

Canny's filter gives very good results, but its downside is high computational load, which is unacceptable in real-time applications. Rachid Deriche came out with an improved version of Canny's edge detection algorithm, which is known as the Canny/Deriche Algorithm [5]. It is implemented as 2-D second order Infinite Impulse Response (I.I.R) digital filters [3].

Figure 2: Original input image

Figure 3: original Canny/Deriche filter applied. Alpha parameter set to 3.8 to produce similar output as the edge detector below

This algorithm has been taken and optimized by Garcia Lorca, as described in the next chapter (2.2.2).

2.2.2 Smoothing and Edge Detection – Garcia Lorca

A raw image sometimes contains excessive amount of noise. Especially when dealing with real images captured by digital imaging devices, such as cameras, that do not deliver the best image all the time. To get rid of the noise, we use smoothing algorithms and only then edge detector algorithms in order to detect the edges.

Figure 4: Original input image

Figure 5: Smoothed image - the output of smoothing process with the depth set to 0.2

In my research I dealt with Garcia Lorca's improved version of the Canny/Deriche edge detection filter (above). It consists of Canny/Deriche based recursive smoothing filter followed by horizontal and vertical Robert's gradient. The smoothing was specially optimized for embedded systems, RISC processors and DSP (digital signal processor) units [2]. The depth of the smoothing algorithm can be set by a floating-point number parameter. The algorithms produce good results in relatively short time [2]. This is the reason why we focused mainly on this edge-detection method in our project.

Figure 6: Previously smoothed image

Figure 7: Application of the edge-detection filter

2.2.3 Thresholding

After detecting all the edges the image has, we have to sort out the important ones. We use double thresholding to determine the leading contours and the auxiliary contours. The result is then passed to the following algorithm (2.2.4 - Contour Closing – Image Segmentation). The auxiliary contours help us find a closed region if the leading ones do not suffice.

The principle of thresholding is based on value comparison between a threshold and a pixel in a grey-scale image. A single-thresholding produces an array of pixels having minimum or maximum values (black or white). Double-thresholding produces an array of pixels that have 3 different values (black, white and grey).

Figure 8: The image passed to thresholding as an input...

Figure 9: and the image processed by automatic double thresholding

In our image-processing chain we use automatic double thresholding. The values of the thresholds are computed automatically from the values of the input picture. We use average of all the pixel luminance values to determine the lower threshold (for the auxiliary contours), then we do the thresholding. Afterwards, we evaluate the average again, but this time we only use the values of those pixels that passed the first thresholding. The pixels to pass this second threshold are the pixels defining the main contours of our image.

2.2.4 Contour Closing – Image Segmentation

This is the most demanding part of the chain. The algorithm selects which of the white points compose contours and which points can be removed so that the edges and the image topology would still be retained.

We use an algorithm Michel Couprie described, implemented and placed into an image handling library called *Pink*. It is based on skeleton algorithm [6]. In the first part, the algorithm iteratively looks for *simple points*, which are points of the raster that do not change the image topology upon being removed. Then it keeps removing those with lower priority until the state of stability is reached and no other simple points can be removed. To be able to do this, the algorithm builds up a morphological tree that represents the picture.

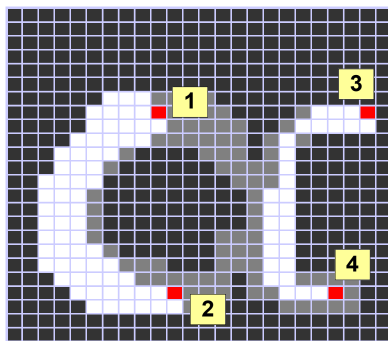


Figure 10: without skeleton – we are not sure how to connect the contours

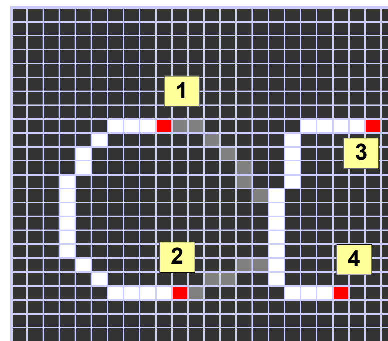


Figure 11: after applying the skeleton algorithm, closing of the contours is trivial

After the application of the skeleton filter, the closing of contours can be easily done using the main contours to surround an area and (if unsuccessful) trying to do so using the auxiliary ones.

Figure 12: the double-thresholding processed image

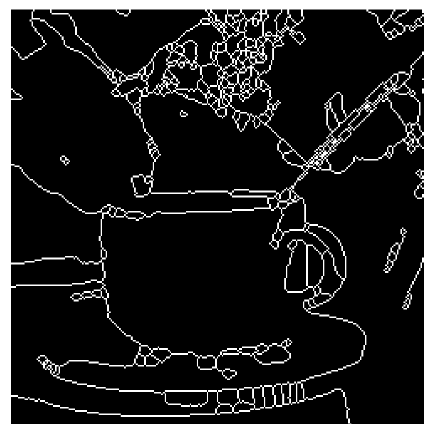


Figure 13: image divided into segments, segmented image

2.2.5 Labelling

Labelling is done in one fast step. Two lines of the image are scanned at once to determine, if the points are in the same region or not. If they are, the same colour is used for them. But if a new region is found, a different colour (or luminance value for grey scale images) is used.

Figure 14: after dividing the image into separate segments, the image is waiting to be "labelled"

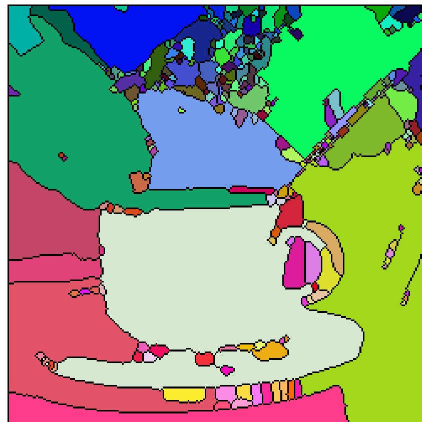


Figure 15: colour-labelled image regions

2.2.6 Thinning and Crest Reconstruction

These two algorithms produce output similar to the contour-closing one described above in the chapter 2.2.4. Thinning works also with grey scale images, so there is no need of placing a thresholding function in front of it. First thinning segmentation is used then the lines are reconstructed. The authors of these algorithms are M. Couprie and F.N. Bezerra [7].

Figure 16: thinning applied (after edge detection)

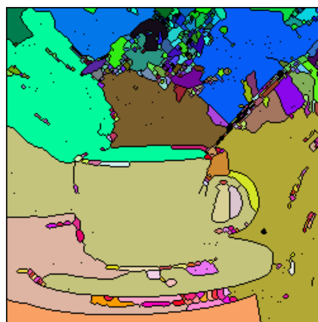


Figure 17: when labelling is applied (after crest reconstruction) we get a result similar to the one from the main chain

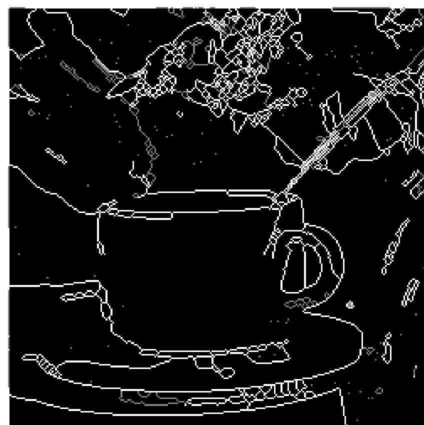


Figure 18: crest reconstruction (applied after thinning)

2.3 IBM Cell Processor

The *Cell Broadband Engine*, *Cell BE*, or simply *Cell* is a microprocessor designed to fill the gap between conventional desktop processors and more specialized high-performance processors (such as GPU) [8]. Cell is a heterogeneous architecture that combines one Power Processing Element (PPE) and eight synergistic processor elements (SPE). The PPE is optimized for control tasks and is fully compliant with the 64-bit Power Architecture, able to execute both 64 and 32 bit operations. This is why at the first sight Cell behaves like a conventional AMD or Intel desktop CPU. What makes it different is the presence of eight coprocessors designed for data processing delivering extraordinary computational power. Supporting single-instruction, multiple-data (SIMD), the Cell processor provides a high-performance multi-threaded execution environment for all applications.

To utilize the SPE coprocessors, special programming techniques have to be used. To create a well behaving and swiftly responding real-time application we have to take into account every aspect of the architecture, such as communication between PPE and SPE or between SPE and another SPE. This is where SynDEx comes handy. See chapter 3.1 for more details.

Cell processors use *big endian* byte order, on contrary to conventional processors, which use little endian byte order in majority of cases.

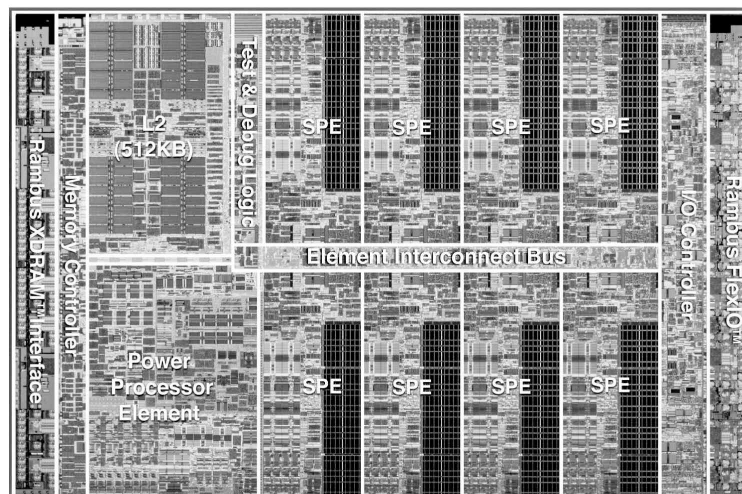


Figure 19: Die photo of the IBM Cell Broadband Engine processor

The Figure 19 shows us, how the PPE and eight SPE elements are placed on the chip. Even though the SPE elements take much less space on the chip than the PPE, they deliver similar computational power.

3 Tools Description

In this chapter tools, techniques and software I used to design, implement and optimize individual program modules as well as build whole applications are described.

3.1 SynDEx

3.1.1 Short Description

SynDEx is a system level CAD (computer-aided design) software based on the algorithm-architecture adequation (AAA) methodology, for rapid prototyping and optimizing the implementation of distributed real-time embedded applications onto multicomponent homogeneous or heterogeneous architectures [9].

This software features rapid prototyping of complex distributed real-time embedded applications based on automatic code generation. It is able to generate code for single processor architecture, as well as for multi-processor homogeneous or heterogeneous architectures, depending on architecture model. Its improved version SynDEx IC is able to generate code for mixed architectures, adding integrated circuit support and field-programmable gate arrays (FPGA) support. It generates safe and highly optimized distributed real-time code thanks to formal verifications and built-in heuristics.

3.1.2 Usage

SynDEx GUI is simple to use. We (as a user) create so called *boxes* which refer to corresponding source code modules and have a certain encapsulated functionality, depending on type. There are 5 types of boxes in SynDEx. Functions, Actuators, Constants, Sensors and Delays. Every box (each reference) is called an operation and has several input and/or output ports. Then we connect these boxes (their ports) with lines, edges. We can only connect an output port with another input port. These determine the data flow. In fact, using the GUI, we create an oriented data-flow graph, that is later treated by SynDEx adequation heuristics. Similarly, we create an architecture graph, where we state operator types and interconnecting media type.

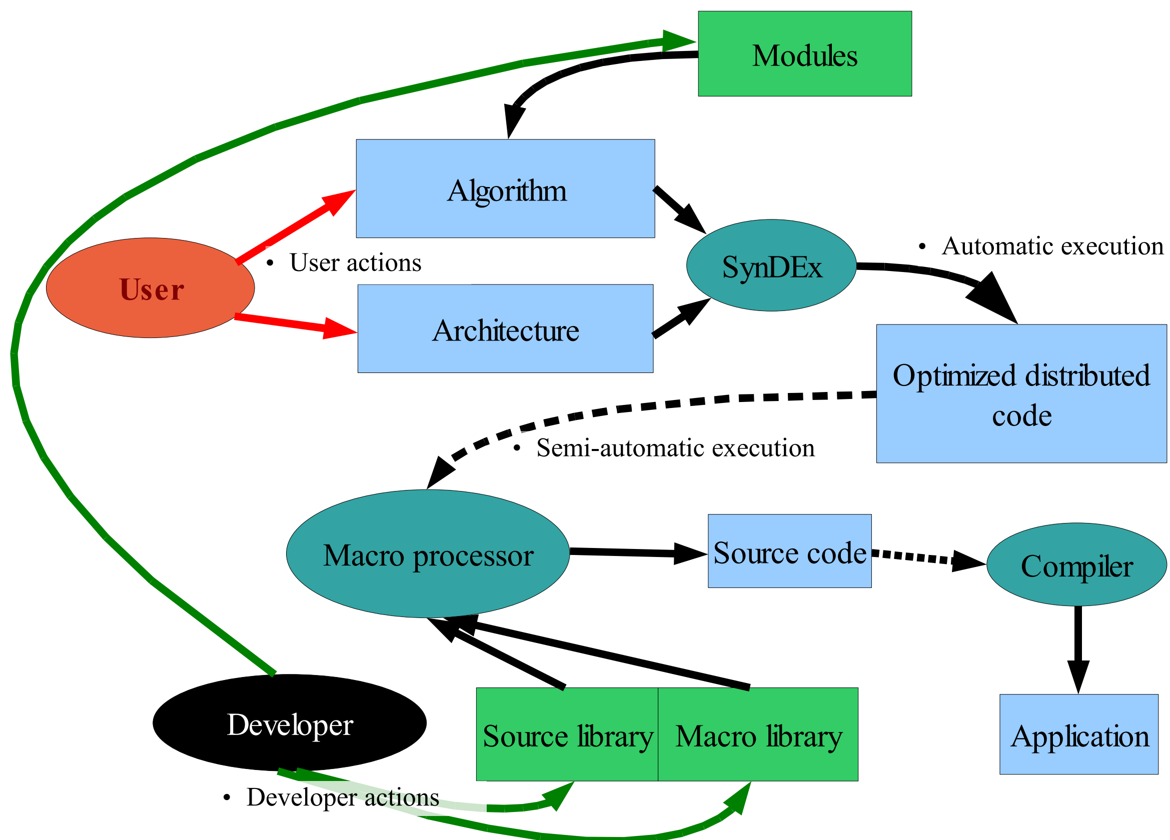


Figure 20: the process of using and extending SynDEx

SynDEx generates intermediate code for every processing unit and communication between them (if there are more of them), which is then processed by GNU M4 macro processor, see chapter 3.2. Basic SynDEx macros are provided with the software, but we have to create module specific macros on our own. The resulting output code is a fully functional code, that can be used even without SynDEx. Finally, we compile the code by desired compiler, depending on preferred output language.

SynDEx is a well documented multi-platform freeware.

3.1.3 Modular Programming

Modular programming is a technique used by developers, that is based on separating the source code into several usually independent modules, connecting them with interfaces (header files).

It is essential for developing SynDEx modules that modular programming is used. This way it is easier to debug modules one after another and the modules once created can be used again. SynDEx actually uses rapid prototyping methods to build applications. Using SynDEx, we (as a user) are able to easily build applications based on modules in a short time.

3.2 M4 Macro Processor

M4 is a GNU version of traditional LINUX macro processor. It is a powerful and versatile universal multi-platform tool [10] for automatic macro handling and text modifying and extracting. It features a great variety of standard and non-standard functions.

To write a functional SynDEx module, we have to write short macros for every module to let the M4 recognize it. We use M4 when building applications with SynDEx, because we need to translate the intermediate code rendered by SynDEx into the target programming language code in order to pass it on to a compiler of the language.

The input of M4 in our case is SynDEx-generated intermediate code, that is processed by M4. The result of this process is target programming language source code for each processing unit including the inter-processor communication, ready to be compiled.

3.3 GNU C and C++ Compilers, GNU Make

These are the standard tools for compiling ***C*** and ***C++*** modules for both Linux and Windows platforms and both little and big endian based machines.

I use ***C*** programming language in this project.

3.4 Extendibility, Modifiability and Modularity

In order to *recycle* an application, it has to be written well. SynDEx modules represent independent source and functional blocks that can be used as ordinary functions even outside SynDEx, while inside the software they can be used (referenced) any time and as many times as needed.

To provide easy way to modify the modules, the communication part of the module and the computational part should be separated. If the interface changes, it is easy to only change the communication part and the rest can remain untouched. In the same way it should be easy to extend the software by retaining the interface and designing new modules built on it.

4 Implementation Methods

To extend SynDEx I had to deal with several major difficulties. These are presented in this chapter along with the solutions and techniques used.

4.1 File and Stream Handling

4.1.1 Endianness

The term may be better known as the *byte order*. It refers to the way a processor (and whole system) deals with bytes stored in memory. Using Big-Endian approach, the most significant *bytes* of a string, number or any other data structure are situated at the lowest addresses and consequently their end is assigned the highest addresses, hence the name Big-Endian. Vice versa, the Little-Endian approach stores the most significant byte of a data element in memory block with the highest address having the end on the lowest addresses, that's why it is called the Little-Endian.

4.1.2 IBM Cell versus Intel Based Machines

First I have to mention that among other image formats only the BMP file handling has been implemented for it's simple format. It is uncompressed and easy to read. Nevertheless, there is no problem in creating another module reading any other image format, providing the interface is the same.

If we need to read an image on both little and big endian systems, we need to byte-wise read the data from a file, as well as store it byte-wise into a file.

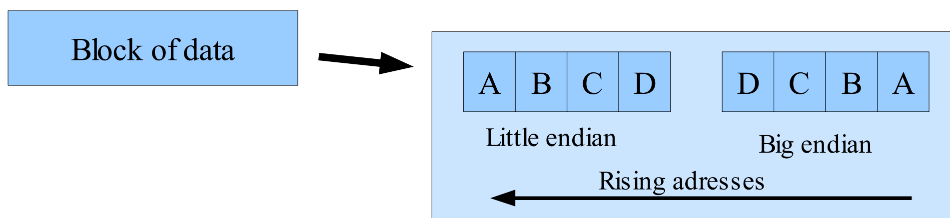


Figure 21: little and big endian 4-byte (double word) variable representation

Intel uses little endian and IBM Cell uses big endian byte order. This makes them incompatible when applications that perform file reading and writing are taken into consideration.

4.1.3 Word and Double-word Splitting and Merging

We solve the problems mentioned above by reading from a file byte after byte and then putting these together using shifting operations. According to the target data type, we read a certain amount of bytes before doing the shifting.

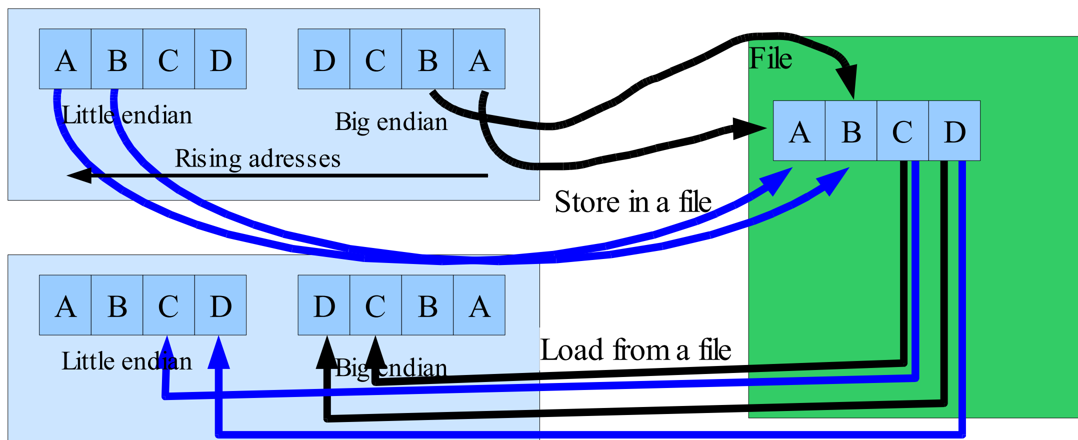


Figure 22: byte-wise storing and reading of a double-word variable

Vice versa, for storing we have to split the data type we are about to print into a file into bytes, only then store them one by one in the order to meet the file format requirements.

```
inline int split2bytes(PTR_PIXEL pContenu, int offset, unsigned int x,
unsigned int factor){
  if(factor>4 || factor==0)return -1;
  for(unsigned int i= 0; i<factor; i++)
    *(pContenu +offset +i)= (x>>(i*8)) &0x000000ff;
  return 0;
}//split2bytes()
```

I provide the source code of the splitting function (used for file writing), and a rip of the code for merging the data read byte after byte from a file:

```
...
  *width= (*(pContenu + BM_IWIDH+1));
  *width<<= sizeof(char)*8;
  *width|= (*(pContenu + BM_IWIDH));
...
```

This way we can be sure that we get the correct byte order regardless of the platform we run the application on.

4.2 The Interface

One of the major problems when working on a bigger project is the definition of a communication interface that would specify the data types used by the functions as well as conventions regarding the inter-function data handling.

4.2.1 Memory Allocation

Image processing usually requires lots of allocated memory, depending on the size of an image.

A relevant question here would be, if we have to optimize the applications for memory usage. In that case we would use only a single buffer for all the processing operations. It would be a fast and lightweight solution – the only thing to do would be correct pointer handling, regarding the function-to-function data transfers. However, if we used that approach, we could not take benefits from parallelism, not to mention heterogeneous architectures.

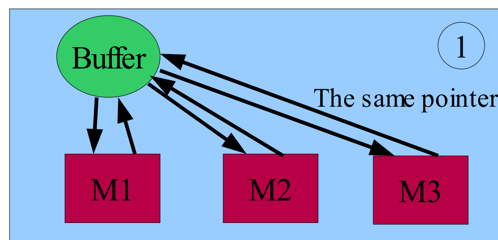


Figure 23: memory saving solution, one buffer for every operation, one pointer used for everything

The second solution would be to use multiple buffers each allocated inside every functional module, letting the modules handle pointers and memory allocation themselves. This way the product of one module would be immediately passed to the following module as a dynamically allocated array, thus creating a dynamic buffer between every two communicating modules. These would then be sequentially written and read. Parallelism would be possible if the writing and reading stage were well separated using mutexes (mutual exclusion). Still this solution is not really suitable for our type of applications where we do not need dynamic memory allocation, because the dimensions of the images we are going to process are known ahead. Dynamic allocation needs extra handling – reallocations, exceptions and extreme cases handling.

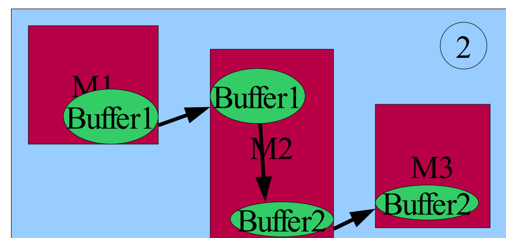


Figure 24: more buffers, one for every operation, robust modules, dynamic memory allocation, overhead

We have come to the stage, where we know the dimensions of the pictures we want to

process, so there is no need for dynamic memory handling. We are going to use static memory allocation and static fields as buffers. For every functional module a static buffer is created. We do not handle anything dynamically and consequently we do not need to check the pointers – we reduce overhead. As a result we have parallelism-capable program modules which still are as lightweight as possible. Moreover, it is very important in real-time applications to do as many static verifications as possible in order to avoid crash of an application.

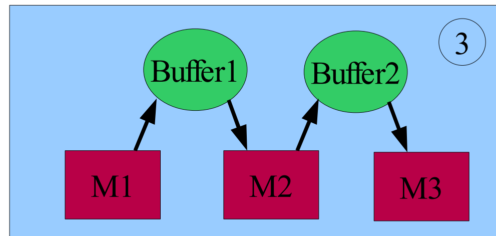


Figure 25: static memory allocation, more buffers, more pointers, but still a lightweight solution

4.2.2 Structures

SynDEx uses special functions to send data between processing units. Even though there are functions able to handle structures, they are far more complicated than the regular ones and the communication primitives have to be rewritten when using them. For this reason we avoid using structures. This way the modules are ready to be used with any homogeneous and heterogeneous architecture models and are not too demanding.

If there are any structures used in the code, we try to unwrap them and use unstructured data types and arrays.

4.2.3 The Interface – Final Decision

We can finally decide upon the interface and memory handling conventions to be used. We know that we are going to use static allocation of main buffers (for processed images) and let SynDEx handle the pointers. We also know that we need to use non-structured data types, scalar variables, and arrays for all function calls and consequently for SynDEx ports.

```
typedef unsigned char              PIXEL;
typedef unsigned int               CPIXEL;
```

These are the basic types used for images as arrays:

```
PIXEL buffer1[BUFF_SIZE];
. . .
function_call(buffer1, buffer2, scalar_variable_x, scalar_variable_y);
```

They are used as shown in the Figure 25.

4.3 SynDEx Modules – The Definitions

Now that we have the interface, it is time to design the so called boxes in the SynDEx software. This happens in three steps:

- first we create a local definition
- then we create input and output ports, name them and assign a data type
- finally, we have to describe behaviour on different kind of processing units

For the SynDEx part, the modules are done.

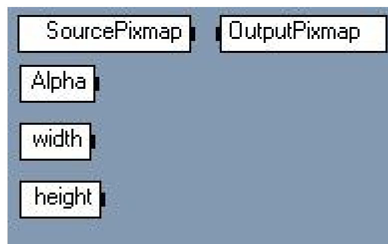


Figure 26: example of a definition in SynDEx. The Deriche filter

4.4 Encapsulation of Algorithms into Modules

We need to create modules for all of the algorithms described in the chapter 2.2 – The Algorithm Chain, which actually means we do it for all the definitions we created in SynDEx. We do this by wrapping the algorithms into functions that are compatible with the interface sketch we presented in chapter 4.2.

In the first part of the functions we try to prepare a suitable environment for the algorithm that follows – we do not use function parameters inside these, so that it is easy to modify the first part (interface communication) without altering the actual algorithm in case we need to change something.

The input and output ports actually represent parameters sent to the modules (functions). We deal with port inter-connecting in the chapter 4.5 below.

4.5 Macro Definitions

As the input of the SynDEx software is only an intermediate code, it has to be processed by the M4 macro processor in order to be prepared for compilation. M4 uses macros defined by developers. A macro is an essential part of a SynDEx module. For every module there has to be one (or more, depending on the definitions we created). Let me introduce a simple example to understand the issue.

4.5.1 Macro Example

In the algorithm definition, we have a SynDEx definition reference (a box) called *save_image_black_white* with 4 input ports (an image, its two dimensions and a colour) and one output port (an output image). Upon SynDEx code generation, the result looks like this:

```
save_image_black_white(buffer, width, height)
```

After being processed by M4, this line of code produces the following string (except possible others):

```
SaveImageBMP("image_name.bmp", threshold_buffer, img_width, img_height,
BLACK_WHITE);
```

This is actually a ***C*** code function call. As you may notice, there are 3 ports for the box, but 5 parameters for the function, three of them carry values previously computed by previous functions. Correct port connections is exactly what is handled by the macro definitions (on the lowest level) and SynDEx (on the level of boxes). A macro for this example would look like this:

```
define(`syndex_save_image_black_white',`save_image_black_white')
define(`save_image_black_white',`ifelse(
    processorType_,processorType_,`ifelse(
          ...
          ...
          MGC,`END',``if( SaveImageBMP( "$0.bmp", $1, $2[0], $3[0],
BLACK_WHITE )==ERROR ) return ERROR;
'')')')
```

Where `BLACK_WHITE` and `ERROR` are ***C*** macros and `$1`, `$2`, `$3` are the parameters passed to the M4 macro, `$0` is the name of the box (`processorType`, `MGC`, `END` are SynDEX specific macros).

There is also a macro called `syndex_save_image_color` calling the same function, but with parameter `COLOR` instead of `BLACK_WHITE`, as the first call stores a black and white image, this one would store a colour image.

NOTE: This is only an **abridged** example showing the usage of two M4 macros calling one C function.

4.5.2 Macros – Part 2

As shown in the example above, macros handle port connections between inputs and outputs, especially the parameter order and correct function calls. Macros also handle correct initialization of the real-time data-flow application, then the computational loop and finally its appropriate termination.

If we want to have a functional SynDEx module, our M4 macros have to be written well. The failure to do so results in malfunctioning code, that sometimes may not even compile.

Now we are ready to encapsulate any algorithm to create modules.

5 Program Components

Particular program components (so far also referred as the SynDEx modules) I managed to design and (or) implement are introduced and briefly described in this chapter.

5.1 The Display Component

The most visible part of my work was to design and implement a flexible SynDEx component that would be able to display data processed in real-time application as well as to be able to display multiple streams in parallel.

I decided for OpenGL libraries because of their great versatility. To handle the OpenGL I chose to use the OpenGL Utility Toolkit (GLUT). GLUT is a cross-platform solution and is free to use.

5.1.1 How It Works

In the initialization phase of SynDEx-generated applications windows are created. The count of windows differs according to the number of *Display* boxes created (referenced) by a user in the SynDEx software using its GUI. These windows are independent and have their own graphical context.

In the computational phase of applications, the windows are updated from within the loop and immediately reflect the stream (image) the user wants to process and observe.

A window is redrawn every time its internal timer schedules to or the framework requests to. There is a possibility not to use the timer and redraw the image after each update instead, but this variant does not work on all machines.

Whole display component runs in separate thread, created using POSIX threads and handled in the initialization phase of applications.

5.1.2 What It Offers

Users have several keys that the display component is sensitive to. Features include:

- the windows can be resized to fit the size of a stream (picture) being displayed, either manually one by one or automatically all at once
- the streams can be paused and resumed again
- a feature to save contents of one or all windows is also available
- immediate response (limited by host machine performance and OS)
- unlimited window count (virtually)

5.2 Two Algorithm Chains

All together 8 modules containing different image processing algorithms. Modules include:

- Garcia Lorca's smoothing and edge detection,
- Canny/Deriche complex edge detection filter,
- automatic double thresholding module,
- contour closing,
- thinning and crest restoration modules,
- and finally the labelling module.

These algorithms are fairly described in chapter 2.2.

Two different image processing chains producing similar output (segmented image) can be created using these modules. However, these can also be interconnected in any way using SynDEx, to create any desired combination and resulting real-time processing application.

5.3 File Handler

Modified file handling functions to enable loading and storing BMP format picture on both little and big endian platforms. Including macros for black and white or colour images.

The module for file saving creates and writes into documents with given name and path, rewrites the file if necessary. The module for file reading is able to read from any existing BMP file.

5.4 IR Camera Stream Reader

I used a very special device to capture a real stream for image processing. It actually was an infra red camera, producing high quality video output. In order to capture the stream, a reader had to be created. It runs in a separate thread. In the first stage it negotiates communication using the RSTP and PTP protocols, then receives the stream using UDP datagrams and stores it into internal buffers.

This module has been created thanks to Jakub Šiška's source code he used in his bachelor's project.

I performed modifications to make it more lightweight and compatible with the interface described in the chapter 4.2 and created a wrapper function for buffer reading.

So far the stream reader has only been implemented for Windows platforms.

5.5 Dilatation, Erosion

The dilatation and erosion modules are based on Eva Dokladalová's and Petr Dokladal's source code. I did the encapsulation as with the segmentation algorithms.

These algorithms are the base for computer vision and shape recognition.

NOTE: more about this interesting algorithm in [11] document, which is about to be released.

6 Conclusion

The purpose of this project was to extend possibilities of SynDEx software focusing on further extendibility and modifiability.

During my research I learned about rapid prototyping methods, I studied about memory allocation and discovered the importance of a well designed interface. I led conversations with researchers working on interesting algorithms and I got in touch with modern and unusual hardware. I was shown usage of real-time applications and also software and hardware development for various purposes. But I did not only study and observe. The point of my research was to design and create.

I designed a dozen of program components as well as an interface that interconnects them, along with many macros and macro definitions. It is a good base for future SynDEx expansion.

The display component, although having a couple of limitations, performs very well and brings a whole new dimension of SynDEx-generated real-time stream processing applications. It also presents a few features that make stream processing more interesting and its viewing more comfortable.

The algorithm modules have no known flows and behave exactly as needed and specified. The system of definitions, easy way of building custom applications, highly precise performance prediction and well behaving heuristics for multicomponent heterogeneous architectures designate SynDEx along with its modules to succeed in the sphere of customizable real-time application rendering.

Future extensions may include other image processing modules and modules for capturing the image or stream from different kinds of imaging devices like webcams, other IR cameras, a universal streamer client, compressed video decoders and many more. They may also introduce module-level parallelism.

This project was elaborated at ESIEE and I think I used well all the resources and devices I was provided with.

7 Bibliography

[1] T. Grandpierre, C. Lavarenne, Y. Sorel: “Optimized Rapid Prototyping for Real-Time Embedded Heterogeneous Multiprocessors”, INRIA, 1999
Available at http://www-rocq.inria.fr/syndex/pub/codes99/codes99.pdf

[2] Cédric ALLENE, Silas BAKARY-GANDO, Julien GRACIA, Zhiyi LEI, Patrick ROUBAUD, Chi Dat TRUONG: “Application à une chaîne de segmentation d'images”, ESIEE school project, ESIEE, 2003

[3] Mohamed AKIL, Nizar ZARKA: “VLSI Optimal Edge Detection Chip: Canny-Deriche Filter”, Tokyo, 1992

[4] J.F. Canny: “Finding Edges and Lines in Images”, technical report no 720, M.I.T., 1983

[5] R. Deriche: “Fast Algorithm for Low Level Vision”, IEEE, 1988

[6] G. Bertrand, M. Couprie: ”Transformations topologiques discretes”, ESIEE, 2008
Available at http://www.esiee.fr/~coupriem/Pdf/chapitre_topo.pdf

[7] M. Couprie, F.N. Bezerra, G. Bertrand: “Topological operators for grayscale image processing”, ESIEE, 2001
Available at http://www.esiee.fr/~info/a2si/gt/GT_biblio.html

[8] Michael Gschwind, H. Peter Hofstee, Brian Flachs, Martin Hopkins, Yukio Watanabe, Takeshi Yamazaki: “Synergistic Processing in Cell’s Multicore Architecture”, IEEE, 2006
Available at http://www.research.ibm.com/people/m/mikeg/papers/2006_ieeemicro.pdf

[9] Yves Sorel: “SynDEx”, WWW document, Internet source, last revision 2009/03/23,
Available at http://www-rocq.inria.fr/syndex/

[10] Rene Seindal, Francois Pinard, Gary V. Vaughan, Eric Blake: “GNU M4, version 1.4.13”, 2009
Available at http://www.gnu.org/software/m4/manual/m4.pdf

[11] Petr Dokladal, Eva Dokladalova: “Dilatation by Flat Rectangles: An Optimal Memory, Zero Latency Algorithm”, ESIEE, not yet published

8 Attachments

- a CD medium with electronic form of this document, sample programs and a simple tutorial for building applications using SynDEx