CACE

A case study in embedded system design

Robert Belleman

CACE

A case study in embedded system design

Robert Belleman
robbel@wins.uva.nl

Universiteit van Amsterdam
faculty of Mathematics, Computer Science, Physics, and Astronomy

Many of the designations used by the manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this thesis, and I was aware of a trademark claim, the designations have been printed in initial caps or all caps.

This documentation was prepared with LaTeX 2_ε. The version you're now holding was last edited on September 23, 1997.

On the cover:
Photograph by Steve Rood (used with permission; thanks Steve!). The overlay shows a raytraced representation of CACE data.

Contents

Voorwoord

Het is werkelijk een verademing te noemen om na het produceren van een document als het deze, eindelijk weer eens iets luchtigs op te mogen schrijven. Maar eindelijk is dan het moment daar dat ik alle mensen mag bedanken voor hun inzet tijdens mijn studie.

In de allereerste plaats wil ik Peter Sloot heel hartelijk bedanken voor alle mogelijkheden die hij mij heeft geboden sinds ik hem leerde kennen bij het Nederlands Kanker Instituut tot en met de meest recente ontwikkelingen op het gebied van de Virtual Reality. Daartussen zit domweg te veel om op te noemen. Tijdens de donkere perioden in de ontwikkeling van CACE waren de gesprekken met hem altijd weer een bron vol inspiratie en motivatie. Na ruim tien jaar snap ik nog steeds niet hoe hij het doet.

Alfons Hoekstra bleek altijd weer in staat mij bij te staan in de kunst van het knoeien aan waterige en licht weerkaatsende proef opstellingen tijdens metingen aan CACE. Daarnaast hebben zijn gedetailleerde op- en aanmerkingen over de inhoud van vroege versies van dit document een grote invloed uitgeoefend op het uiteindelijk resultaat ervan. Alfons; ontzettend bedankt voor al je hulp. Je krijgt een biertje van me.

Samen met Ruud Veldhuizen zal ik heel wat uurtjes achter de *logic analyzer* hebben gezeten om het zoveelste probleem in de hardware te kunnen achterhalen. Dit gebeurde op een gegeven moment zo vaak dat het ding bij mij Pavlov reacties teweeg bracht, maar gelukkig bleek Ruud altijd weer in staat het probleem te achterhalen. Mijn complimenten, en dank.

Ook de mannen van de PSC&S groep waren altijd weer in staat om op mijn meest stompzinnige vragen een intelligent (nou ja) antwoord te geven. Daarnaast bleken zij ook op sociaal gebied geweldige kerels te zijn. Berry, Frank, Benno, Walter, Arjen, Jeroen, Jaap, Jan, Martin, Drona, Arjan en Diederik: allemaal bedankt. Ik hoop dat er nog geweldige tijden gaan komen.

Ook Joep Vesseur, die de groep inmiddels geeft verlaten, wil ik bedanken voor zijn kritische houding op momenten dat ik *weer* een heel stuk software ging herschrijven. Samen met Joep, Gert en Huch bleek zelfs ook nog dat we menig muzakstukje konden produceren. Ik hoop dat we dat binnenkort weer kunnen oppakken.

Ook ben ik veel dank verschuldigd aan Tobias Kuipers, Bas Luttik en in het bijzonder Ramin Monajemi met wie ik gedurende een jaar huis en haard heb mogen delen. Met alle drie heb ik sinds het begin van mijn studie verschrikkelijk leuke momenten mogen meemaken, niet in de laatste plaats in onze stamkroeg, Eik en Linde, waar wij heel wat discussies hebben gevoerd over drank, favoriete TV programma's en studie ervaringen. Ik vind alleen wel dat ze me te veel eer in de voeten wilden

schuiven door te beweren dat CACE door mijn programmeerkunsten een dergelijke vorm van intelligentie verkreeg dat het zichzelf gedeeltelijk vernietigde.

Mijn ouders, Gerard en Nel Belleman, wil ik graag bedanken voor hun interesse en steun tijdens mijn studie, want laat ik duidelijk zijn; zonder hun zou dit alles nooit mogelijk zijn geweest. Ook dank aan mijn broertje en beide zusjes (ik ben nog altijd een paar centimeter langer) en hun aanhang. Jeroen & Joëlle, Saskia & Durk, José & Ron: bedankt voor jullie steun! Ik wil Jeroen graag nog in het bijzonder bedanken voor het oplossen van een hardnekkig probleem in het VME systeem waarmee ik heb gewerkt. En dat via e-mail!

Tenslotte ben ik oneindig veel dank verschuldigd aan Yvon. Sinds het begin tot en met het eind van dit werk heeft zij mijn eindeloos geraaskal vol geduld aangehoord en slaagde ze er altijd weer in om mij op nieuwe, verrassende inzichten te brengen. Tijdens hectische tijden vond ik bij Yvon de broodnodige rust en vooral liefde. Vooral de laatste tijd heeft ze zo vaak voor mij gekookt *en* de afwas gedaan, dat ik vrees dat ik heel wat heb in te halen.

Chapter 1

Introduction

1.1 Embedded systems

Embedded systems differ from general purpose (personal) computer systems in that they are dedicated to perform a specific task [11]. The computer in an embedded system is used as a functional component within a system, not as a computing engine in its own right. The hard- and software in an embedded system are uniquely designed to perform the specific task at hand. This results in an intimate interaction between the two which allows small yet flexible, fast yet cost effective, complex yet reliable systems to be built.

Embedded systems can be found in a widening range of applications, e.g. portable telephones, washing machines, automobiles, airplanes, weapon systems and space exploration vehicles. As microprocessors continue to become smaller, more versatile, and faster at decreasing cost, today's manufacturers find it is often more effective to build embedded systems into their products than custom designed electronic or mechanical devices.

In designing an embedded system, careful attention must be paid to the requirements of such a system [8]. In particular, the following questions need to be answered:

- **What environment will the system be used in?**
  The environment in which embedded systems must operate often impose constraints on the design of both the hardware and software in a system. Especially in cases where a system must operate in hostile environments (consider for example temperature, humidity, vibrations), special attention must be paid on whether each and every component still works under the most extreme expected conditions, down to the smallest detail. Additional limits on size, weight and energy consumption may further complicate the design of the system.

- **What will the interface to the outside world look like?**
  The interfaces that connect an embedded system to the environment under control come in two forms. The first are the sensors which are used to obtain input data on the basis of which the embedded system must fulfill its purpose. Examples of sensors range from simple switches to serial communication links and analog-to-digital converters. The second form of interface are the output devices, also known as actuators, that are used to control external devices. Examples of these range from electronic switches and valves to motors and digital-to-analog converters.

  Furthermore, embedded systems always have some form of interface to the user. These user interfaces need not necessarily be as explicit as the graphical display with keyboard and mouse

that we see with personal computers; the interfaces are often as simple as a couple of buttons or lights. In ideal cases a user may not even be aware of the existence of the embedded system (take for example the fuel injection system in modern cars).

- **What performance is required?**
  A common differentiation in embedded systems with respect to response time is based on the consequences of a "late" response on the correctness of the system as a whole. The most strict form are the "hard real-time" systems which have to respond to externally generated input stimuli within a finite and specifiable delay [67]. Consider for example the automatic pilot in an airplane: the pilot sets a course and altitude from which the automatic pilot (an embedded system) controls the power to the engines and the position of the wingflaps. Based on readings acquired from sensors, the automatic pilot needs to compensate for differences quick enough to ensure a smooth control of the airplane. Any significant delays would result in a very uncomfortable flight for the passengers, or worse.

  Embedded systems where a late response has no consequences on the correctness of the system, are referred to as "on-line" systems.

- **What if it fails?**
  An additional question to consider is how the system should react to failures. Unforeseen failures may be induced by hardware and software but irrespective of the cause and nature, an embedded system designer must carefully asses the consequences of a failure and provide means to recognize such a situation and define how the system should react to it. Nevertheless, no matter how carefully a system is defined, exceptional situations can always occur (indeed, the infamous Murphy's Law states that "everything that *can* go wrong, *will* go wrong").

1.2 About this thesis

Within the Parallel Scientific Computing and Simulation Group, I have worked on the design and implementation of an embedded system for the control of a biophysical experiment known as "Centrifugal Elutriation". In this thesis, I discuss the design considerations in the development of an embedded system, applied to the development of this Computer Assisted Centrifugal Elutriation (CACE) system.

- Chapter 2 provides some basic background information that is required to understand the purpose of the CACE system.

- Chapter 3 provides a detailed overview on the CACE equipment developed by researchers at the department of Immunology at the Netherlands Cancer Institute, including an overview on how it was used during experiments. The experiences acquired during early tests resulted in a list of desired improvements.

  Based on these experiences, an indepth analyses of the requirements for the new CACE-II system, both in hard- and software, are addressed. From this, the choices in computational architecture and the selected programming language are discussed.

- Chapter 4 provides a detailed overview on the computational architecture that has been selected for the CACE-II system, including the choice in programming language that was used to implement the software on this architecture.

- Chapter 5 describes the design of the new CACE-II architecture and goes into detail on the software that has been implemented on this system.
- Chapter 6 describes the automatic focusing subsystem of CACE-II that frees inexperienced users from the intricate details in aligning the optical system.
- Finally, Chapter 7 summarizes the experiences that have been obtained from early experiments with CACE-II thus far and provides recommendations for future work.

Chapter 2

Setting the scene

2.1 Introduction

Human blood roughly consists of plasma, platelets (or thrombocytes), red blood cells (or erythrocytes) and white blood cells (or leukocytes). Of these, the white blood cells form an important part of the human immunological defense system against diseases. Human white blood cells are commonly differentiated into the classes shown in Table 2.1 [26].

class	*subclass*
lymphocytes (30%)	B lymphocytes
	T lymphocytes
	large granular lymphocytes
	natural killer cells
monocytes (5%)	
granulocytes	neutrophiles (60%)
	eosinophiles (4%)
	basophiles (1%)

Table 2.1: The main classes and subclasses in human white blood cells. The percentages shown are relative occurrences in a typical healthy donor.

For clinical purposes, analysts may require accurate knowledge on the composition of these classes in a particular donor, e.g. in cases where malignant leukocytes are suspected, or to measure reduced lymphocyte counts in AIDS patients. Furthermore, in research situations (e.g. drug design), an immunologist may require particular classes of pure leukocytes so that experiments can be performed on isolated types of cells. In that case some method must be devised by which a mixture can be separated into the different classes. For both purposes, some kind of detection equipment is required by which the different classes of leukocytes can be recognized. Since the purified cells may be used in further biological experiments, no modification (e.g. staining) of the cells under separation is allowed. This limits the applicable methods for detection to non-destructive remote sensing techniques.

2.2 Flow Cytometry

One particular method by which different subpopulations in a mixture can be discriminated is a technique called *Flow Cytometry* (FCM) [15]. A flow cytometer (see Figure 2.1) usually consists of a flow chamber in which cells are forced to flow in sequence through a technique called hydrodynamic focusing. A laser beam is focused on the cells such that the cells pass the beam one by one. Each cell causes the light to be scattered in some specific pattern depending on its morphological properties.

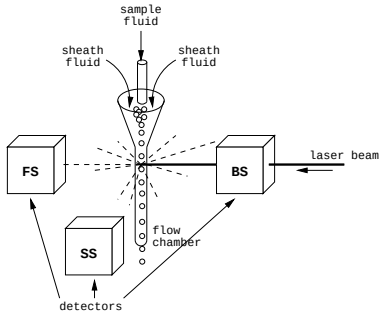

Figure 2.1: Schematic drawing of a flow cytometer (FCM) including forward, side and back scatter detectors (resp. FS, SS and BS).

It has been shown that various subpopulations of particles can be discriminated by measuring the scattered light pulses in two principal directions: Forward Scattering (FS, where the detection angle ϕ relative to the direction of the laser beam is usually $1° < \phi < 3°$) and Side Scattering (SS, usually $65° < \phi < 115°$) [26, 52]. In some cases a third detector is used to allow a broader or more accurate differentiation by the use of a Back Scattering detector (BS, usually $160° < \phi < 180°$) [57]. The signals from these detectors are amplified and often visualized on specialized measuring equipment that create histograms based on the pulse height of the signals. Based on these histograms, expert users are able to discern which kind of leukocytes have passed the flow chamber. In other cases the amplified signals are converted to digital values using *analog-to-digital converters* (ADC) and stored in a computer where they may be subsequently analyzed.

Note that this method for differentiation is radically different from morphological methods in which digital images are taken from the cells under investigation that are then analyzed using image processing techniques (see [3] for an example). The combination of a flow chamber and light scattering equipment allows rapid cell identification for great numbers of cells, while image processing techniques are often very time consuming and appropriate only for the identification of cells in small batches.

2.3 Centrifugal Elutriation

A flow cytometer allows an immunologist to discriminate different subpopulations in a mixture, provided the light scattering properties of the subpopulations are dissimilar. A technique that allows the different subpopulations to be *separated* has been improved at the Netherlands Cancer Institute and is known as *Centrifugal Elutriation* [18, 19]. Centrifugal elutriation (henceforth referred to as CE) exploits differences in sedimentation velocity of human peripheral blood cells to isolate various classes of cells from inhomogeneous cell populations.

CE is based on the following principle: a fluid containing the mixture of interest is pumped into the center of a centrifuge rotor through a rotating seal. The fluid flows through internal tubing

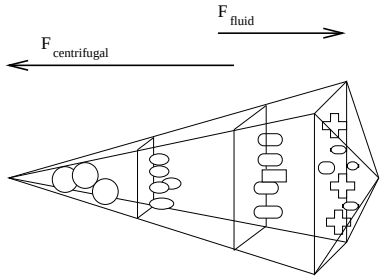

Figure 2.2: Schematic drawing of a separation chamber applied in centrifugal elutriation (CE).

into a divergent separation chamber situated inside the rotor (see Figure 2.2). At the end of this chamber, the flow reconverges into a small tube and leaves the system back through the rotating seal. Two forces act on the cells in the separation chamber: the centrifugal force caused by the rotation of the rotor and the force caused by the fluid flow through the chamber. When these forces are in balance, the special shape of the separation chamber causes the different classes of cells to reposition to an equilibrium state based on their differences in sedimentation velocity. By slowly decreasing the spin rate of the centrifuge, the centrifugal force decreases while the fluid flow remains constant so that the different subpopulations wash out in consecutive batches.

2.4 Computer assisted cell separation

The combination of a centrifugal elutriator and a flow cytometer allows different cell populations in a mixture to be separated from each other (see Figure 2.3). Here the CE equipment is used for the actual separation of the different subpopulations in a mixture while the FCM equipment is used for the identification of the cells that exit the elutriator [55]. A major problem with the use of such a system is that accurate cell separation requires specialized knowledge on both the FCM and CE equipment that can not always be expected from users in a clinical environment.

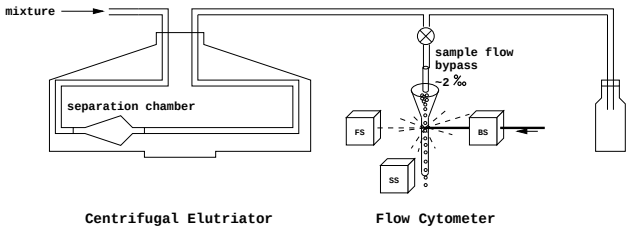

Figure 2.3: Schematic drawing of a coupled CE and FCM system.

In order to facilitate a cell separation experiment, a system has been developed at the Netherlands Cancer Institute consisting of a three-parameter flow cytometer and an embedded computer system, interfaced to a centrifugal elutriator [56, 59]. The objective of this so called Computer Assisted Centrifugal Elutriation (CACE) system is to allow the control of a cell separation experiment in such a way that optimal cell separation can be achieved by non-experts. Early experiments with the CACE system showed that it is capable of controlling a cell separation experiment. As expected however, these experiments also showed that improvements are required before the CACE system can be used in a non-expert environment.

In the following chapter, a historical perspective is given on the design and implementation

of the original CACE system, including an overview on the experiences that were obtained with this system. These experiences form the basis on which the requirements for an improved CACE-II system are formulated.

Chapter 3

CACE: design considerations

3.1 Historical perspective

In an effort to control a cell separation experiment, a three-parameter light scattering device has been designed and built at the Netherlands Cancer Institute that simultaneously measures Forward Scattering (FS), Side Scattering (SS) and Back Scattering (BS) parameters, interfaced to a CE system [56, 59] (see also Figure 2.3). It has been shown that it is possible with these three parameters to discriminate between various subpopulations present in a mixture [52, 62].

In this so called Computer Assisted Centrifugal Elutriation (CACE) system[1], a dedicated data acquisition module is integrated with on-line monitoring primitives and off-line software to analyze the experimental multivariate data, including the detection of the various subpopulations that are present in a mixture. This allows for controlled centrifugal elutriation and facilitates optimal cell separation where previously specialized knowledge and time consuming cell identification techniques were required. Figure 3.1 shows a schematic representation of the components in the original CACE system.

3.1.1 Data acquisition

Data acquisition in CACE is achieved by means of a special purpose module called the Direct Memory Access (DMA) interface. The analog signals from the three light scattering detectors are each amplified and then converted to 6 bit digital values by analog-to-digital converters (ADCs) thus providing $2^6 = 64$ possible values. The DMA interface uses these three 6 bit digital values as an 18 bit pattern to address $2^{18} = 262,144$ memory elements that are each 16 bits wide. In total, this results in an addressed memory area of 512 kilobytes.

By means of a read-modify-write (RMW) scheme, the DMA interface increments the contents of the 16 bit memory element corresponding with each pattern occurrence (up to a maximum of $2^{16} - 1$) thus creating a three-dimensional histogram of data in memory. This histogram represents the light scattering information of all particles that passed the laser beam while moving through the flow chamber. When this memory area is thought of as a three-dimensional box of size $64 \times 64 \times 64$ such a histogram can be visualized as shown in Figure 3.2.

This acquisition technique is different from regular "list mode" methods where the acquired data is often stored as separate items for each occurring event [15]. The obvious advantage of the technique used here is that it requires a constant amount of memory while all information that is required for parametric and non-parametric analysis of the data remains available. In addition,

[1] USA Patent 4939081 (1990).

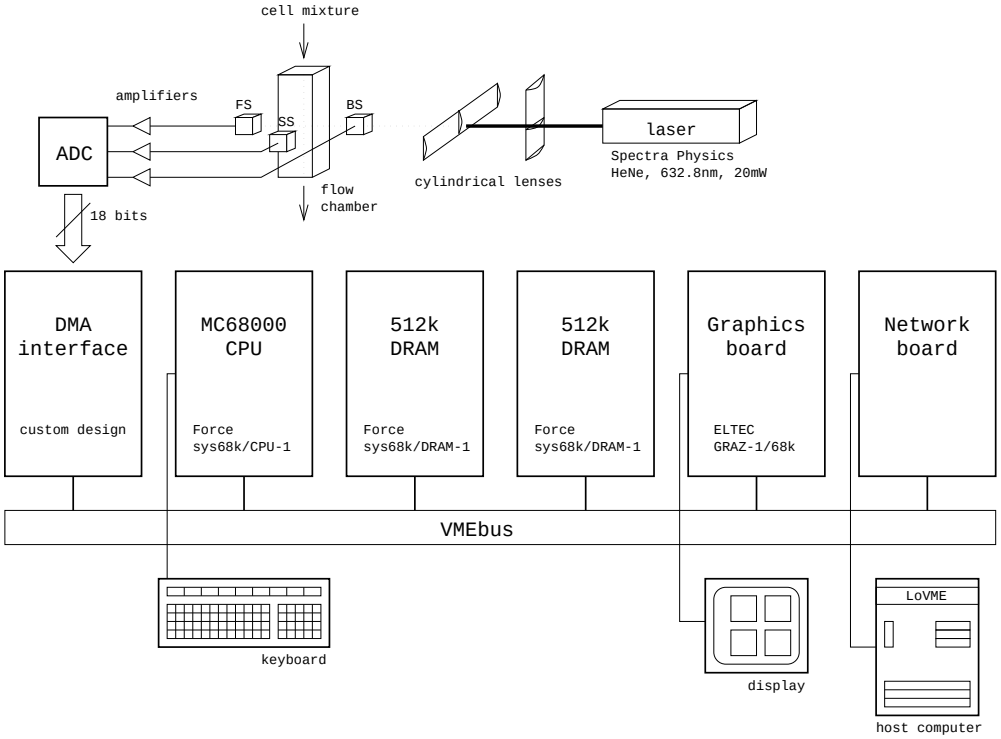

Figure 3.1: Schematic representation of the original CACE equipment.

list mode data acquisition usually requires active participation of the central data processing unit to gather data from the ADCs and store them in memory, at the same time preventing memory overrun. The processing cycles used by such a process prevent the processing unit from performing data analyses operations. The DMA interface however functions autonomously from the rest of the system, allowing the central data processing unit to be used for analyses of the data. However, a disadvantage of this method compared to list mode acquisition is that an increase in resolution of the digital values obtained from the ADCs requires a rigorous redesign of the DMA interface. Furthermore, list mode data allows a more detailed analyses, as all information on each individual cell remains available [46, 53].

3.1.2 On-line monitoring primitives

A special purpose stand-alone operating system was implemented to control a separation experiment, including data acquisition, data manipulation, graphical data representation and a user interface. Using *Modula II* as its programming language, the operating system offers low-level device access, event handling, and concurrent process scheduling [66].

The graphical data representation routines continuously provide the analyst with multiple simultaneous scatter and contour plots of the data buildup in memory. In addition, a "gating" function allows subregions of particular interest to be specified within the memory area as a whole, providing on-line quantitative information on the number of cells contained in a subregion.

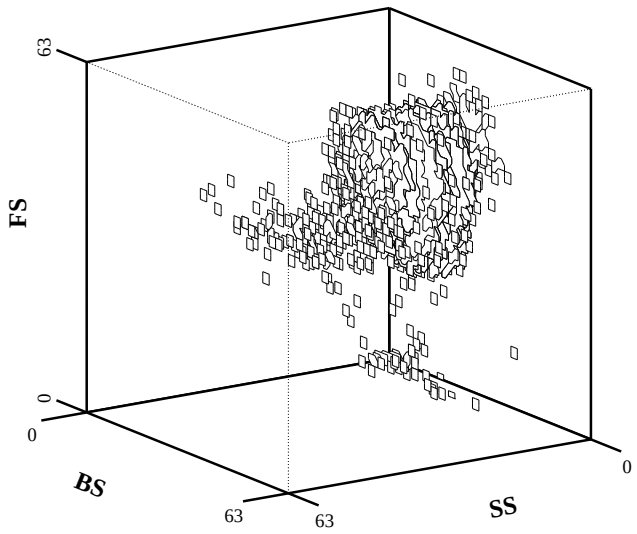

Figure 3.2: Visualization of data acquired during a typical separation experiment. All (FS,SS,BS) points with contents ≥ 2 are shown.

3.1.3 Off-line data analysis

A network connection provides an interface by which the acquired data can be dumped from the CACE system to a host computer. Special purpose off-line statistical analysis software has been developed for this host computer to analyze the acquired data using non-interactive methods [59].

It is possible to discriminate between various subpopulations using the acquisition system just described [57]. Moreover, it has been shown that the subpopulations can be parametrically described by multivariate normal distributions [55]. It is the objective of the CACE system to discriminate these different distributions from the acquired data. In order to achieve this, the acquired data is analyzed through the following data analysis steps:

Noise elimination

Due to inevitable instrumental noise introduced by the scatter detectors and the amplifiers, and stochastic noise caused by the conversion of analog signals to digital values, the acquired data contains a considerable amount of noise that needs to be removed before any subpopulations can be identified. A three-dimensional convolution filter has been developed that eliminates this noise from the data while preserving the information content of the raw data [58].

Subpopulation detection

Since it is assumed that the density functions that describe the individual subpopulations in a mixture can be described by multivariate normal distributions, parametric analysis of the acquired data is justified. A two phased approach was chosen; the first phase determines the number of subpopulations and the initial estimates of the multivariate parameters. A second phase takes these estimates as the initial parameters for a statistical optimization algorithm known as *Expectation Maximization* (EM). With this technique, the *Maximum Likelihood* (ML) of the parameters that describe the multivariate normal distributions is iteratively optimized [59].

3.2 Outline of a CACE experiment

Figure 3.3 illustrates a typical sequence of actions that need to be performed during a CACE separation experiment. First, the CE and FCM equipment are prepared. The flow rate through the CE equipment is adjusted to a constant 18 ml/min and the separation chamber inside the centrifuge is loaded with the mixture of interest. The rotor speed of the centrifuge is adjusted so that the first cell fractions assemble close to the separation plane in the separation chamber. The sheath flow rate through the FCM equipment is adjusted to a constant 6 ml/min and the cylindrical lenses and flow chamber are aligned so that the flow chamber is in the path of the laser beam and light scattering signals from passing cells can be measured. These preparations before an experiment take approximately 5 minutes but little harm is done if it takes longer.

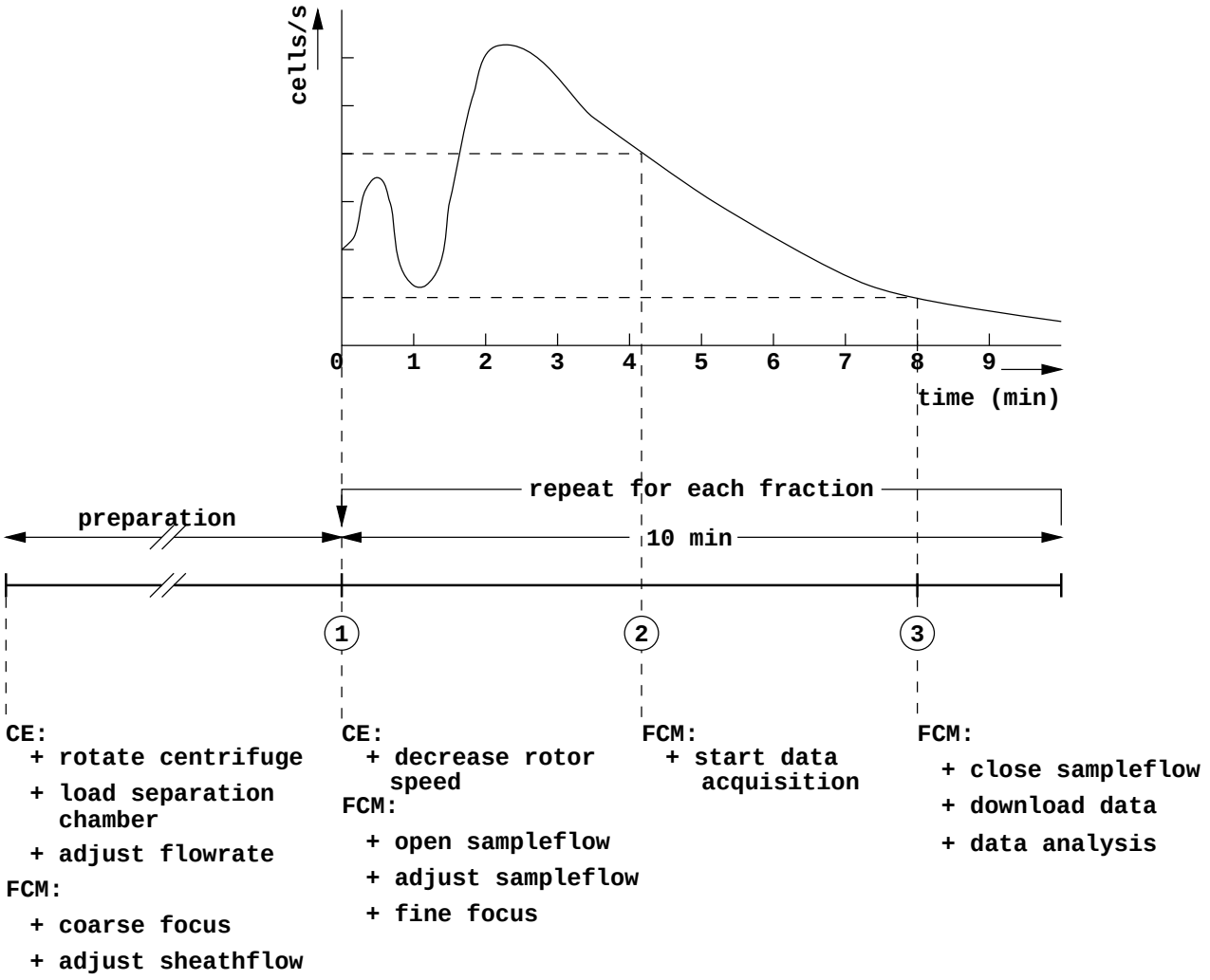

Figure 3.3: Time line of a typical CACE experiment (based on a Figure in [55, page 51]).

For each fraction that is to be elutriated from the mixture, the following actions are repeated (the numbers below refer to those encircled in figure 3.3):

1. The rotor speed is decreased and the sample flow bypass is opened so that the next fraction washes out of the rotor and in part through the detection equipment ($\pm$ 2 $^{o}/_{oo}$). Initially, there

will be some residual cells washing out of the rotor from the previously elutriated fraction. After approximately 2 minutes, a new equilibrium is established and the new fraction can be elutriated. Next, the sample flow rate must be adjusted so that not too many cells flow through the detection equipment. Once this is done, the optical system must be fine focused so that proper light scattering signals from passing cells can be measured. As the trajectory of the cells through the flow chamber may change each time the flow rate is altered, this focus adjustment needs to be repeated for each elutriated fraction [15]. All this should be done as quickly as possible or too many cells will have left the separation chamber, making optimal separation infeasible.

2. Once the sample flow rate and focus are in order, data acquisition may commence when a few hundred cells per second flow through the detection equipment. Data acquisition continues until the cell rate descends to tens of cells per second. Normally, this occurs 8 to 10 minutes after the sample flow bypass was opened.

3. The sample flow bypass is closed to prevent the non-sterile sheath flow from flowing back into the separation chamber. Next, the acquired data is downloaded to the off-line host computer where the data may be subsequently analyzed. Downloading takes approximately 1 to 2 minutes using a simple data compression technique. Analysis of the data on the off-line host computer in most cases takes place after the experiment has finished.

3.2.1 Experiences

Experiments with the CACE system have shown that it is capable of controlling a separation experiment. The on-line graphical representation of the data buildup during a separation experiment proved to be of great value to analysts that previously had to resort to standards and personal experience. In addition, the off-line data analysis software allowed non-interactive parametric analysis of the experiment, providing qualitative and quantitative information after a separation experiment. On the other hand, these experiments also revealed some issues of improvement:

- The overall use of the CACE system would significantly increase if quantitative information could be obtained on-line, *during* an experiment instead of afterwards, downloading the acquired data to an off-line system and analyzing it there.

- Depending on the number of studied parameters and the number of subpopulations in a mixture, analysis of the acquired data on the off-line system could take anything from a couple of minutes to many hours [2]. This makes off-line analysis during an experiment unviable. Indeed, the requirement of an additional computer system for off-line data analysis can be considered cumbersome if the goal is to achieve an embedded system for the control of a separation experiment.

- Most data analysis programs are limited to two dimensions; in order to increase flexibility towards the number of studied parameters, the analysis programs would have to be uplifted to their three dimensional counterparts.

- Accurate data acquisition depends on a proper alignment of the optical system (in particular the flow chamber and the lens system). Manual focusing of the optics requires expertise that

[2] On a LoVME 68010 (Microproject BV, the Netherlands) with floating point unit, running the Unix operating system.

often can not be expected from a typical user of the system. Moreover, focusing the optics may not take too much time. To facilitate routine application, an automatic focus subsystem must be added to the CACE system.

- Autonomous computer assisted cell elutriation can only be achieved if additional logic is integrated into the system that is capable of controlled regulation of the complete flow system, including the rotor speed.

- The inherent complexity of this type of expert systems used in a non-expert environment requires a structured extendable design and a well considered user interface.

From this, it was concluded that the overall use of the CACE system would benefit greatly if it could be improved on the above mentioned points.

3.3 New requirements

From the overview that has been presented on the CACE system and the experiences obtained during experiments, described in the previous section, it becomes clear that the new requirements impose strong demands on the design of both the hardware and software in such a system. The need for on-line parametric analyses of the light scattering data obtained from the FCM equipment in particular implies that an increase in computational power in the updated system is obligatory. Furthermore, the additional requirements to control the optical system, the rotor, and the flow system add substantial demands on the choice of new hard- and software.

Based on the overview of actions performed during an experiment with the original CACE system, the requirements in terms of functionality and time constraints can be identified. From that, the required changes in both hard- and software are discussed, including the choice in programming languages.

3.3.1 The computational architecture

As explained in section 3.2, the elutriation of each fraction from the separation chamber takes approximately 10 minutes. During this time, the analyst should be provided with frequent updates on the progress of the separation experiment to ensure that optimal cell separation can be achieved. Experiences with the CACE system showed that an updated display once every 30 to 60 seconds during the elutriation of each fraction provides the analyst with sufficient information to achieve this. The information provided to the analyst should, at the least, consist of a graphical representation of the data buildup in memory.

Furthermore, a parametric representation of the subpopulations detected by the data analyses software allows the analyst to obtain valuable quantitative and qualitative information on the elutriated fractions and the relative occurrence of these fractions in the mixture as a whole. In the original CACE system, the acquired data was transferred to the off-line host computer (which could take 1 to 2 minutes) where it was then analyzed (which could take many minutes to hours). Given the objective of attaining a parameterized representation of the data within less than a minute, it is clear that the computational hardware should be closely coupled to the acquisition system and must be of sufficient strength to complete data analyses within this time. Moreover, the upgrade of the data analyses programs from two to three dimensions imposes an additional demand on the required computational power which must be taken into consideration.

Parallel processing

Provided a problem can be decomposed into smaller disjoint subproblems, a parallel computing platform should, ideally, allow problems to be solved faster than on a sequential computing platform. The decomposition method of a problem has a direct relation with the algorithmic and data properties of the underlying algorithms, thus the most common methods are known as algorithmic (or functional) and data decomposition.

A point of careful consideration before parallelizing a program however, should always be the anticipated efficiency of the resulting program. Efficiency is defined as the ratio of the obtained speedup of a program to the number of processors employed. Speedup is defined as the ratio of the program's execution time on a uniprocessor system to the execution time on a multiprocessor system [29].

Once a suitable decomposition of a problem into disjoint subproblems has been found, the subproblems can be computed by subprocesses on separate processors. Communication between these subprocesses either takes place using a shared common memory or via a communication network using message passing. While an advantage of shared memory communication over message passing systems is that the copying of memory structures can often be avoided, it should be noted that the access bandwidth of memory on most architectures is limited by the bus that interfaces the processors to the shared memory. This bandwidth limitation can have a severe impact on the scalability of a program when more processors are employed.

In message passing systems, the amount of communication between processes and the delay caused by the communication (the so-called *latency*) can have a severe impact on the efficiency of a program. Load balancing techniques, either static (before run-time) or dynamic (during run-time), can aid in minimizing the effects of this communication. However; communication overhead should always be minimized as much as possible.

Architecture requirements

Since the data analysis programs have an inherent parallelism that can be exploited when implemented on a parallel architecture, it was decided to substitute CACE's processing unit (a Motorola 68000) for an architecture by which a programmer can express this parallelism. In choosing this new architecture, careful consideration must be taken to ensure that the custom designed DMA interface can still be used. This means that the architecture must interface to the VMEbus for which the DMA interface was designed. One particular issue in this respect is that the new architecture should be capable of taking over the role of VME*bus arbiter* from the substituted processing unit. This device arbitrates accesses to the VMEbus made by so called *masters* (such as the DMA interface) to prevent simultaneous accesses to the bus and to schedule requests from multiple masters for optimum resource use [1]). Furthermore, the new architecture should be scalable so that any future computational demands can be met by adding more processors to the system. As the VMEbus bandwidth is limited to 7 Mb/s [1], the architecture's communication model should be based on a separate, message passing communication network.

Programming language requirements

Beside the hardware requirements, careful consideration should also be taken in the choice of programming language that is used to actually implement the software on this architecture. Given the discussed requirements for this particular embedded system so far, the following issues have to be considered [11, 54]:

- Multitasking support and scalable multiprocessor support – Modula II, the programming language used in the original CACE system provides constructs for expressing concurrency, but not parallelism. As the new architecture will consist of multiple processing units, a programming language must be selected that allows a system designer to express parallelism in his algorithms.

- Structured design – Software on an embedded rarely consists of one single omniscient program; such "monolithic programs" would be hard to understand and maintain. Instead, the programming language should provide constructs by which the task at hand can be decomposed into simpler subtasks that can be implemented relatively independently from each other.

- Maintainability – A programming language should help in writing programs that are easy to read, easy to understand, and easy to modify.

- Error checking and fault handling – The language should protect the programmer from making common programming mistakes, such as the incorrect use of typed variables. The language should be able to detect these faults and provide methods to handle them.

- Low-level device handling – The programming language must contain provisions to directly access the hardware devices that are under the control of the embedded system. Closely related to this is the programming language's handling of interrupts that are generated by these devices in situations of urgency.

- Efficiency – The compiler that translates the programs to the architecture's native object code should be able to produce efficient code. Some programming languages allow a programmer to include assembly code into a high level program to compensate for inefficiencies in a compiler. As assembly code is hard to produce and, quite frankly, impossible to maintain, this should be avoided at all cost.

- Availability and portability – A trivial question to consider, but important nevertheless; what programming languages are available for the selected architecture? The vice versa question however, is far less trivial for reasons of portability; for what architectures is the language available? As the architectures we use today may not be a suitable choice for the future, it is a wise decision to choose a programming language that will still be in use in the future so that a redesign of the software can be avoided.

It should be noted at this time that it is not the aim of this work to implement a "hard real-time system" which has to respond to externally generated input stimuli within a finite and specifiable delay [67]. In this case, the objective is to design and implement an on-line system that allows for rapid data analysis while controlling a separation experiment. However, some of the techniques applied in real-time programming languages and systems can be employed here.

Transputers and `occam`

Very few alternatives were available that could meet the above requirements at the time this decision had to be taken (1989), but a satisfactory solution was found nevertheless: The computational architecture that has been selected for the embedded architecture consists of multiple VMEbus based *transputer* modules. Transputers are versatile microprocessors that can easily be coupled to form a parallel computing platform. The selected programming language is `occam`, which provides

a powerful methodology to implement parallel software on a parallel architecture, the transputer in particular. Chapter 4 discusses both the transputer and the `occam` programming language in detail.

3.3.2 Automatic alignment and focusing of the optical system

As discussed in section 3.2, the cylindrical lenses and the flow chamber must be properly aligned before data acquisition may commence. This alignment must be repeated for each elutriated fraction as the trajectory of the cells through the flow chamber changes each time the flow rate through the chamber is altered. In the original CACE system, the proper location of the cylindrical lenses and the flow chamber was obtained through visual inspection of the FS detector signals generated by passing cells. As this required that cells exit the elutriator, the correct focuspoint had to be found quickly, i.e. within approximately two minutes, or the cell fraction would be lost. Furthermore, visual inspection of the FS detector signals requires expertise that can not be expected from a typical user of the system. It was therefore decided to design and build a focus subsystem that could achieve this automatically.

To allow the cylindrical lenses and the flow chamber to be positioned under software control, they have each been mounted on a servo controlled table. A VMEbus based servo controller (based on the Motorola 68010 processor) has been added to control these servos. In addition, a special purpose VMEbus card has been designed and built to amplify and sample signals from the FS detector. Based on these signals, the *autofocus* software automatically focuses the laser beam onto the particles that flow through the channel of the flow chamber.

The autofocus software runs as a separate subsystem on the servo controller board which allows this system to operate autonomously from the CACE-II system. Communication from this system to CACE-II takes place via the VMEbus. For the implementation of the software, the C programming language and 68000 assembly language were used. The unfortunate use of assembly language was necessary in order to access the on-board routines provided by the servo controller board. Chapter 6 describes the autofocus subsystem in detail.

3.3.3 Automatic control of the flow system

Two flow paths are important for the correct operation of the CACE system (also recall Figure 2.1 on page 6): The sheath flow rate through the flow chamber should be adjusted to a constant 6 ml/min. The sample flow rate is adjusted so that approximately 200 cells per second pass through the flow chamber. The exact sample flow rate depends on the number of cells that elutriate from the CE system per second, but experience has shown that the sample flow rate generally lies between 0.05 to 0.1 ml/min. The special construction of the flow chamber and the order of magnitude difference in the sheath and sample flow rates cause the cells in the sample fluid to be pulled through the center of the flow chamber, without touching the edges (this technique is called hydrodynamic focusing).

To automatically control the flow system, a device is required that is able to measure flow rates while another device adjusts the flow rate. At the department of Biophysics at the Netherlands Cancer Institute, some research has been done on the design of a device to measure flow rates through Laser Doppler methods. Unfortunately, this project has recently stopped. In Chapter 7 I will briefly touch on this subject again and provide some ideas that may compensate for this omission, including the control of the rotor speed.

3.3.4 The user interface

In the original CACE system, the interface to the user was implemented through a VME based graphics board that was controlled by the operating system running on the central processing unit. Although this interface worked quite well, a change in the interface required recompilation of the complete software for the embedded system.

Over the last years, programming environments have become available that allow powerful, flexible and portable user interfaces to be built relatively quickly [48, 65]. In Chapter 5 a brief description will be given on the user interface that has been developed for the CACE-II system.

Chapter 4

Building blocks of the embedded architecture

4.1 Introduction

In the previous chapter, the design requirements for the new embedded architecture were discussed from which the building blocks were selected. In this chapter, a detailed discussion will be provided on these building blocks and their use in implementing the CACE-II embedded architecture.

4.2 The Transputer

In 1984 a company named INMOS[1] launched a new microprocessor which became known as the *transputer* (derived from *trans*mitter and com*puter*). The most innovating aspects about this processor[2] at its time of introduction was that besides its 32 bit architecture and fast on-chip static RAM it also featured hardware multitasking support for the execution of concurrent tasks (capable of switching processes in less than a microsecond) and it had four independent bidirectional serial communication interfaces ("links") for the communication with other transputers or external devices. These communication links allow engineers to construct parallel computing systems containing many interconnected transputers. Several families of transputers have been produced ranging from the T2 series to the T9, each with slightly different characteristics. Table 4.1 shows an overview of these families.

family	*CPU*	*FPU*	*SRAM*	*MIPS*	*MFLOPS*	*link speed*
T2	16 bit	no	2/4 kb	30	0.5	5/10/20 Mbit/s
T4	32 bit	no	2/4 kb	25	0.5	5/10/20 Mbit/s
T8	32 bit	yes	4 kb	25	3.6	5/10/20 Mbit/s
T9	32 bit	yes	16 kb	120	8	160 Mbit/s

Table 4.1: Some of the characteristics of the four transputer families [31]. Both MIPS and MFLOPS are *peak* performance figures.

By far the most popular transputer is the IMS T800 and its later derivative, the IMS T805. Its popularity is mostly due to the fact that this transputer was the first with an on-chip 64-bit floating

[1] INMOS is now a member of the SGS-THOMSON Microelectronics group.
[2] The words *transputer* and *processor* are used promiscuously throughout this chapter.

point unit capable of a sustained performance of approximately 0.9 MFLOPS (million floating point operations per second) [50] allowing it to be used in serious number crunching applications. Also, it is worth mentioning that the IMS T800 was probably the first processor that was proven to be correct through formal design methods [44]. A block diagram of this processor is shown in Figure 4.1.

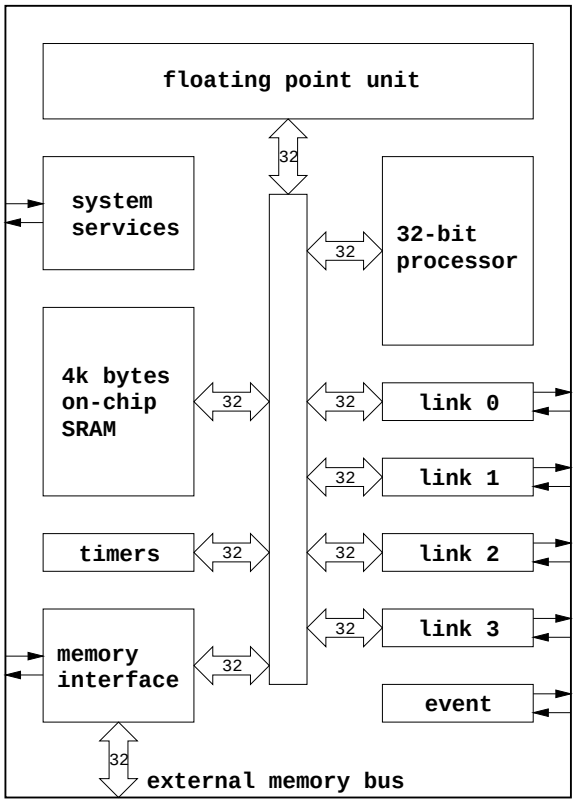

Figure 4.1: Block diagram of the IMS T800 Transputer.

The transputer's communication links allow multiple processors to be coupled to form a parallel computing platform. Only 8 wires are required to connect two processors so that compact parallel systems can be built quite easily. Many manufacturers have taken this processor as the basic building block to construct *Massively Parallel Processing* (MPP) platforms, some of which contained in excess of 1,000 processors. Much of the software development for CACE-II was done on a *Meiko CS1* system containing 64 IMS T800 transputers.

The communication links can only be used for point-to-point communication between two processors. The larger transputer systems often contain *crossbar switches* that allow the network topology to be configured by software prior to loading a program.

4.2.1 Reduced Instruction Set Computing

The transputer is essentially a *Reduced Instruction Set* (RISC) processor. Each instruction is one byte (8 bits) long and is divided into two 4 bit parts. The most significant 4 bits of a byte represent a function code, the least significant bits a data value [39]. This representation allows sixteen basic instructions, each with a data value ranging from 0 to 15. Some function codes represent prefixes

for building larger instructions, allowing for a total of approximately 150 instructions (including floating point instructions).

The transputer contains six registers, each one word long. Three of these (the `Areg`, `Breg`, and `Creg` registers) are used as data registers and implement a three level evaluation stack. Each loaded variable or constant is pushed onto this stack. Every time an operation is performed, the two values at the top of the stack are popped and used as operands for the operation, and the result is pushed back onto the stack. Operating with a data stack removes the need to add extra bits to an instruction to specify which register is accessed. As a result, instructions can be packed in smaller words so that program code fits tighter in memory, and less time is spent fetching the instructions from memory. The remaining three registers in the transputer consist of the program counter, the stack pointer and an operand register.

4.2.2 Programming transputers

The key concept in programming transputers are *Communicating Sequential Processes* (CSP, first introduced by Tony Hoare who was partly inspired by Dijkstra in [14]) [24]. The basic idea in CSP is that multiple concurrent (or parallel) processes synchronize by communicating with each other: a process A states that it is ready to send a message to a specific process B, and process B states that it is ready to receive a message from process A specifically. If either process starts synchronizing before the other, it is put on hold until *both* are ready. After communication, both processes continue their execution independently.

Conceptually, CSP is quite simple and yet it provides a good solution to many of the more common synchronization problems (such as the *Dining Philosophers* and *Producer-Consumer* problems). But it has some drawbacks. For example, the requirement of CSP that two communicating processes need to be explicitly named and the lack of message buffering so that communicating processes always block. The latter shortcoming can however often be solved by the programmer. Of special importance for the design of hard real-time systems is that CSP offers no temporal provisions that allow timing constraints to be specified. Extensions to CSP have been proposed in the form of *Timed CSP* [51], but it is unlikely that these extensions will be incorporated into future designs of the transputer.

Concurrency

The transputer includes hardware multitasking support for the execution of multiple concurrent tasks on one processor. Multitasking allows a process to execute when others on the same processor are idle. This allows a processor to be used more efficiently when, for example, a process is waiting for a delayed event to take place. The maximum number of concurrent processes on one transputer is bounded only by available memory.

The transputer's scheduler grants time slices of approximately 1 ms to all active processes in a round-robin fashion. Context switches occur in less than 1 μs. When a task eventually terminates it is removed from the list of processes. At any time, new tasks may be created and added to this list.

Process communication

The transputer essentially has two instructions that allow processes to communicate: *in* and *out*. Both take three arguments: the memory address to communicate at, the address of a message buffer and the message length. There are four "special" memory addresses that allow processes

to send messages over one of the external communication links to processes on other transputers and, likewise, four addresses that can be used to receive messages from the external links. The external communication links are DMA driven so that the transputer can continue the execution of concurrent processes while communication takes place.

The one-way path from one process to another is commonly referred to as a *channel*. Any location in memory can function as a channel thereby allowing a process to communicate with a process running concurrently on the same transputer in exactly the same way as with a process running in parallel on another. This allows programs consisting of multiple processes to execute on one processor in the same way as when these processes are distributed over multiple processors (see Figure 4.2).

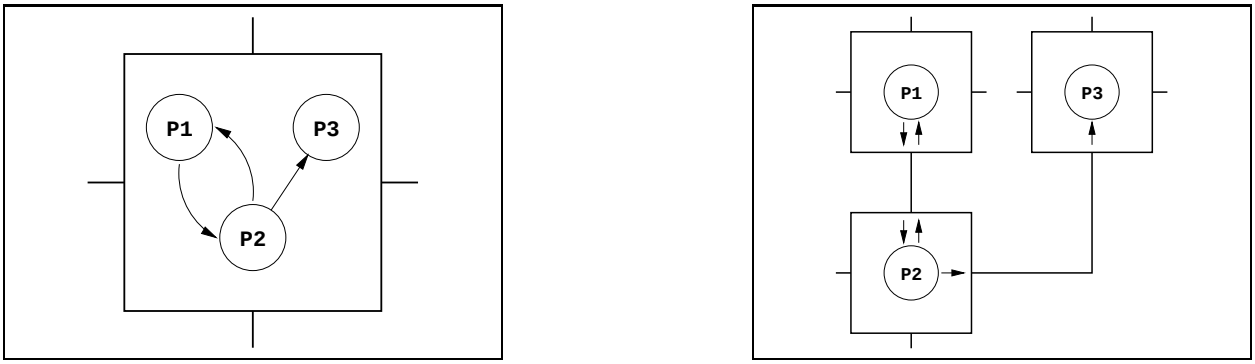

Figure 4.2: On the left: three processes running concurrently on the same processor. On the right: the same processes running in parallel, each on a separate processor.

Parallel processing

Concurrency is the lowest level of parallelism that is offered by the transputer. Using the transputer's high speed links, multiple processors may be coupled together allowing the execution of parallel message passing subprocesses.

Once a suitable decomposition for an algorithm is found, the disjoint subprocesses must be mapped onto the target architecture. In a dedicated system, the configuration of the network that connects all processors together (the network topology) may directly reflect the logical interconnections that result from the chosen decomposition (this was already suggested in Figure 4.2). In general purpose systems or in situations where no direct mapping of processes to processors is possible, a more general topology is often used (see Figure 4.3). A consequence of the use of a general topology is that routing techniques must be applied to relay messages between logically connected processes over the physical network.

4.3 The `occam` programming language

Most high level programming languages are available for the transputer, including C, Fortran and Pascal. To gain most benefit from the transputer's architecture however, the language of choice is known as `occam` [38]. The basic principle behind this language is to avoid unnecessary duplication of mechanisms to write programs. It thanks its name to William of Occam, a 14th century philosopher who proposed that new entities should not be duplicated beyond necessity[3], a proposition that later

[3] *Entia non sunt multiplicanda praeter necessitatem.*

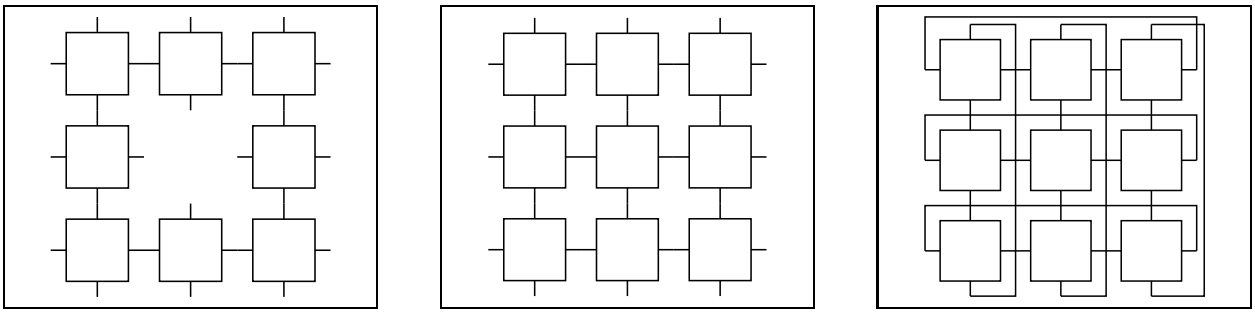

Figure 4.3: Examples of network topologies. From left to right: ring, grid and torus.

became known as "Occam's razor".

The simplicity of the language makes occam useful in the specification and verification of processes that must be free of deadlocks, livelocks and starvation. The language can even be used for the description of processes that later need to be implemented in hardware [45]. Indeed, the design and implementation of the transputer has gone hand in hand with the development of the occam language and thus the transputer can be seen as an occam machine (occam has for this reason often been called "the assembly language of the transputer" although the actual assembly language of the transputer is quite different from occam). Unfortunately, the development environments that have been available for occam rarely enforce the strict adherence to paradigms that are required for the development of applications that can be proven to be correct. One exception is the *occam Design Tool* (ODT) by the University of Kent at Canterbury which, when used properly, "guarantees freedom from race-hazards and aliasing problems" [6, 7]. Unfortunately, ODT still lacks some important features that prevent it from being used in the design of serious applications.

The language occam 1 was a direct derivation from Tony Hoare's CSP and David May's *Experimental Programming Language* (EPL). This prototype language was extended with floating point numbers, functions and a data typing system to form occam 2 (this language was used in the design of all programs in CACE-II). In 1992 INMOS released yet another version (occam 3, of course) that introduced structured datatypes, remote call channels, shared communication channels and modules into the language [4]. From this point on, each mention of occam refers to occam 2.

Without going into too much detail, the following two subsections provide a global overview of occam's syntactic and semantic characteristics. For a detailed description, see [38].

4.3.1 Occam's syntax

Processes

All occam programs are built from *processes*. From the lowest level to the highest, processes can either be:

1. *actions* — an assignment, an input, an output, SKIP or STOP, for example:

```
x := 1                      -- assignment
from.process.a ? result     -- input over channel
to.process.b ! x            -- output over channel
SKIP                        -- do nothing and terminate
STOP                        -- do nothing and never terminate
```

2. *constructions* — sequences, parallels, alternations, conditionals, selections or loops of actions, for example:

```
-- do the following in sequence
SEQ
  x := 1                    -- first; assignment
  from.process.a ? result   -- second; input over channel
  to.process.b ! x          -- third; output over channel

-- do the following in parallel
PAR
  x := 100                  -- assignment, and
  from.process.a ? result   -- input over channel, and
  to.process.b ! 10         -- output over channel

-- do one of the following
ALT
  from.process.a ? result   -- input from process A, or
  from.process.b ? result   -- input from process B

-- conditional: do one or more of the following
IF
  x < 10                    -- if this is true
    value := 1              --   do this assignment
  x > 10                    -- if this is true
    to.process.b ! 10       --   do this channel output
  TRUE                      -- otherwise
    STOP                    --   stop execution

-- selection: do one of the following
CASE letter
  'a', 'e', 'i', 'o', 'u'   -- if one of these is true
    vowel := TRUE           --   do this assignment
  ELSE                      -- otherwise
    SKIP                    --   do nothing

-- loop
WHILE x < 10                -- while this is true
  from.process.a ? x        --   do this channel input
```

3. *instances* — a new process built from constructions, for example:

```
PROC decrement (INT x)      -- definition of name and argument(s)
  x := x - 1                -- body of the instance
:                           -- a colon ends an instance declaration
```

`Occam` processes running concurrently on the same processor may have different priorities. Lower priority processes execute when all higher priority processes are idle. Prioritized processes are spawned using the special keyword `PRI` with the keyword `PAR` or `ALT`, for example:

```
PRI PAR
  process.A()    -- highest priority process
  process.B()
  process.C()    -- lowest priority process
```

Unfortunately, the implementation of occam for the transputer limits the number of processes within a PRI PAR or PRI ALT construct to only two, thus allowing only high and low priority processes.

Layout rules

Indentation in occam programs is important. All processes that are to be part of the same construction must be indented two spaces with respect to the construction keyword. The following examples for instance, are radically different:

```
  1:  SEQ
  2:    x := 3
  3:    PAR
  4:      x := 10
  5:      y := 2
! 6:    to.process.b ! x + y
```

```
  1:  SEQ
  2:    x := 3
  3:    PAR
  4:      x := 10
  5:      y := 2
! 6:      to.process.b ! x + y
```

The main difference here is that in the code on the left there are three sequences and two parallels while in the code on the right there are two sequences and three parallels. There's more to this example than just that, as will be shown in the discussion on *Parallel disjointness* in Section 4.3.2.

Replication

At first glance it might seem there's a construction "missing" from the language that is equivalent to the for loop so familiar from other languages. In occam an even more powerful mechanism is available known as "replication". The SEQ, PAR, ALT and IF constructions may be replicated using the following syntax:

construct name = *base* FOR *count*
construct = SEQ | PAR | ALT | IF

For example, the construction SEQ i = 0 FOR 10 is equivalent to our familiar sequential for loop, but using this construction it is now also possible to do things like this:

```
-- start 3 worker processes
PAR i = 0 FOR 3
  worker(i)
```

In occam this is effectively the same as:

```
-- start 3 worker processes
PAR
  worker(0)
  worker(1)
  worker(2)
```

The replicated construct allows a programmer to start a specified number of parallel processes in just two lines. Similarly, the following example on the left uses a replicated alternative to monitor the input from a number of processes and serve whichever process is ready to transmit a value. The example on the right uses a replicated conditional to compare two strings:

```
ALT i = 0 FOR 10
  from.process[i] ? value
    to.process[i] ! 2 * value
```

```
IF i = 0 FOR length
  string1[i] <> string2[i]
    equal := FALSE
```

Data types

`Occam` programs act on variables and channels. A variable may be assigned a value through an assignment or a channel input. Channels communicate values. The primitive data types available in `occam` and their representation are shown in Table 4.2.

data type	*values*
`BOOL`	`TRUE` or `FALSE`
`BYTE`	8 bit unsigned integer value from 0 to 255
`INT`	implementation specific, most efficient twos complement signed integer
`INT16`	16 bit twos complement signed integer
`INT32`	32 bit twos complement signed integer
`INT64`	64 bit twos complement signed integer
`REAL32`	ANSI/IEEE 754-1985 floating-point number with 8 bit exponent, 23 bit fraction, and 1 sign bit
`REAL64`	ANSI/IEEE 754-1985 floating-point number with 11 bit exponent, 52 bit fraction, and 1 sign bit

Table 4.2: Primitive data types in `occam`.

Communication channels are declared using the keyword `CHAN OF` followed by a *protocol*. A protocol may be a primitive type or the name of a protocol definition defined using the keyword `PROTOCOL`:

```
-- simple protocol channel
CHAN OF BYTE terminal :

-- sequential protocol definition and channel declaration
PROTOCOL COMPLEX IS REAL64; REAL64 :
CHAN OF COMPLEX c :

-- variant protocol definition and channel declaration
PROTOCOL EVENT
  CASE
    error;   BYTE
    start;   INT16; BOOL
    record;  INT::[]BYTE
    halt
```

```
:
CHAN OF EVENT e :
```

A special form of communication channel are the *timers*. Timers behave as channels except that they have 3 important differences over process communication channels:

1. timers only provide output (i.e. they can not be set to a value),
2. timers may be accessed by any number of concurrent processes, and
3. timers are always ready to communicate.

The rate at which a timer is incremented is implementation dependent. Using `occam` on a transputer, two types of timers available; one with a period of 64 μs (available only to low priority processes) and one with a period of 1 μs (available only to high priority processes). Timers are declared via the special keyword `TIMER`:

```
PRI PAR
  -- high priority process:
  TIMER clock :                      -- 1 us resolution timer
  INT now :
  SEQ
    clock ? now                      -- get current timer value
    clock ? AFTER now PLUS 1000000   -- wait one second

  -- low priority process:
  TIMER clock :                      -- 64 us resolution timer
  INT now :
  SEQ
    clock ? now                      -- get current timer value
    clock ? AFTER now PLUS 15625     -- wait one second
```

The only non-primitive types in `occam` are arrays. Arrays are formed from the primitive data types and may have any number of dimensions:

```
[4]INT vector :              -- a four component array of integers
[64][64][64]INT16 cacedata : -- CACE data can be represented this way
[n]CHAN OF INT sensors :     -- n channels of type integer
[4]TIMER clock :             -- four timers
```

4.3.2 Occam's semantics

Parallel disjointness

Please recall the boxed `occam` code examples on page 25 in which I discussed the significance of indentation in an `occam` program.

There is nothing wrong with the code on the left; `x` and `y` are assigned literal values in parallel in line 4 and 5, the sum of which is output to process B in line 6. In the code on the right however, the variable `x` is assigned while its value is used in an output to process B *in parallel*! To put this in other words, the variable `x` is shared between two processes thereby creating a *dependency*. As

the order of execution of processes in a PAR construct is undefined, this code will be rejected by an occam compiler. If this were not the case, process B could receive the value 3 + 2 = 5 or 10 + 2 = 12, we really can't say. The general rule in occam is that parallel processes must be *disjoint*, meaning they may not share resources by which they can influence each other's execution other than through synchronization via communication channels.

An array may be used in more than one parallel process if and only if the indices used to select the items of the array can be determined at compile time. The compiler determines whether the accesses to the array are non-overlapping (i.e. disjoint). Occam allows an array to be decomposed into non-overlapping, disjoint parts using *abbreviations*:

```
[1000]INT array :
lower IS [array FROM 0 FOR 500] :
upper IS [array FROM 500 FOR 500] :
PAR
  process.A(lower)
  process.B(upper)
```

Note the fundamental implication parallel disjointness has on an occam programs: all communication between processes must take place through channel communication (i.e. message passing). Occam does not directly allow processes to communicate via shared memory.

IF and CASE must be closed

Now please recall the boxed occam code on page 26 which gives an example of a replicated IF statement on the right.

Assuming that the variable equal was assigned the value TRUE before we entered the code in this example, what happens when, say, string1[4] *equals* string2[4]? Indeed, there is no expression in the conditional that states what to do in this situation, only what to do when string1[4] *not equals* string2[4]. Conventional languages like C, Pascal and Fortran would simply do nothing so that the variable equal would just remain unchanged so that at the end of execution this variable would reflect whether the two strings are equal or not.

In occam however, the conditional (and also the selection) must be *closed* which means that if all expressions in the conditional evaluate to FALSE, the process will behave as the primitive process STOP: further execution of the process will cease. It is therefore convenient to add an expression that is guaranteed to be performed. The previous example would in this case look like this:

```
IF
  IF i = 0 FOR length
    string1[i] <> string2[i]
      equal := FALSE
  TRUE
    equal := TRUE
```

This "feature" of occam forces a programmer to asses all possible situations that may occur while writing a conditional or selection statement. While this can aid in the construction of better, perhaps even correct programs, it is very easily forgotten.

The behaviour of a program (which may consist of more than one process) as a whole when one of its processes behaves as the primitive process STOP is specified at compile time. See Section 4.4.2 for more details.

4.4 Embedded system design using transputers and occam

The new requirements for the CACE-II computational architecture and programming language were addressed in Section 3.3.1. In the following subsections, the adherence of the transputer and the occam programming language to these requirements is assessed.

4.4.1 The transputer

The transputer forms a good building block for the construction of an embedded parallel architecture. As shown in Section 4.2, the on-board math unit and the powerful RISC processor allow the transputer to be used as a high performance stand-alone processor. In addition, the multitasking support in hardware and the high speed communication links allow an easy expression of concurrency and parallelism.

A suitable candidate for integration into the CACE system was found in the Parsytec BBK-V2 [13]. The Parsytec BBK-V2 is a VME module containing a 20 MHz T805 transputer with 2 megabytes of dual-ported (DP) dynamic RAM. In CACE-II the DMA interface can be configured in such a manner that it addresses a 512 kilobyte memory area on the BBK-V2. The use of dual-ported memory on the BBK-V2 transputer module allows the DMA interface to access memory locations without interfering with accesses that are performed by processes running on the main transputer in its remaining memory. The BBK-V2 is also capable of performing VMEbus arbitration.

Additional transputers were added using the Parsytec VMTM [43]. This VME module contains four 20 MHz T805 transputers, each with 1 megabyte of private dynamic RAM. The network configuration between the four transputers can be software configured using an on-board crossbar switch (an IMS C004). The BBK-V2 and the VMTM can be coupled using RS-422 buffered interfaces on the front panel of each module.

4.4.2 Occam

Section 4.3 provided an overview on the syntax and semantics of the occam programming language. In the following, these properties are compared to the requirements that have been put down in Section 3.3.1.

Multitasking support and scalable multiprocessor support

The transputer's unique hardware multitasking support allows multiple processes to be executed on the same processor very efficiently and without any additional support by an operating system. In addition, the communication links provided by the transputer allow parallel systems to be built on which processes can execute in parallel, communicating with each other through message passing. Occam provides direct support both for expressing concurrency as well as message passing.

Structured Design

The concept that everything in occam is a process makes an Object Oriented Design (OOD) of a program follow quite automatically. Using a top-down design, a program is split up into several logical independent subprocess (which may be considered objects) that communicate with each other via channels. This type of abstraction allows an embedded system to be decomposed into disjoint objects, each consisting of input channels to which *events* can be sent to activate an object, after which the object sends the response over an output channel. These event-driven objects can be implemented relatively easy and allow the construction of efficient and manageable systems [49].

A disadvantage of occam over most high level programming languages is that occam does not support structured data types. Structured types are composed of elements that are each of different type and help in the natural abstraction from the information that needs to be represented [54]. Although data structures can always be rewritten into occams simple data types and arrays, this has a negative impact on the readability of a program.

Maintainability

The abstraction of a program into independent subprocesses in occam results in a reduction in complexity that makes it easier to understand and maintain a program. On the other hand, the strict layout rules of occam impose an often frustrating restriction on a programmer's freedom to create an easily readable program. A ruler is an indispensable tool in reading a large occam program in order to figure out what construct a specific statement is part of.

Error checking

The purpose of type checking is to prevent errors [54]. Type checking systems enforce that operations on data items properly adhere to the defined types of those items. Type errors occur when a function is applied to a data item of incompatible type. In occam, type checking is performed both *statically* (at compile-time) and *dynamically* (at run-time).

Statically determined type errors are signaled by the compiler when an operator is applied to data items of incompatible types (such as adding a floating point value to an integer value). Operands may explicitly have their data type converted. Run-time errors occur when the result of a process does not "fit" the destination variable (so called *range errors*). A process that generates a run-time error behaves as the primitive process STOP.

Fault handling

An occam process that causes a run-time error is mapped to the primitive process STOP. Run-time errors in occam are handled in one of three ways (specified at compile time) [30, page 50]:

- HALT mode – In this mode, all run-time errors bring the whole system to a halt. This prevents an errant process in the system to corrupt any other part of the system.
- STOP mode – In this mode, only errant processes are mapped to the primitive process STOP again ensuring that no errant process in the system can corrupt any other part of the system. As the remaining processes are still allowed to continue, the system can detect that another process has failed (e.g. via watchdog processes) thus allowing gracefully degrading systems to be constructed.
- UNIVERSAL mode – In this mode, processes may behave as either HALT or STOP mode depending on the transputer's Halt-On-Error flag. When a library compiled in UNIVERSAL mode is linked with modules in HALT mode, the processes in the library behave as if compiled in HALT mode. Mutatis mutandis for STOP mode.

Low-level device access

Occam provides access to memory mapped hardware devices by placing variables over locations in memory. In addition, occam allows these placed variables to be specially typed as if they were channels, allowing processes to access devices in a way similar to communicating over channels:

```
INT16 control.register, value :
PORT OF INT16 command.register :
PLACE control.register AT #FFE00100 : -- map INT16 onto memory address
PLACE command.register AT #FFE00102 : -- map PORT onto memory address
SEQ
  control.register := #0000 (INT16)   -- reset interface (memory mapped)
  command.register ! #1234 (INT16)    -- initialize (port mapped)
  command.register ? value            -- read result (port mapped)
```

The explicit definition of data types with a defined number of bits helps in the construction of programs that need to access memory mapped hardware devices, which in most circumstances must be accessed using specific data widths. With many other languages, such as C [36], the width of data types is not explicitly defined by the language, but depends on the compiler implementation. This makes it hard to write hardware accessing programs that are portable over different compilers.

Interrupt handling

The interrupt handling mechanism in the transputer is slightly different from that seen on other processors. The transputer's event channel merely signals the occurrence of an event by unblocking an event handler process that attempts to read from it:

```
CHAN OF BYTE event :
PLACE event AT 8 :        -- map to transputer event channel
BYTE dummy :
WHILE TRUE                -- repeat 'till doomsday
  SEQ
    event ? dummy         -- wait for event to happen
    handle.event()        -- recognize and handle it
```

The event handler is assumed to be able to recognize the origin of the event from the hardware resources available elsewhere in the system and handle the event. Fortunately, the BBK-V2 module provides ample support to achieve this.

The event handler should be a high priority process to ensure that events are quickly serviced once they occur.

Efficiency

As already stated in Section 4.3, the transputer can be considered an `occam` machine. The use of `occam` on the transputer provides the equivalent efficiency to programming a conventional computer in assembly language [38].

Portability

Although the `occam` language is not restricted to execution on a transputer, very few compilers are available for the compilation of `occam` programs to other platforms. Until this date, the following portable `occam` compilers are now available:

- *Southampton's Portable Occam Compiler* (SPOC) translates `occam` source into ANSI C and then uses the native C compilers to generate the target code [12]. This approach allows a

portable occam compilation system to be produced and allows back-end optimization to be performed by the C compiler in a manner optimized to the target processor. The implementation of an occam-to-C translator is not a trivial task since occam supports many features that have no direct equivalent in C. Preliminary performance results have shown that the level of efficiency that can be achieved is reasonable. In fact, on some processors the results exceeded those of the T800 transputer, but it should be noted that these processors were not of comparable performance (translated code running on a 40 MHz SuperSparc was compared to the original occam code running on a 25 MHz T800 and showed an increase in performance of approximately 30%).

- The *occam For All* project is an effort by industrial end-users, technology suppliers and academic researchers to provide occam compilers for all major computing platforms [47]. This project was funded by the *UK Engineering and Physical Sciences Research Council* (EPSRC) as part of its *Portable Software Tools for Parallel Architectures* (PSTPA) programme. The formal duration of this project was from February 1st, 1995 to May 31st, 1997, but work is said to continue.

  The *Kent Retargetable Occam Compiler* (KROC) is a development of the occam For All project, now providing a portable occam compiler for Sparc (with SunOS 4.1.3 or Solaris 2.5) and Alpha (with OSF/1 3.0) processors [4].

4.5 Conclusions

In this chapter the basic building blocks for the CACE-II embedded architecture have been presented. It has been shown that the transputer and the occam programming language fulfill most of the requirements that have been discussed in Section 3.3.1. In Chapter 5 the CACE-II embedded architecture and the software that has been implemented will be presented.

During the development of the CACE-II system, lots of experience was obtained that shed additional light on both the transputer and occam. The following subsection address some of the most important problems that have been encountered during that time.

4.5.1 No dynamic memory allocation

One particular problem that is very hard to escape from in occam is the lack of a dynamic memory allocation mechanism. This omission from the language was deliberate as dynamic memory allocation makes formal reasoning about programs difficult. All memory in occam programs is statically declared at compile time so that a program's memory usage is known at runtime. For more serious applications this shortcoming results in tedious efforts by most programmers to either implement dynamic memory allocation primitives themselves, or otherwise use static memory structures that are big enough for most cases, thereby wasting memory.

No recursion

A consequence of the lack of a dynamic memory allocation mechanism is that occam does not allow recursion. Recursive procedure calls require stacks to maintain the data structure for each called level which can not be deduced at compile time. The difficulty with recursion is computing the stack size required. This is not possible in the general case (similarly, it is not possible, in general,

[4] Version 0.9 Beta, released into the public domain on March 31, 1997.

to confirm that programs containing WHILE loops will ever terminate). Therefore occam does not allow recursion. The implementations of the C language for the transputer offer the potential for recursion, but requires the programmer to manually specify the size of the stack for every process, and allow a safety margin.

There exist mappings for all real recursive programs such that the recursion is replaced by iteration. It then falls to the programmer to provide an array for the data structures which, in the recursive version, would be provided by the compiler using the stack.

4.5.2 No non-blocked communication

Timers allow occam programmers to implement semi non-blocked channel input when no data from channels is available. Using the ALT construct it is possible to let a channel input instruction time-out after a specified interval, as illustrated in the following example:

```
CHAN OF INT input :
TIMER clock :
INT value, now :
VAL timeout IS 1 :  -- 64 us for standard priority processes
SEQ
  clock ? now
  ALT
    input ? value
      do.something.with(value)

    clock ? AFTER now PLUS timeout
      SKIP
```

This construction is especially useful in situations where processes only need to revert their execution when input is available from other processes. Unfortunately, it not possible in occam to implement non-blocked channel *output*. This is due to the way that communication is handled by occam and the transputer: once a process offers to output data, this offer can not be withdrawn [33].

4.5.3 Process scheduling on the transputer

In the case of low priority processes, it is important to realize that (in contrast to common belief) the transputer's process scheduler is not a true pre-emptive scheduler in the sense that a process can be suspended at an arbitrary instant, without warning [61]. Low priority processes are pre-empted *only* when a process is idle. A process is idle when it has suspended itself (by executing a SKIP instruction) or when it initiates synchronization (i.e. channel communication) with another process that is not ready to synchronize. This is especially important to consider in the design of hard real-time systems where it is essential that running processes can be descheduled in favour of processes that have to meet a deadline.

4.5.4 Multidimensional arrays

Most programming languages provide multidimensional arrays as a method to clearly express the structure of multiparameter homogeneous datasets. A disadvantage with these arrays however, is that some run time overhead can not be avoided each time an item in the array is referenced.

The compiler needs to insert program code to translate the array indices into a memory position relative to the start of the array. The amount of time that is required to evaluate these expressions depends on the number of dimensions in the array. Especially in cases where the entire array must be traversed (which is typical for many processes in the CACE-II system), the total overhead caused by this may substantially accumulate.

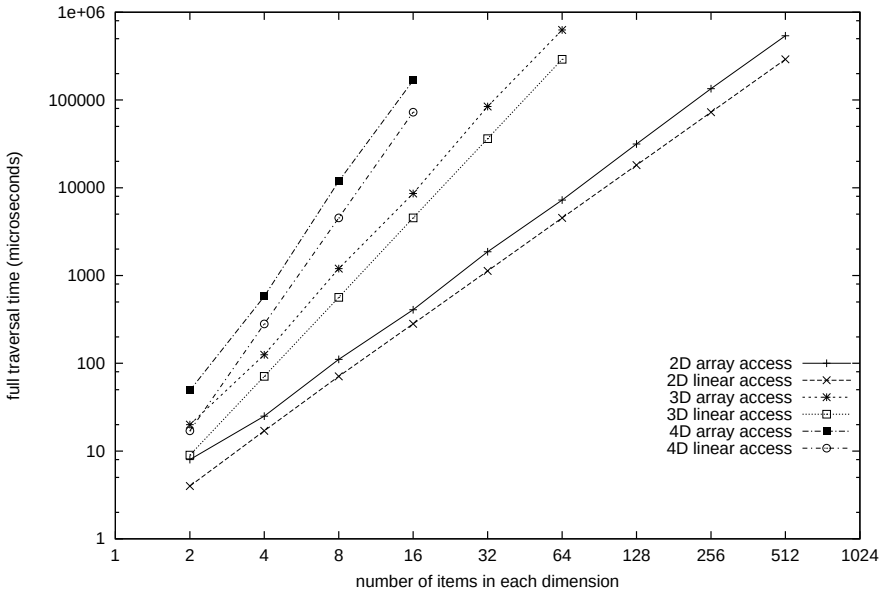

Figure 4.4: Traversal time required to access different sized multidimensional arrays using either multidimensional array indexing or linear access in `occam`. Note the logarithmic scale on both axes.

Figure 4.4 shows the effect of this overhead in `occam` for 2, 3 and 4 dimensional arrays of various sizes. As can been seen in this figure, the overhead is quite substantial, in some cases even well over twice the time required to linearly access the same array. Part of the `occam` programs that were used to measure this difference is shown here for a three-dimensional array; on the left array indexing is used, on the right linear access:

```
[N][N][N]INT array :
SEQ i=0 FOR N
  SEQ j=0 FOR N
    SEQ k=0 FOR N
      array[i][j][k] := 0
```

```
[N][N][N]INT array :
[]INT linear RETYPES array :
SEQ i=0 FOR (SIZE linear)
  linear[i] := 0
```

The main cause for the difference in access times is the limited set of registers that are available in the transputer. The array indexes need to be calculated into an address using the transputer's stack evaluation based instruction set. As the number of dimensions in an array increases, more indexes need to be pushed and pulled from the register stack which accounts for most of the time in a single access to an array item. This overhead is significantly reduced in a linear array.

Wherever possible, linear array accesses were used in CACE-II processes. However, this did have a very negative influence on the readability of the programs.

4.5.5 ALT implementation on transputer is unfair

The semantics of the `ALT` instruction permit that *any* of the alternatives is chosen as the path of subsequental execution, provided the alternative is ready. Unfortunately, if all alternatives in the `ALT` construct are ready *all* the time, the transputer implementation guarantees that the first alternative is executed. This makes a fair scheduling of processes using the `ALT` construct difficult [5, 34].

Chapter 5

The CACE-II embedded architecture and implementation

In the previous chapters, the requirements and building blocks for the new CACE-II system were discussed. This chapter describes the new architecture and the software that has been implemented on the CACE-II system for the on-line analysis of data obtained during a separation experiment.

5.1 The CACE-II architecture

Figure 5.1 shows a schematical representation of the CACE-II embedded architecture. A comparison of this figure with Figure 3.1 shows that the 68000 CPU and the dynamic RAM modules have been substituted for the BBK-V2 and VMTM modules. The graphics interface and the network board have been removed. The DMA interface and the FCM equipment are unchanged.

The interface from the embedded system to the outside world consists of an Apple Macintosh II computer which has a TPM-MAC transputer card inserted into one of its slots [64]. The TPM-MAC card contains a 17.5 MHz T805 transputer with 1 megabyte of dynamic RAM. This card is connected to the embedded system via the transputer's communication links. The Macintosh functions as a gateway to an Ethernet network [1] so that the CACE-II system can be accessed from workstations located anywhere on the network.

A TSVME-440 motor controller and a sampler module were added for the autofocus subsystem, which is described in Chapter 6.

5.2 A parallel convolution filter

Apart from the instrumental noise that is known to be present in the analog signals from the scatter detectors and amplifiers, quantizing the analog data during the conversion by an analog-to-digital converter also adds stochastic noise to the acquired data that needs to be removed before any subpopulations can be identified. This additive noise signal can be regarded as composed of frequency components with no preference to any special frequency (i.e. *white noise*), while the signals generated by the light scattering cells are basically low frequency. The frequency content of the N-channel histogram function (represented by $f(x)$) can be studied by means of spectral

[1] With thanks to Jos Vermeulen of NIKHEF who graciously provided the adapted software for this.

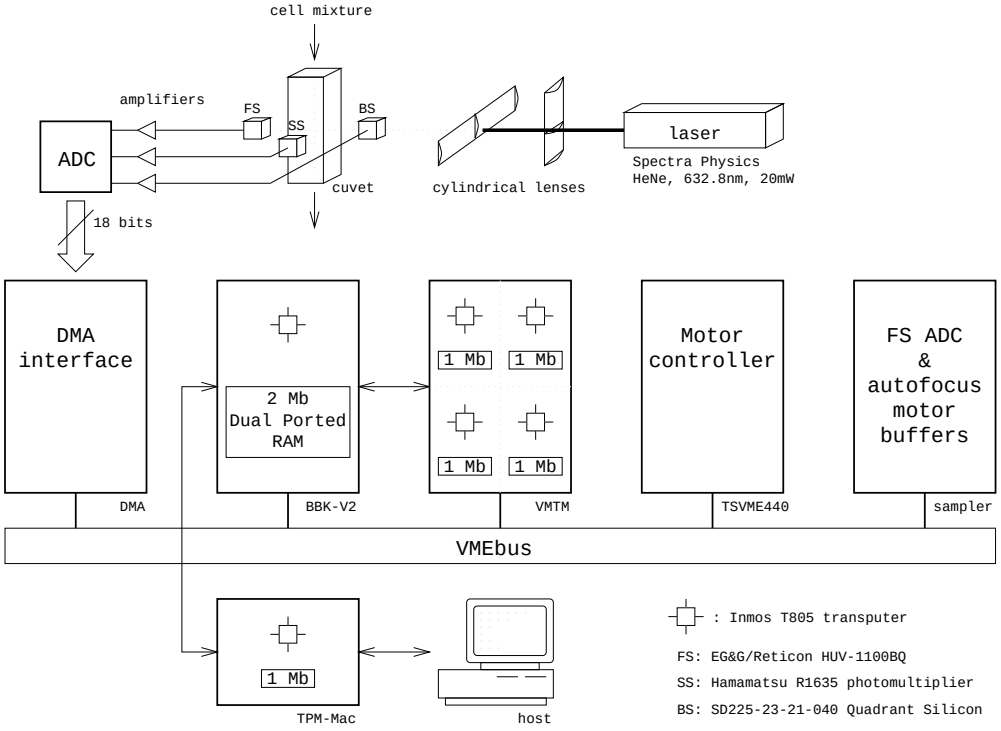

Figure 5.1: Schematic representation of the CACE-II architecture.

decomposition. This is accomplished by calculating the Fourier transform $F(\omega)$ from $f(x)$:

$$F(\omega) = \sum_{x=0}^{N-1} f(x) \exp(-i\pi x\omega).$$

Here $F(\omega)$ is a complex number; the normalized frequency $0 \leq \omega \leq 1$ is defined by $\omega = fT$ where T is the interval between the N datapoints; and $0 \leq f \leq 2\pi$.

The frequency content of the acquired histogram $f(x)$ can be described by a signal function $s(x)$ containing low frequency components which form the response from the relevant information function, and a noise function $n(x)$:

$$f(x) = s(x) + n(x).$$

The frequency content of the data is extremely regular [58]; $N(\omega)$, the Fourier transform of $n(x)$, will be mostly constant while $S(\omega)$, the Fourier transform of $s(x)$, will be a decreasing function. From this spectrum, a cut-off frequency ω_c can be deduced from which a low-pass filter function $h(x)$ can be constructed that removes the noise and guarantees consistency of the data contents.

Convolution of the histogram function $f(x)$ with a function $h(x)$ is represented by

$$s(x) = (s(x) + n(x)) * h(x)$$

where $*$ represents the convolution operator. Convolving two signals in the time domain results in a Fourier transform that is the multiplication of the Fourier transform of the two original signals. Fourier transformation results in

$$S(\omega) = (S(\omega) + N(\omega)) \cdot H(\omega).$$

For the construction of the filter function $h(x)$, the kernel of a very simple Lancsoz window is used to facilitate the implementation. The application of such a kernel is necessary to reduce the "ringing" effect that would be present otherwise, i.e. if a rectangular filter function was used.

5.2.1 Basic function requirements

The parallel filter performs a three-dimensional Fast Fourier Transform (FFT) on the raw data acquired by the DMA interface. The transformations to the Fourier domain can be done without restrictions to the dependency of the data [58]. After transforming the data, the filter function in the Fourier domain is executed. This function multiplies the first $Nc + 1$ and last Nc (where Nc is the cut-off channel, determined from ω_c) elements of an array by the filter function and sets the other elements to zero. After the inverse Fourier transformation in all dimensions, the convolution is completed. For these operations, the following Fourier transformation primitives are required:

- rFFT - a real-to-complex FFT, used for the transformation in the first dimension;
- FFT - a complex-to-complex FFT, used for the following transformations;
- iFFT - a complex-to-complex inverse FFT, used for the inverse transformations;
- HiFFT - a complex-to-real Hermitian inverse FFT, used for the last inverse transformation (see section 5.2.3).

5.2.2 Problem decomposition

The parallel digital filter is built on the *single program multiple data* (SPMD) paradigm, meaning that the same program is spread over the available processors, each working on a different portion of the input data. This method of decomposition is commonly referred to as *data decomposition*: the input data is split into equally sized sections that are spread out over the available processors (see Figure 5.2). If the N planes can not be evenly divided over all processors, the remaining planes are each divided over the first N mod P processors.

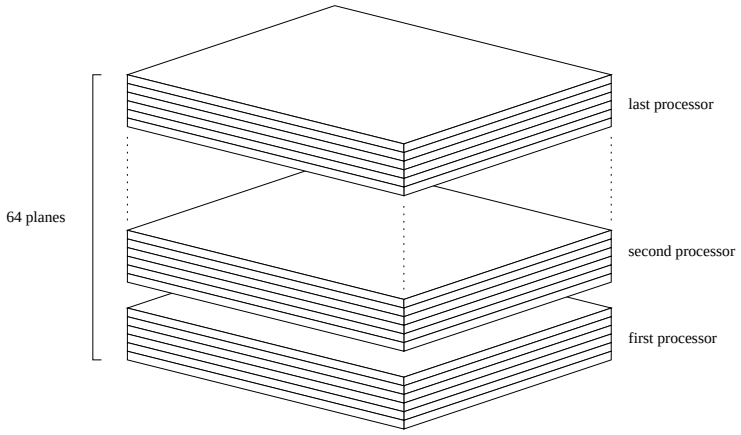

Figure 5.2: Data decomposition used in the parallel digital filter.

A slight exception to the SPMD model must be made for the first processor in the network that executes the *root process*; apart from performing filter operations, the root process is also concerned with "scattering" the raw input data acquired by the DMA data acquisition module over the *worker*

processes. The root process reads this raw data and spreads it over all available processes, including itself. When the filter processes are ready, the root process "gathers" the results back. Furthermore, as communication with the other processors occurs via the CSP model, the root process essentially synchronizes the rest of the network. The worker processes only perform filter operations.

5.2.3 Description of the filter process

An overview of the execution structure of the filter process is shown in Figure 5.3. This structure is explained in some more detail below. The numbers below refer to those in Figure 5.3.

```
 1. initialize tables of complex e-powers
 2. scatter raw data over all processes
 3. transform and filter in x direction
 4.     transform and filter in y direction
 5.         forward reorder
 6.             transform and filter in z direction
 7.             inverse transform in z direction
 8.         backward reorder
 9.     inverse transform in y direction
10. inverse transform in x direction
11. gather the filtered data and store
12. (repeat from step 2)
```

Figure 5.3: Global execution structure of the filter process.

1. The tables of complex e-powers which are required for the Fourier transform primitives are pre-calculated at process startup. Computing these tables at this point greatly reduces the number of calculations that need to be executed when the filter process is executed repeatedly as now they only need to be performed once.

2. The root process starts the filter process by first distributing the data as blocks over the available transputers. Depending on the number of transputers available, a number of two-parameter planes will be passed to each of the workers on each of the transputers (as explained in Section 5.2.2).

   The raw input data for the parallel filter is built up out of 16 bit integers. As the Fourier transform primitives operate on real data, all processes first convert their data to real numbers right after they have received their data from the root process. Looking at the structure of the input data as a box of $N \times N \times N$ items, it would seem logical to store this data into a three-dimensional array. However, since `occam` is notoriously inefficient at accessing multidimensional data arrays, the data is linearized into a one dimensional array. All administrative work on accessing this array is performed in the filter processes.

3. In each process, a real-to-complex rFFT procedure transforms the data in x direction. We are concerned with a low-pass filter, so the (high) noise frequencies to be removed are easily

located in the transformed arrays (they reside in the middle of the array). Therefore, this part of the data can be skipped in the following steps of the transformation (see Figure 5.4).

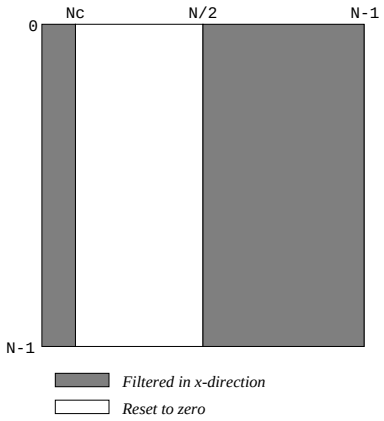

Figure 5.4: A single data plane after x-transformation. By filtering directly after the first transformation, the non-shaded area does not have to be transformed in the proceeding steps. In the next step, only the shaded area has to be transformed.

Also, the input data of the filter consists of real numbers, i.e. the imaginary part is zero. As a consequence, the transformed data is now *Hermitian*. This implies that half of the data can be skipped in all further operations due to the symmetry in the data (see Figure 5.5) [9].

4. For the y-transformation, each column of each data plane is copied and rotated into a buffer. For each column this buffer thus contains each of the Nc consecutive columns but *only* those that have not been set to zero by the previous filter phase. After each column is copied into this buffer, the transformation in the y direction is performed by a complex-to-complex FFT procedure. Right after that, all elements between the cut-off frequencies are set to zero (both for the real and the imaginary part) whereafter the line is restored back into the original arrays.

   After the x and y transformations, each data plane looks like Figure 5.5.

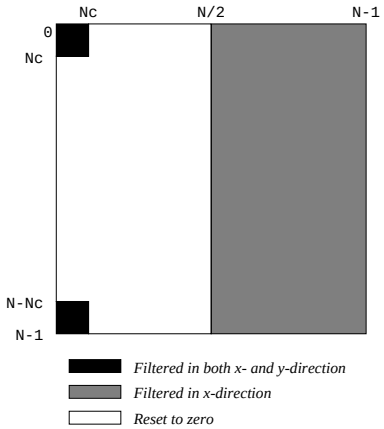

Figure 5.5: The data plane after x and y transformations. By skipping the right half of the data after the first transformation, the second transformation has to be performed through the whole shaded box and the third transformation only through the black shaded boxes.

5. Before the transformation and filtering in the z direction can take place, a reordering must take place as some of the data required for this transformation resides on the other processes. This concerns the black shaded boxes in Figure 5.5: each process contains part of this data but requires it all before z transformation can be performed. To achieve this, all processes send the black box to their right neighbours while receiving and storing the data coming in from the left. This is repeated for as many times as there are processors, after which all processes have all data to perform the z transformation.

6. Once the transformations in all three dimensions are performed and the data is filtered in all three dimensions, the data needs to be transformed back to the spatial domain in reverse order as the transformations took place. First, the data is inverse-transformed in the z direction by a complex-to-complex iFFT procedure.

8. Before the inverse transformations in the y and x direction can take place, the data has to reordered again so that each process contains its respective data plane again as before forward reordering. This backward reordering is essentially the inverse operation of the forward reordering mentioned previously.

9. After reordering, another complex-to-complex iFFT procedure inverse-transforms the data in the y direction.

10. A Hermitian complex-to-real iHFFT procedure inverse-transforms the data in the x direction and completes the basic filter operations.

11. Recall that the input data was converted from a 16 bit integer array to a linear array of real numbers. It is necessary to convert the resulting filtered data back into this format so that it can be processed by other processes in the data analysis pipeline. All worker processes convert their part of the data right before they send it off to the root process. The root process collects all data from the worker processes and stores it in the correct order.

12. In circumstances where the filter process is used in a digital analysis pipeline or in a running separation experiment where the DMA interface continuously acquires new data, the filter process is restarted.

5.2.4 Results

Figure 5.6 shows a parametric representation of a single plane extracted from a three-dimensional dataset which was acquired with the CACE-II system. The noise is clearly visible in this picture and it is hard to discern how many normal distributions are present in this plane. Figure 5.7 shows the same plane after the dataset has been filtered. Most of the noise is gone and it is clear to see that there are two overlapping normal distributions in this plane. Figure 5.8 shows the result of the parallel filter when the dataset shown in Figure 3.2 is used as input.

Performance of the parallel filter

Figures 5.9 and 5.10 respectively illustrate the execution time and speedup of the parallel filter process. This timing was performed on a Meiko CS-1 system containing 64 T800 transputers. As the operations in the filter process are not data content dependent, execution time shows a decrease as the number of processors employed increases.

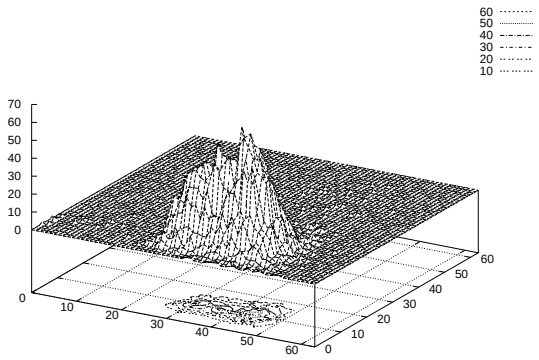

Figure 5.6: Raw data plane.

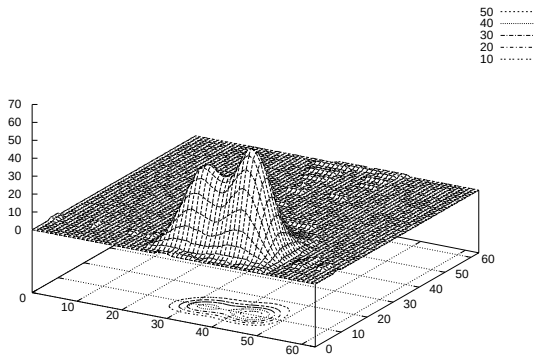

Figure 5.7: Filtered data plane.

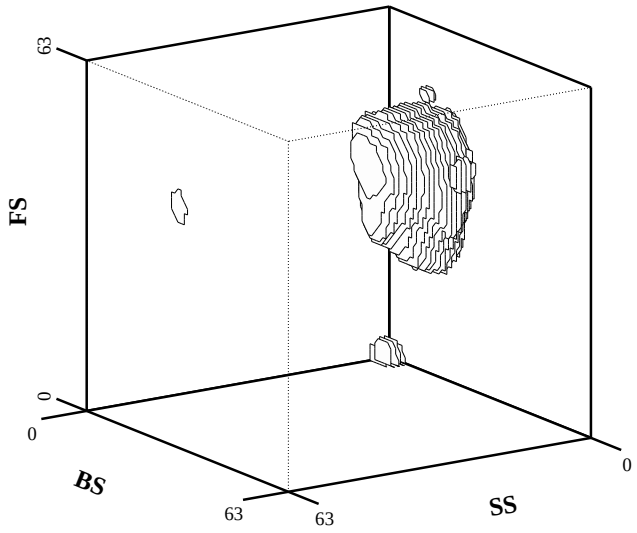

Figure 5.8: Filtered dataset of Figure 3.2. All (FS,SS,BS) points with contents ≥ 2 are shown.

The "staircase" patterns in these figures show the influence of the domain decomposition strategy used in the filter process: As the `occam` programming language used for the implementation of this process adheres to the CSP model, the complete filter process is only finished when *all* processes in the network have finished. In cases where the number of data planes can not be evenly divided over the available processors, some processors need to work longer than others because of the additional planes.

5.3 Subpopulation detection

Once the noise has been removed from the raw data, the different subpopulations in the acquired data may be detected. Unfortunately, this is not a trivial task. Especially in cases where subpopulations overlap one another because of similar light scattering characteristics in one or more detector directions, problems arise in deciding which elements belong to one subpopulation and which to another. An example of this can be seen in Figure 5.7 where two normal distributions

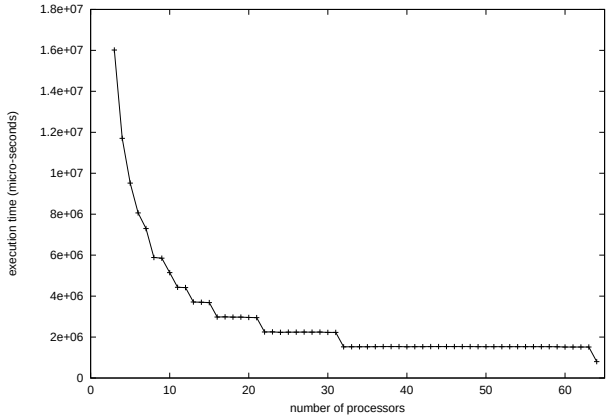

Figure 5.9: Execution time of the parallel filter process.

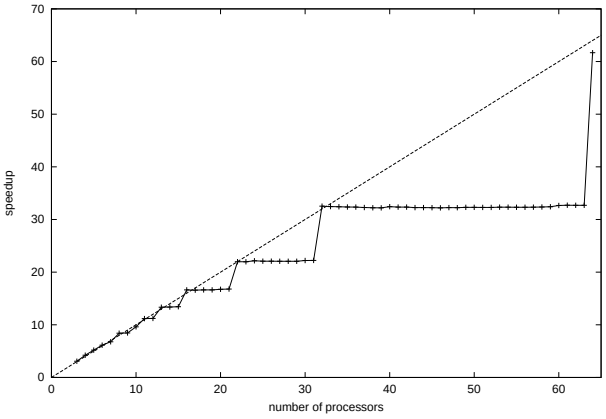

Figure 5.10: Speedup of the parallel filter process.

overlap one another.

5.3.1 Design issues

The histogram that is created during a CACE-II experiment consists of a mixture of multivariate normal distributions. The complete dataset can be described by:

$$P(\overline{r}|\Phi) = \sum_{i=0}^{m-1} \alpha_i \cdot p_i(\overline{r}|\phi_i); \overline{r} = (x, y, z)^T \in I\!R^3; 0 \leq x, y, z \leq 63 \tag{5.1}$$

Here m is the number of distributions and α_i the fraction of each population. Each distribution is furthermore characterized by $\phi_i = (\overline{\mu_i}, \overline{\overline{\Sigma_i}})$ in which $\overline{\mu_i}$ represents the vector of the mean and $\overline{\overline{\Sigma_i}}$ represents the (co-)variance matrix.

A two phased scheme has been devised in which first rough estimates are determined as to how many multivariate normal subpopulations reside in the data and what the approximate parameters are that describe these populations. These parameters are then optimized using a statistical method know as Expectation Maximization/Maximum Likelihood (EM/ML). Ultimately, these parameters then provide detailed information about a mixture such as the number of populations in a mixture, the total number of cells that have been detected, the number of cells in a particular subpopulation and the relative percentage in the complete mixture. In theory, the system could even identify the different kind of cells in each population given some prior knowledge about the mixture.

It should be noted that the EM/ML algorithm described in the next section is not able to create new distributions. Therefore, it is of utmost importance that the cluster process at least correctly determines the number of distributions in a dataset. Unfortunately, this has proven to be a difficult task.

5.3.2 Cluster analysis

Cluster analysis is the process of classifying objects into subsets that have a high degree of similarity to each other while the different subsets are relatively distinct [2, 17, 20, 32]. This definition implies that a distance (or proximity) measure needs to be devised in order to obtain a successful

classification of a set into different subsets. Here, a specialized algorithm is described that assumes that each individual cell population can be described by a multivariate normal distribution [59].

For a clustering operation in three dimensions, the data produced by the parallel filter process is divided over the available processors using exactly the same method as described for the filter process in Section 5.2.2. Each of the clustering processes makes an estimation of the parameters describing the populations in all two-dimensional planes by scanning each for local maxima in both directions. Once all planes have been scanned, neighbouring cluster processes negotiate on which distributions may be connected, and then combine their maxima to form a skeleton in all three directions of maxima for each detected distribution. Local maxima are then eliminated (i.e. considered not to be part of a distribution) according to the following criteria:

- each distribution that has a density less than a predefined value is discarded,
- each distribution that has a width less than a predefined value is discarded,
- two distributions that are closer to each other than a predefined value are "merged" to one distribution.

After this, the percentage of overlap between neighboring distributions and their relative orientation is estimated. This overlap is used to determine the "purest" part of a distribution. After this, the initial parameters for every distribution can be determined by fitting the maxima to normal distribution functions.

Performance of the parallel cluster process

The clustering algorithm described in this section performs fairly well, although it was observed that it is not always successful in detecting all distributions in a dataset containing randomly generated distributions. In all these cases, the distributions that were missed had some form of overlap with others. This is an important point to stress, as the statistical optimization algorithm that is described shortly is only able to optimize detected distributions; it is not able to create new distributions. Alternative algorithms have been reported in literature, but with these either the number of distributions must be known beforehand [23, 40], or they require other preknowledge about a mixture [60], or they also have problems with overlapping distributions [37].

The next figures illustrate the performance and speedup of the parallel cluster process. **Note however that the data shown here is for one particular dataset**. As the operations in the cluster process are clearly dependent on data content, this timing should be averaged over multiple different (i.e. randomly generated) datasets. Time constraints have inhibited me to do this.

5.3.3 Parallel Statistical Optimization

The result of the clustering process consists of an estimation of the number of distributions, and the parameters that describe each normal distribution in a mixture. The optimization process optimizes these parameters by means of a *maximum likelihood* (ML) technique called *expectation maximization* (EM).

The Expectation Maximization algorithm

The maximum likelihood technique estimates the parameters of a density function. The expectation maximization algorithm is a special iterative procedure which can be used to numerically estimate the ML of multivariate mixture densities while preserving statistical consistency [35].

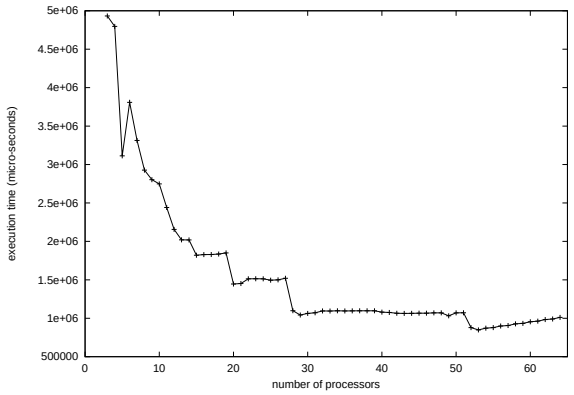

Figure 5.11: Performance of the parallel cluster process.

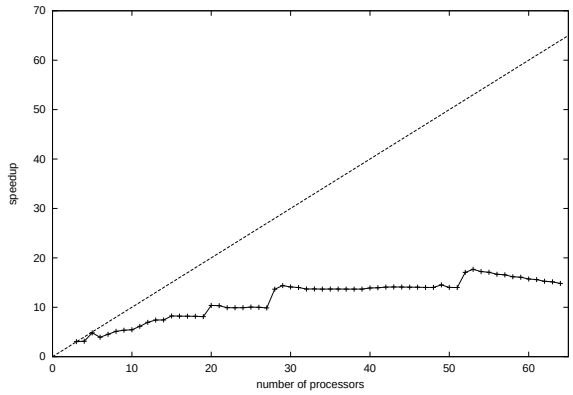

Figure 5.12: Speedup of the parallel cluster process.

From equation 5.1, the *log-likelihood* function can be derived. By setting the partial derivatives of this function to zero, the parameter values that maximize this function can be calculated by:

$$\nabla_{(\alpha,\overline{\mu},\overline{\overline{\Sigma}})} L(\Phi) = 0 \tag{5.2}$$

Solving equation 5.2 results in the following iteration schemes for α_i, $\overline{\mu_i}$ and $\overline{\overline{\Sigma_i}}$:

$$\alpha_i^{(r+1)} = f(\alpha_i^{(r)}, \overline{\mu_i}^{(r)} \overline{\overline{\Sigma_i}}^{(r)}) \tag{5.3}$$

$$\overline{\mu_i}^{(r+1)} = f(\alpha_i^{(r)}, \overline{\mu_i}^{(r)} \overline{\overline{\Sigma_i}}^{(r)}) \tag{5.4}$$

$$\overline{\overline{\Sigma_i}}^{(r+1)} = f(\alpha_i^{(r)}, \overline{\mu_i}^{(r+1)} \overline{\overline{\Sigma_i}}^{(r)}) \tag{5.5}$$

Within the iteration, the integration areas are used which were estimated by the cluster process. However, the iteration schemes imply that the data decomposition in two-dimensional sets of planes, as was used in the filtering and clustering process, can no longer be applied. A different strategy can be applied to parallelize this algorithm by devoting a processor to the iteration of the parameters for one distribution.

However; as can be seen from equation 5.5, $\overline{\overline{\Sigma_i}}$ is dependent on $\overline{\mu_i}$ from the *current* iteration step instead of the *previous* step. This would imply that $\overline{\overline{\Sigma_i}}$ can not directly be calculated in parallel to α_i and $\overline{\mu_i}$. One way to solve this implicit set of equations is to resort to matrix elimination techniques, but these are very computationally intensive. However, it can be shown that substituting $\overline{\mu_i}^{(r)}$ in equation 5.5 still provides an accurate estimate of all parameters at a cost of slightly slower convergence. But as the three iteration processes are now made explicit, they can be executed in parallel which still results in a faster iteration process.

As a stop criterion for the iteration, the Bhattacharyya distance measure was used to determine the distance between two consecutively iterated distributions P_1 and P_2:

$$dB(P_1, P_2) = -\ln \int_{-\infty}^{\infty} \sqrt{P_1(\overline{r}|\phi_1) \cdot P_2(\overline{r}|\phi_2)} d\overline{r}.$$

Since this distance measure includes all significant statistical parameters of the distributions, it can be applied as a non-ambiguous condition for the termination of the iteration process.

Functional decomposition

The iteration schemes just presented can each be executed on a separate processor. In this functional decomposition, three processors are used to calculate the iteration schemes of respectively equation 5.3, 5.4 and 5.5 (see Figure 5.13). A fourth processor controls which distribution is iterated and calculates the Bhattacharyya distance between the consecutively iterated distributions. Clearly, this functional decomposition scheme can not be run on more than four processors and is therefore not scalable.

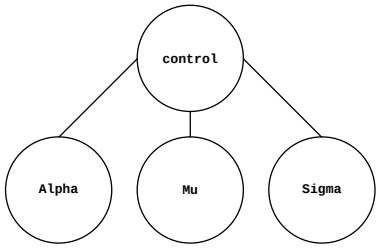

Figure 5.13: Functional decomposition used in the parallel statistical optimizer process.

Performance of the parallel statistical optimizer

The current version of this software still has some problems with convergence. In some cases the algorithm does not converge at all, while in others the stop criterion seems to be reached while in reality it isn't. Reasons for this have as of yet not been found.

Some tests were performed using a sequence of automatically generated datasets containing one random generated normal distribution. The initial parameters of this distribution were determined using the cluster process described in the previous section and then passed to the parallel statistical optimizer process. The average time until convergence (if the algorithm converged at all) was 2.11 seconds. The worst and best case times were respectively 3.22 seconds and 2.05 seconds. A sequential version was also tested which, on average, took 3.58 seconds to reach convergence. The average speedup figure given these figures is 1.7.

5.4 Processor network topology

The decomposition methods used in the filter and cluster processes are the same. This method of decomposition can be easily mapped onto a ring topology (recall Figure 4.3 on page 23). The optimization process however uses a very different form of decomposition which requires a network configuration that can not directly be mapped onto the same ring topology (recall Figure 5.13). One way to solve this is to map three optimizer processes onto the same ring topology used by the filter and cluster processes, and put the fourth process on a fourth processor. A routing process is then needed to relay messages out to the fourth process. However, this was considered cumbersome and would unnecessarily complicate the system's design.

Figure 5.14 shows the topology that has been built for the CACE-II system. The network basically consists of a ring, but an additional "short-cut" connection has been provided to access the fourth optimizer process.

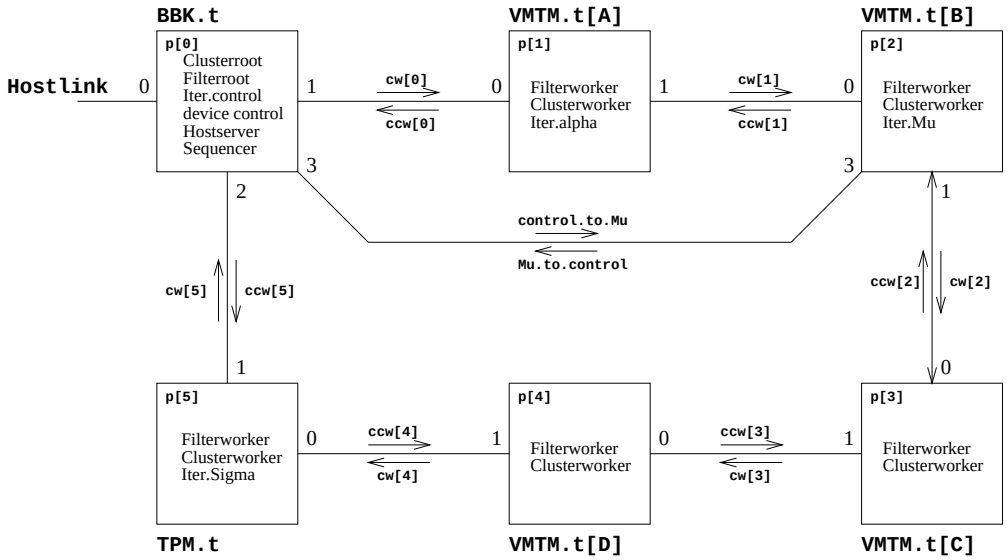

Figure 5.14: Processor network topology used in CACE-II.

5.5 The parallel data analyses pipeline

The parallel data analysis tools are configured into a "parallel data analysis pipeline" (see figure 5.15) so that results of an analysis process are passed on to the next process in the pipeline and immediately restarts. The first process in this pipeline reads its input from the raw data area accessed by the DMA data acquisition interface.

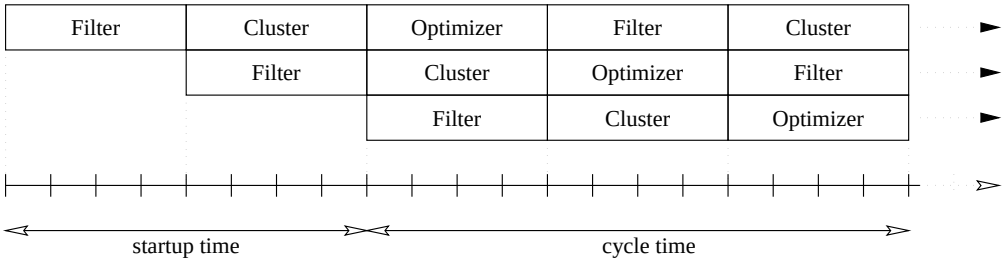

Figure 5.15: Execution model of an ideal data analysis pipeline.

When all processes in such a pipeline have at least started execution once (*startup time*) all processes run in parallel at the same time producing a steady throughput of data through the pipeline to the output. In order for such a structure to work properly however, all processes in the pipeline would have to take exactly the same time to execute to get the pipeline in balance. In practice this is not the case since some processes are finished sooner than others. Furthermore, since the execution times of the cluster and optimization algorithm are data content dependent, their execution time may vary constantly. The filter algorithm is not data dependent, so its execution time is constant.

5.5.1 The parallel digital analysis pipeline sequencer

The operation of the parallel data analysis pipeline is controlled by a process called the "sequencer". The sequencer controls the operation of each process, detecting the state each process is in and signaling each when a process should start executing. During an online experiment, the sequencer automatically guides each process through the pipeline process in such a way that the throughput of the pipeline is optimal.

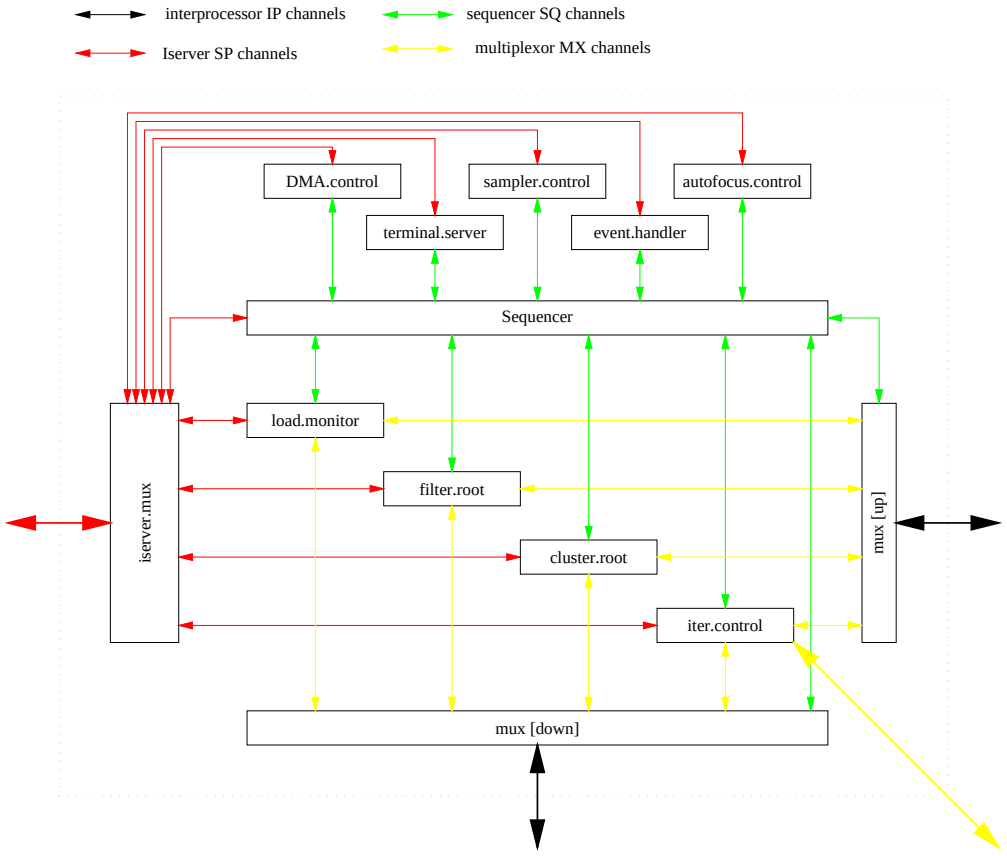

Figure 5.16: Process configuration on the root processor. The complete system is governed from this processor.

The sequencer plays a central role in the control of all software in the CACE-II system, as can be seen from Figure 5.16. The sequencer keeps track of the state each of the data analysis processes are in. The data analysis processes provide this information to the sequencer by sending messages at predefined positions during their execution. The sequencer sends confirmations in reaction to each of these messages to allow the analysis processes to proceed execution. This allows the sequencer to stall a process so that another process may have exclusive access to a resource in the system. For example, the filter process may not be allowed to store its resulting data while the clustering process is at the same time loading this data and spreading it over its worker processes. In this case the sequencer must stall the cluster process until the filter process has finished.

The sequencer is controlled from the user interface.

5.6 The CACE-II user interface

A user interface should allow a user to control an experiment and to see the results on screen whilst at the same time protecting him from any awkward details that are not of interest. The user interface is the part of CACE-II most users will see and work with.

The Tcl/Tk environment that was used to implement the CACE-II user interface provides an extremely powerful set of tools to create graphical user interfaces consisting of "widgets" (buttons, windows, menus, etc.) [48, 65]. Tcl/Tk is freely available for most common computer platforms and operating systems such as Unix, Windows or Macintosh systems so that programs are consistent and portable across these platforms. The CACE-II user interface exploits the network capabilities of Tcl/Tk to access the CACE-II embedded system via a TCP/IP oriented network. This allows the CACE-II system to be controlled from any location as long as there is a network connection.

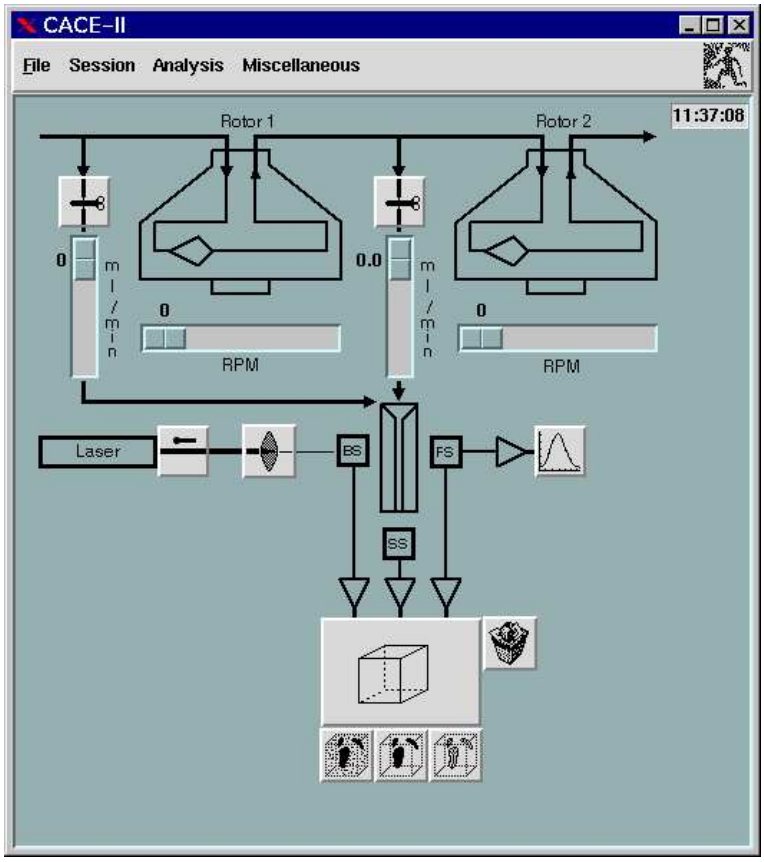

Figure 5.17: The CACE-II startup screen.

Figure 5.17 shows the main interface to the CACE-II system. An attempt was made to provide a clear representation of the devices under control, including the flow system, the rotors, the autofocus system and data analyses. This user interface has been used frequently during tests of the embedded system, with satisfying results.

Chapter 6

The autofocus subsystem

During a separation experiment it is essential that the optical detection system is properly aligned. A system has been built to facilitate some of the steps of this alignment process where previously expert knowledge was required. This automatic focus equipment (or autofocus equipment for short) has been developed as a separate subsystem that operates independently from processes on the CACE-II system. This chapter describes how the autofocus system works and how to use it during CACE-II experiments.

6.1 Introduction

The optical system in CACE-II consists of the following components (see also Fig. 6.1):

- a laser,
- a lens system consisting of two crossed cylindrical lenses ($f = 75$ mm and $f = 15$ mm),
- a flow chamber (5×5 mm quartz with a 0.5×0.5 mm channel in the center),
- the forward, side and back scatter detectors.

Of these, the laser must be manually adjusted so that the beam passes the lens system unobstructed (i.e. the beam must be at water level with respect to the lens system) and the FS detector must be manually positioned such that the beam hits a vertical "beamstop" in front of the detector. These manual adjustments need normally be performed only once just before the start of an experiment.

The position of the lens system and the flow chamber need adjustment for each fraction that is elutriated during an experiment as the particle trajectory through the channel may change with variations in the fluid flow rate [15]. Recall from Figure 3.3 (page 12) that elutriation of each fraction takes approximately 10 minutes and that data acquisition for each fraction commences when a few hundred cells leave the elutriator per second. This implies that the lens system and flow chamber must be in the new focuspoint before data acquisition commences.

To enable the components of the optical system to be positioned automatically, the lens system and BS detector are mounted on a movable table, controlled by a servo, allowing the lenses to be positioned over a range of 20 mm along the laser beam's path. Likewise, the flow chamber holder and SS detector are mounted on a movable table that allows the flow chamber to be positioned into the laser beam's path. Both servos are controlled by a VMEbus servo controller[1] allowing them to

[1] Themis TSVME-440, based on the Motorola 68010 processor.

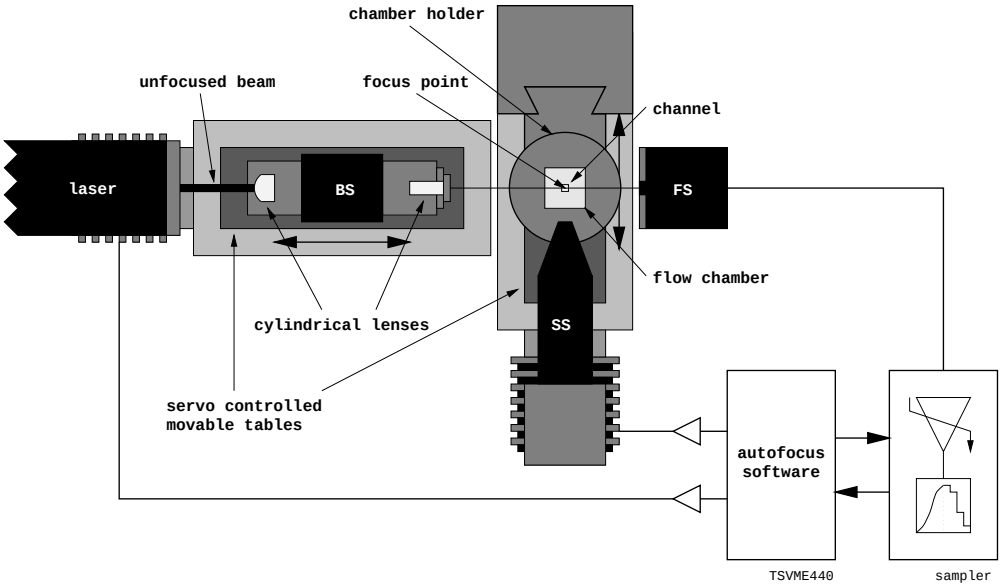

Figure 6.1: Configuration of the optical detection system.

be positioned with an accuracy of 1 μm [63]. In addition, a special purpose VMEbus card has been designed and built to amplify and sample signals from the FS detector. Based on these signals, the objective of the *autofocus* software is to automatically focus the laser beam onto the particles that flow through the channel of the flow chamber.

6.1.1 Properties of the focus point

The incident beam waist ω of the laser used in CACE-II is 1.9 mm. This beam must be focused on the particles that pass through the channel in the flow chamber such that no particles are missed (i.e. the beam may not be narrower than the width of the flow chamber's channel) and no more than one particle passes the beam at once (i.e. the height of the beam may not exceed the diameter of a particle). To achieve this, a lens system consisting of two crossed cylindrical lenses focuses the beam into a line of light to produce a "slit" of laser illumination through which the cells flow. An optical system using this technique is commonly called a *slit-scan cytometer* [15]. In our case the lenses have focal lengths of 75 mm and 15 mm and are 60 mm apart. The first lens "narrows" the beam in vertical direction while a second lens "flattens" the beam in horizontal direction into the flow chamber's channel (see Fig. 6.2). The beam waist radius ω_0 in the focus point is proportional to the focal length f [55, page 151]:

$$\omega_0 = \frac{4}{\pi}\frac{\lambda}{\omega}f. \tag{6.1}$$

In this case the wavelength of the laser $\lambda = 632.8$ nm, so from Eq. 6.1 we can deduce that the focused laser spot has a width of $\omega_x = \frac{4}{\pi}\frac{632.8\cdot10^{-9}}{1.9\cdot10^{-3}} \cdot 7.5 \cdot 10^{-2} = 32$ μm, and a height of $\omega_y = \frac{4}{\pi}\frac{632.8\cdot10^{-9}}{1.9\cdot10^{-3}} \cdot 1.5 \cdot 10^{-2} = 6.4$ μm. In practical situations the focused laser spot will be somewhat larger due to inevitable misalignment and impurities in the lenses. When moving away from the focus point, ω_x and ω_y of the laser spot will increase while the total integrated intensity of the spot remains the same.

The intensity of the laser beam inside the spot is Gaussian distributed [55]. The intensity I_{beam}

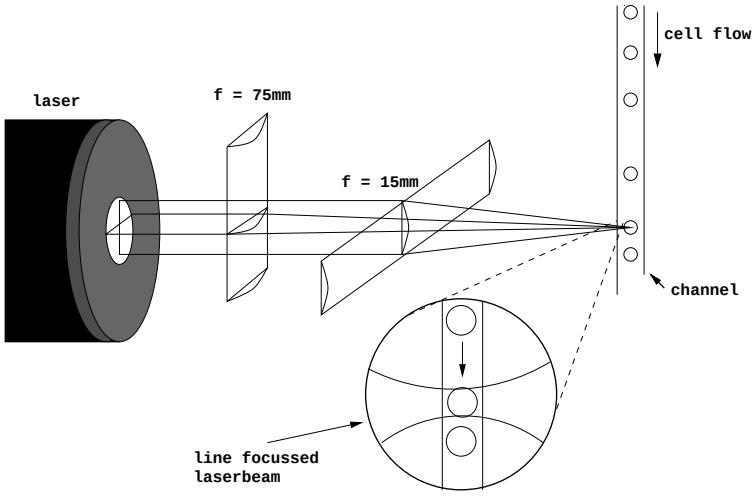

Figure 6.2: The lens system focuses the laser beam using two crossed lenses.

at any point in the spot can be calculated using Eq. 6.2:

$$I_{beam} = E_0^2 \left(\frac{1}{1+4Y_x^2} e^{-\frac{\left(\frac{x}{\omega_x}\right)^2}{1+4Y_x^2}} \right) \left(\frac{1}{1+4Y_z^2} e^{-\frac{\left(\frac{z}{\omega_z}\right)^2}{1+4Y_z^2}} \right), \tag{6.2}$$

where

$$Y_x = y\frac{\lambda}{2\pi\omega_x^2} \qquad Y_z = y\frac{\lambda}{2\pi\omega_z^2}.$$

Here E_0 is the intensity of the incident beam, x and z define the position inside the spot with the center of the spot taken as the origin.

The FS pulse profile as a function of time resulting from a particle traveling through the focus point at a certain speed is expressed by [15]:

$$I_{FS}(vt) = \int\int\int C(x, vt-\xi, z)H(\xi)dxd\xi dz \tag{6.3}$$

where v is the velocity of the particle along the y axis (which is approximately the same as the flow rate through the channel), $H(y)$ the slit irradiance profile as a function of y and $C(x, y, z)$ the distribution of light emission sites in the particle. It can be shown that $H(y)$ and, for homogeneous particles, $C(x, y, z)$ are both Gaussian functions [26]. Convolution of two Gaussian functions, as in Eq. 6.3, yields a new Gaussian function with a variance that equals the sum of the variances of the original functions. Therefore the width of $I_{FS}(vt)$ can be expressed as:

$$width(I_{FS}(vt)) = \sqrt{width(C(x, y, z))^2 + (width(I_{beam}))^2}. \tag{6.4}$$

Realistic light scattering simulations performed within the Parallel Scientific Computing and Simulation Group have shown that for biological particles this is indeed the case [22].

Eq. 6.3 shows that the signals obtained from the FS detector as light scatters from these particles is also Gaussian distributed [15, 21, 41, 42]. Moreover, as the focus point moves closer to the track followed by the particles, the signals obtained from the FS detector will become higher and narrower (see Eq. 6.4). So, at the focus point the value at mean from the obtained signals will therefore be maximal while the variance is minimal.

6.2 The autofocus process

From what is presented in the previous sections it is clear that the autofocus process should fulfill two tasks: position the flow chamber's channel in the path of the laser beam and position the lens system so that the focus point is situated at the particles flowing through the flow chamber. However, if these two tasks are to be based on signals derived from the FS detector as particles pass through the laser beam, we are faced with an interesting dilemma: if the lenses are out of focus, the signals from the FS detector will be too low in intensity to make it possible to locate the flow chamber's channel, but at the same time it is not possible to focus the lenses if the flow chamber's channel is not placed in the path of the laser as it is simply not possible to register any particles.

Through experiments we have been able to develop a reliable method that allows us to locate the flow chamber's channel using the refractive properties of the quartz material from which the flow chamber is built. The resulting position for the flow chamber is not optimal, yet it provides a good starting point from which a refinement is possible.

The autofocus process described in the following subsections is split into three phases: the first phase manoeuvres the channel within the flow chamber in the path of the laser beam, the second phase adjusts the position of the lens system so that pulses are registered caused by particles moving through the channel, the third and final phase optimizes the positions of the flow chamber and lenses obtained from the first and second phase.

6.2.1 Locating the flow chamber's channel

The flow chamber is made from quartz, a transparent material that has the advantage over conventional glass that it has virtually no impurities and its refraction index is similar to that of water. Experiments have shown that, provided the amplification of the FS signal is high enough, it is

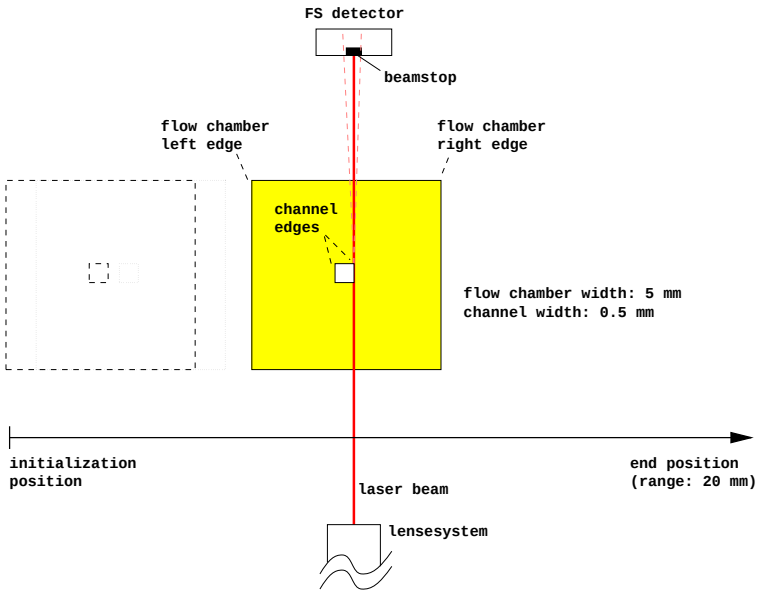

Figure 6.3: Refraction of light on the flow chamber as it passes through the laser beam.

possible to discriminate the edges of the flow chamber and the channel by looking at amplitude variations of the FS signal as the chamber moves through the laser beam at a constant speed (see

Fig. 6.3). This phenomenon is probably caused by refractive effects when the laser beam "graces" the edges of the flow chamber and channel, and from diffractions from the corners of the channel [27]. Moreover, it can be shown that it is important in this case to use *servo* motors to control the tables and not stepper motors (which are more suitable in situations where discrete spacing intervals are required). This can be explained from the fact that the pre-amplifiers used for the scatter detectors are AC-coupled, meaning that only *changes* in amplitude are amplified. If we were to use discrete stepping motors, the distinct features of the flow chamber would not be so apparent[2].

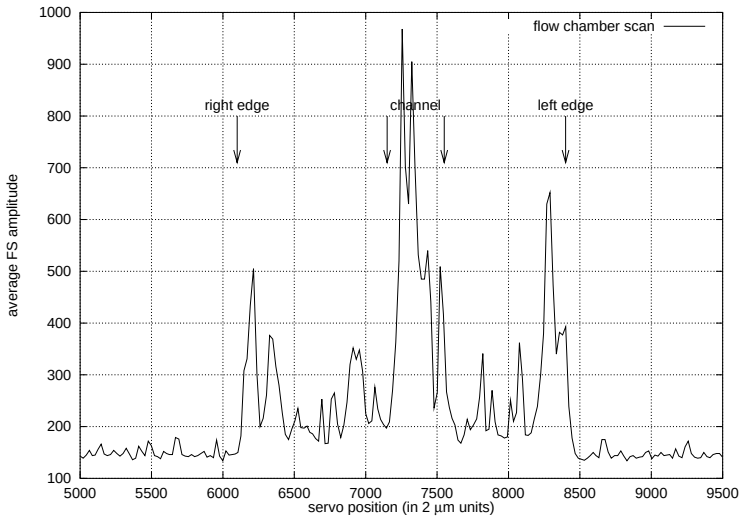

Figure 6.4: FS signal response of flow chamber moving through laser beam.

A typical example of the FS signal response as the flow chamber moves through the laser beam is shown in Fig. 6.4. In this case the amplification of the FS signal was set to 100×, the flow chamber moved through the laser beam at a speed of approximately 0.5 mm/sec and an average over 10 samples per units of 2 μm were taken. Using knowledge of the physical dimensions of the flow chamber and the channel, both the edges and the channel within the flow chamber can be clearly distinguished from this figure. The channel locator algorithm used in this stage of the autofocus process filters the signal obtained from this scan and then determines the first order derivatives. Based on prior knowledge of the physical dimensions of the flow chamber the algorithm then determines the most likely candidates for the channel position upon which the flow chamber is moved to the most likely candidate.

Using this method, an incorrectly inserted (i.e. slightly rotated) flow chamber shows up as a curve where the slope of the tops is diagonal rather than horizontal (data available on request). The curve is presented on the autofocus user interface which allows an analyst to recognize this situation and take appropriate action.

Elaborate tests have been performed on the implemented algorithm, including situations where artificial noise was imposed by banging ones fists on the table, yet the algorithm was still able to correctly detect the channel.

[2]Experimental data that shows this is available on request.

6.2.2 Focusing the lens system

The channel localization routine described in the previous section orients the flow chamber such that the channel is in the path of the laser beam. As the channel is 500 μm wide while the theoretical width of the laser spot in the focus point is 32 μm (see Eq. 6.1), the position of the chamber can hardly be sufficient. Furthermore, the position of the lens at this point is completely arbitrary. Given the length of the laser dot at the focus point of 6.4 μm, finding the correct position for the lens is like searching for a needle in a haystack.

However, from the flow chamber position obtained with the first phase, we are now able to refine the positions of both the flow chamber and the lens system by analyzing the signals from the FS detector caused by particles flowing through the channel. To limit the time required to focus the lenses, the lens focus process is split up into two phases; the first *coarse focus* phase intends to decrease the initial search area in which the focus point lies by means of relatively fast, but coarse method. The second *fine focus* phase iteratively refines this search area towards the final flow chamber and lens positions. These two steps are described in detail in the following sections.

Coarse focus

The first *coarse focus* step moves the lens system from its initialization position towards the flow chamber, meanwhile registering the signals from the FS detector, until Gaussian type signals are registered.

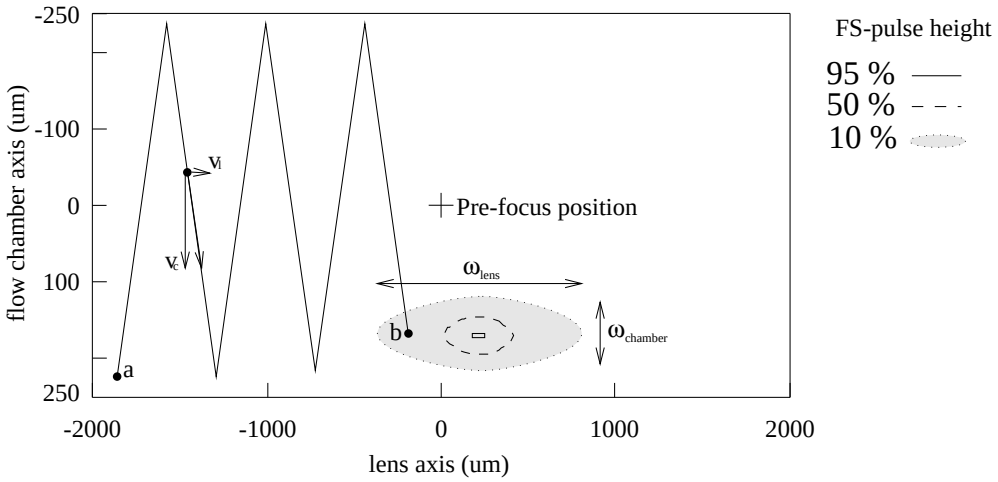

Figure 6.5: The coarse focus algorithm.

The algorithm is illustrated in Figure 6.5. From the position that was found using the channel localization algorithm described in the previous subsection, the area in which the focuspoint resides is defined by the width of the flow chamber's channel (500 μm) and the range over which the lenses can be moved (4000 μm). Assuming that pulses at 10% of maximum intensity can be detected, the area the coarse focus algorithm attempts to find is (using Eq. 6.2) $\omega_{chamber} = 100\mu$m by $\omega_{lens} = 1200\mu$m. The coarse focus algorithm searches this area using a criss-cross search pattern in which the velocities of the servos are constantly adjusted so that this area can not be missed. During this search, the signals from the FS detector are constantly monitored for Gaussian type signals. Once these are detected, the algorithm stops.

At this point the lens system is sufficiently closer to the focus point to further refine both the flow chamber and the lens system locations.

Fine focus

The second and final *fine focus* step employs an optimization algorithm which is similar to the familiar *steepest-descent* method to reduce the dimensions of the search area in which the focus point must be located.

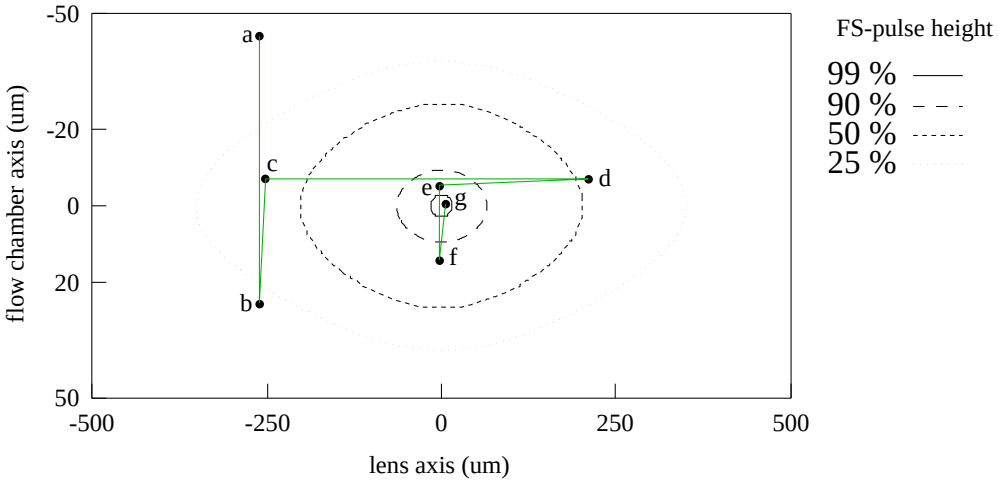

Figure 6.6: The fine focus algorithm.

This algorithm is illustrated in Figure 6.6. From the position that was found using the coarse focus step (point **a** in this figure), the flow chamber servo moves at a constant speed into an arbitrary direction while constantly monitoring the signals from the FS detector. When the mean amplitude of these signals decreases during this movement, the direction of the servo is reversed. This movement continues until the mean amplitude has reached a maximum and decreases again. When this is detected (point **b**), the servo is moved back to the position where the maximum amplitude was found (point **c**) and the search area is reduced. Next, the process is repeated for the lens servo. The algorithm stops when the search area descends a preset value.

Figure 6.7 shows an example of the convergence of the fine focus algorithm towards the focus point. The figure shows the decrease in dimensions of the search areas after each iteration and the trajectory of the estimated focus point which is located in the center of each search area.

6.3 CACE-II implementation details

Fig. 6.8 shows a more detailed picture of the components involved in the autofocus subsystem as it is implemented on the CACE-II system. Three VME modules are involved in the autofocus subsystem; the BBK-V2 transputer board, the TSVME-440 servo controller and the dedicated samplecard. The samplecard also contains the buffers that are required to interface the TSVME-440 to the servo motors. The TSVME-440 is capable of driving four axes simultaneously but only two of these are used in the CACE-II autofocus subsystem.

While the BBK-V2 is the main processor in the analysis system, in the CACE-II autofocus subsystem it plays an intermediate role between the user interface and the central autofocus software which runs on the TSVME-440 servo controller. Two processes in particular are involved with the autofocus software;

- `autofocus control` forms an interface between the autofocus software and the user interface.

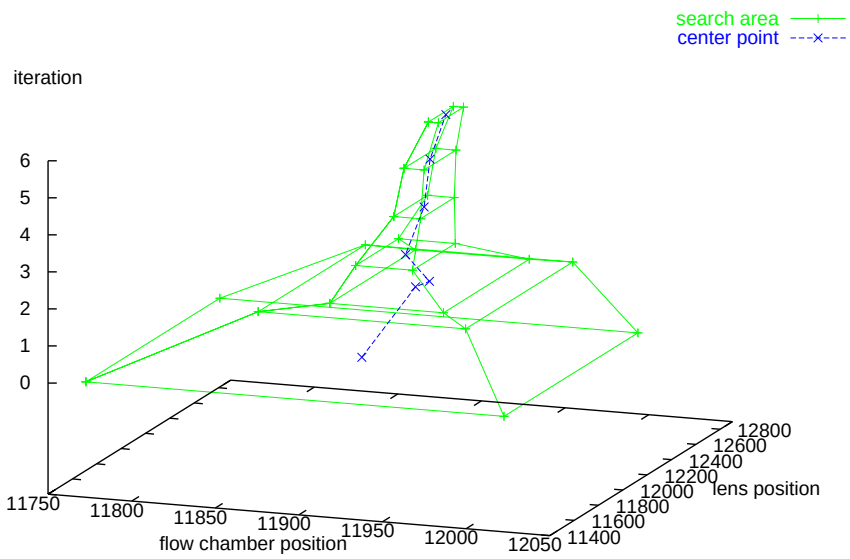

Figure 6.7: Example of convergence in the fine focus algorithm; after 6 iterations, the search area is reduced from 250 μm over the flow chamber axis by 1100 μm over the lens axis to an area of 6 μm by 206 μm.

Commands issued by the user in the user interface are "relayed" to the autofocus software while the progress of the autofocus software is visualized back onto the user interface.

- `sampler control` allows the user to monitor signals from the FS detector, but this process performs a more vital role in the autofocus subsystem which is described in a moment.

Data communication between these two processes and the autofocus software takes place via a shared memory area on the TSVME-440 which is accessible both by the processor on this board (a Motorola 68010) and *master* devices elsewhere on the VMEbus. The BBK-V2 is a so-called VMEbus master, meaning that it can initiate data transfers over the VMEbus [1]. The TSVME-440 is a VMEbus *slave* which means that it can only *be* accessed by bus masters; it can *not* read or write values from a device elsewhere on the VMEbus, including the samplecard. As the autofocus software running on the TSVME-440 requires samples from the samplecard to do its job, the `sampler control` process on the BBK-V2 has been implemented to act as a mediator between the autofocus software on the TSVME-440 and the samplecard. Each time the autofocus software requires a sample from the samplecard, it sets the desired sampler settings in the shared memory area and signals the `sampler control` process on the BBK-V2 by generating a level 3 interrupt request (IRQ). The `event handler` process on the BBK-V2 directs this interrupt to the `sampler control` process which initializes the samplecard with the requested settings. As soon as the samplecard has taken a sample from the input signal it signals the `sampler control` process by generating a level 6 IRQ which copies the sample from the samplecard to the shared memory area. The autofocus software is then signaled of the availability of this sample by a level 5 interrupt.

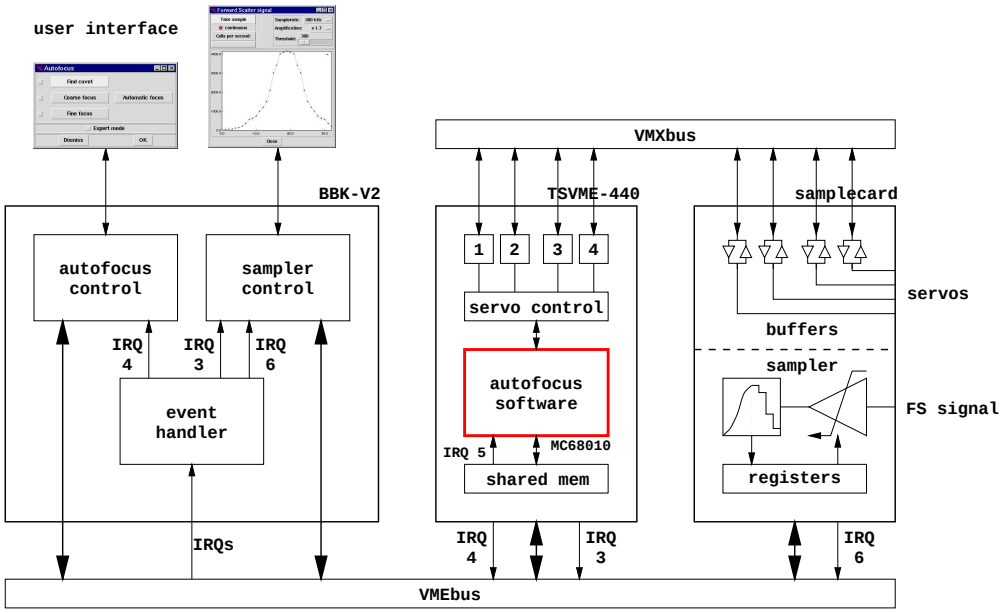

Figure 6.8: Detailed configuration of hard- and software in the autofocus subsystem.

6.4 Conclusions

In this chapter a subsystem has been described of the CACE-II system which aids in aligning the optical system, based solely on signals obtained from the FS detector. Although further experiments still have to be performed on the accuracy and speed of the implemented algorithms, the results thus far look very promising. The reduction of the problem as a whole into a channel localization, coarse focus and fine focus algorithm have proven to be quite elegant. Indeed, it is believed that the first two focus steps will only need to be performed at the start of a CACE-II experiment for the initial alignment of the optical system. During the elutriation of a new fraction, corrections for the new trajectory of the particles can then probably be performed by the fine focus algorithm alone.

6.4.1 Acknowledgments

I gratefully acknowledge the cooperation of the following people in the development of the autofocus subsystem:

- Ruud Veldhuizen (Electronics department, University of Amsterdam) for designing and debugging the samplecard hardware and motor buffers,
- Maarten Kreuger (Hogeschool Alkmaar) for debugging the samplecard hardware and the development of the first version of the autofocus library,
- Steven Chan (Hogeschool Amsterdam) for his experiments with the samplecard and the autofocus library and the first implementations of autofocus algorithms,
- Bart Meeuwissen (Hogeschool Eindhoven) for his implementation of a reliable channel localization routine and his ideas on a robust algorithm to focus the lens system,
- Michael Kleindieck (Hogeschool Alkmaar) for his implementation of the lens focus routines and the autofocus user interface.

Chapter 7

Conclusions and recommendations

In this thesis, I described the design issues involved into the development of an embedded system for the control of a cell separation experiment. Although the system still needs further refinement and extensive tests in the laboratory environment for which it was aimed, initial experiments have shown that the prospects are promising.

The transputer architecture and the `occam` programming language that were used in the development of the embedded system provided the building blocks that allowed the construction of an embedded system that is capable of rapid on-line data analyses while controlling an on-line experiment. A flexible user interface enables easy access to the embedded system and provides on-line monitoring primitives.

An autofocus subsystem was designed and implemented that is capable of aligning the optical system where previously expert knowledge was required.

Some important issues still remain however, which I shall address in the following sections.

7.1 Cluster analysis

I have not been able to refine the cluster analyses process in such a manner that in all cases the correct number of populations are detected. As was noted in Section 5.3, this is a very important issue in the correct analyses of the acquired data. Although more and more literature becomes available on the subject of cluster analyses, I have seen none that are capable of unsupervised clustering.

Further investigation into this subject is required. Possibly, successful algorithms may be found in fuzzy clustering techniques, neural-networks or new parameterized versions of the Hoshen-Kopelman algorithm [28].

7.2 Portability

Although `occam` provides a solid basis for the construction of large parallel systems, its use over the last years has come to an almost standstill. Furthermore, with the demise of INMOS quickly after the introduction of the infamous T9000 transputer, the further development of `occam` is in serious danger. The implication for the CACE-II system is that this could mean that the software that has been developed can not easily be ported to other parallel computing architectures.

The occam compilers that target other architectures will need to be carefully evaluated to see whether it is possible to port the CACE-II software to a new platform. Also, the devices built by other microprocessor manufacturers (such as the Motorola's PowerPC and Digital's Alpha) may provide alternatives to the transputers used in CACE-II in the event that a new architecture must be chosen when the transputer is no longer readily available.

7.3 Recommendations for future work

7.3.1 Automatic control of the flow system

The current CACE-II system provides no control over the fluid flows through the FCM equipment. However, this flow control can be implemented using the remaining servo channels on the TSVME-440 module. The sample card that has been implemented for the autofocus subsystem allows the number of cells that pass through the flow chamber to be determined. Using this feature and the remaining channels on the servo controller, a crude flow control system could be built.

7.3.2 The DMA interface

In the current design of the DMA interface, the analog signals from the three light scattering detectors are first fed to three external amplifiers. Each time a top is detected in the forward scatter signal, the three signals are sampled and converted into 6 bits values resulting in a 18 bit pattern. This pattern then travels over a flatcable into the DMA module which then uses this pattern as a memory address on the VME bus to access a 128 kiloword (512 kilobyte) memory area to build the histogram of pattern occurrences. The DMA module creates this histogram using a read-modify-write scheme so that for each detected signal two accesses are required over the VME bus (one read, one write).

Most of the logic on the DMA interface is built from PALs that date back to 1985 or so. The current state-of-the-art in electronics allow the analog-to-digital converters, the memory and perhaps even the amplifiers to be placed on a single module. When the memory on this module is made dual-ported, the DMA processor can continue to operate while processors running data analyses software make copies of the data locally. Placing all these components on a single module results in a compact, self-contained and generic acquisition module.

Alternatively, a two channel DMA processor [1] could be used where one channel performs the read-modify-write cycles that create the pattern histograms while the second channel could be instructed by the data analyses software to create a copy of the data onto the memory of a local processor.

The proposed design has the following advantages;

- A 'centralized' design in which the ADCs, the memory and amplifiers are integrated into one module is more logical and maintainable compared to the current multi-module design. Provided that the structure of such a module is carefully designed, it should also be possible to add more channels or to increase the resolution of the converters. Todays modern programmable logic may provide a key to this flexibility [10].

- Data acquisition is performed locally on the module so that there need not be VME bus accesses for each detected signal. Instead, the data analyses software (or another DMA channel) copies the data when it is required.

[1] The MC68440 and MC68450 devices from Motorola look like good candidates. For more information on Motorola products, see http://motserv.indirect.com/.

- In the case that a two channel DMA processor is used, copying the data onto a local processor via DMA control is much faster compared to a software controlled copy.

7.3.3 List mode data acquisition

The advantages and disadvantages of both histogramming and list mode acquisition have been discussed in Section 3.1.1 (page 9). One of the disadvantages that was mentioned on list mode acquisition was the large amounts of memory that may be required. Today however, memory is available in high density modules that are relatively cheap. In addition, list mode acquisition is quite common in other research areas so that generic modules have become easily available.

List mode data can always be converted to histogrammed data (but not vice versa), so the current data analysis software can still be used. Furthermore, software has been developed by researchers in the area of flow cytometry for the analysis of list mode data which can also be applied here [46, 53]. Note however that list mode data analyses programs are always non-deterministic in time behaviour due to the variable sized datasets. This can have a negative influence for their application in embedded systems, but should be investigated nevertheless.

7.3.4 The optical detection system

Detectors and lasers have continued to decrease in size and cost while increasing in performance over the last years. Diode lasers in particular are currently available at wavelengths and optical power comparable to the bulky laser used in CACE-II [16]. This new technology can be exploited to reduce both size and cost for this type of embedded systems.

An additional advantage of this type of system is that the complete optical detection system can be contained into a solid-state construction in which all components can be fixed. This could free the device of the need for an autofocus system.

Bibliography

[1] American National Standards Institute (ANSI). *The VMEbus Specification, ANSI/IEEE STD1014-1987, IEC 821 and 297*, 1987.

[2] M.R. Anderberg. *Cluster Analysis for Applications.* Academic Press, NY, 1973.

[3] H.F. Bao, H.C. den Harink, E.S. Gelsema, and A.W.M. Smeulders. Automated white blood cell classification revisited. *Medical Informatics*, 12(1):23–31, 1987.

[4] Geoff Barret. occam 3 reference manual. Technical report, INMOS Limited, 1992. Draft - March 31.

[5] G. Barrett, M. Goldsmith, G. Jones, and A. Kay. The meaning and implementation of pri alt in occam. In Charlie Askew, editor, *Occam and the Transputer- Research and Applications OUG-9*, pages 37–46, Amsterdam, 1988. IOS Press.

[6] D.J. Beckett. *occam Design Tool User Manual, for ODT version 1.01.* See http://www.hensa.ac.uk/parallel/.

[7] D.J. Beckett and P.H. Welch. A strict occam design tool. In C.R. Jesshope and A. Shafarenko, editors, *Proceedings of UK Parallel '96*, pages 53–69, London, UK, July 1996. BCS PPSIG, Springer-Verlag. ISBN 3-540-76068-7.

[8] M. Boasson. Control systems software. *IEEE Transactions on Automatic Control*, 38(7):1094–1107, 1993.

[9] R.N. Bracewell. Note on Hermitian data. Private communication with P.M.A. Sloot.

[10] Steve Casselman. FPGAs for custom computing machines. Paper presented at IEEE workshop, Napa, California, April 5-7 1993.

[11] J.E. Cooling. *Software Design for Real-time Systems.* Chapman and Hall, 1991. ISBN 0-412-34180-8.

[12] Mark Debbage, Mark Hill, Sean Wykes, and Denis Nicole. Southampton's portable occam compiler (spoc). Draft paper presented at WoTUG/OUG 17 conference at Bristol. See `http://www.hensa.ac.uk/parallel/occam/compilers/spoc/spoc.oug17.ps.Z`, 1994.

[13] H.J. Denuell. *BBK-V2: Transputer - Bus - Bridgehead for Interfacing VME-systems to MEGAFRAME Transputer Modules, technical documentation.* Parsytec GmbH, version 2.4 edition, November 1988.

[14] E.W. Dijkstra. Cooperating sequential processes. In F. Genuys, editor, *Programming Languages*, pages 43–112. Academic Press, NY, 1968.

[15] Marvin A. Dilla, Pillip N. Dean, Ole D. Laerum, and Myron R. Melamed, editors. *Flow Cytometry: Instrumentation and Data Analysis.* Analytical Cytology series. Academic Press Inc, UK, 1985.

[16] R.M.P. Doornbos, E.J. Hennink, C.A.J. Putman, B.G. De Grooth, and J. Greve. White blood cell differentiation using a solid state flow cytometer. *Cytometry*, 14:589–594, 1993.

[17] B. Everitt. *Cluster Analysis (2nd edition).* Heinemann Educational Books, UK, 1980.

[18] C.G. Figdor et al. Isolation of large numbers of highly purified lymphocytes and monocytes with a modified centrifugal elutriation technique. *Journal of Immunological Methods*, 40(275), 1981.

[19] C.G. Figdor et al. Rapid isolation of mononuclear cells from buffy coats prepared by a new blood cell separator. *Journal of Immunological Methods*, 55(221), 1982.

[20] A.D. Gordon. *Classification.* Chapman and Hall, NY, 1981.

[21] Gérard Gousbet, Gérard Gréhan, and Bruno Mahue. Localized interpretation to compute all the coefficients g(n,m) in the generalized Lorenz-Mie theory. *Optical Society of America*, 7:998–1007, June 1990.

[22] Michel D. Grimminck. Computer simulation of light scattering of small particles in focused laser beams. Master's thesis, University of Amsterdam, September 1997.

[23] Victor Hasselblad. Estimation of parameters for a mixture of normal distributions. *Technometrics*, 8(3):431–444, August 1966.

[24] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall International, UK, 1985.

[25] C.A.R. Hoare, editor. *Developments in concurrency and communication.* University of Austin at Texas, Year of Programming Series. Addison-Wesley, 1990.

[26] A.G. Hoekstra. *Computer Simulations of Elastic Light Scattering (Implementation and Applications).* PhD thesis, University of Amsterdam, Amsterdam, 1994.

[27] Alfons Hoekstra. Note on light diffraction. Private communication.

[28] J. Hoshen and R. Kopelman. Percolation and cluster distribution. i. multiple cluster labeling technique and critical concentration algorithm. *Phys. Rev.*, B14:3438, 1976.

[29] Kai Hwang. *Advanced Computer Architecture: Parallelism, Scalability, Programmability.* McGraw-Hill, Inc., 1993. ISBN 0-07-031622-8.

[30] INMOS Limited. *occam 2 toolset user manual - part 1 (User guide and tools)*, 72 TDS 275 02 edition. d7205 toolset.

[31] INMOS/SGS-THOMSON Microelectronics Limited. *T2, T4, T8 and T9 transputer family datasheets*, 1995/1996. See http://www.st.com/stonline/books/.

[32] A.K. Jain and R.C. Dubes. *Algorithms for Clustering Data*. Prentice-Hall International, UK, 1988.

[33] Geraint Jones. On guards. In *Parallel Programming of Transputer Based Machines*, Amsterdam, September 1987. IOS Press.

[34] Geraint Jones. Carefully scheduled selection with ALT. occam user group newsletter no.10, January 1989.

[35] B. Charles Peters Jr. and Homer F. Walker. An iterative procedure for obtaining maximum-likelihood estimates of the parameters for a mixture of normal distributions. *Society for Industrial and Applied Mathematics*, 35(2), September 1978.

[36] Brian W. Kernighan and Dennis M. Ritchie. *The C programming language (second edition)*. Prentice-Hall International, 1988. ISBN 0-13-110370-9.

[37] S. Krishnan, K. Samudravijaya, and P.V.S. Rao. Feature selection for pattern classification with gaussian mixture models: a new objective criterion. *Pattern Recognition*, 17:803–809, 1996.

[38] INMOS Limited. *occam 2 Reference Manual*. Prentice-Hall International, UK, 1988.

[39] INMOS Limited. *Transputer Instruction Set (a compiler writer's guide)*. Prentice-Hall International, 1988. ISBN 0-13-929100-8.

[40] Ja-Chen Lin and Wu-Ja Lin. Real-time and automatic two-class clustering by analytical formulas. *Pattern Recognition*, 29(11):1919–1930, 1996.

[41] James A. Lock and Gérard Gousbet. Rigorous justification of the localized approximation to the beam-shape coefficients in generalized Lorenz-Mie theory. II: Off-axis beams. *Optical Society of America*, 11:2516, 1994.

[42] James A. Lock, Gérard Gousbet, and Gérard Gréhan. Partial-wave representations of laser beams for use in light-scattering calculations. *Applied Optics*, 34:2133, 1995.

[43] Juergen Loewenhag. *VMTM: VME Multi Transputer Module, technical documentation*. Parsytec GmbH, version 1.1 edition, October 1987.

[44] D. May. Compiling occam into silicon. chapter 3, in [25].

[45] Peter Moller-Nielsen and Ole Caprani. Replacing an occam process by a chip. In P. Fritzson and L. Finmo, editors, *Parallel Programming and Applications*, pages 396–404. IOS Press, 1995.

[46] Robert F. Murphy. Automated identification of subpopulations in flow cytometric list mode data using cluster analysis. *Cytometry*, 6:302–309, 1985.

[47] P. Welch (University of Kent), Cook (University of Keele), D. May (INMOS Architecture Group), et al. occam-for-all project. See `http://www.hensa.ac.uk/parallel/occam/projects/occam-for-all/`.

[48] John K. Ousterhout. *Tcl and the Tk Toolkit*. Addison Wesley, 1994. ISBN 0-20-163337-X.

[49] John K. Ousterhout. Why threads are a bad idea (for most purposes), January 25 1996. See `http://www.sunlabs.com/people/john.ousterhout/`.

[50] University of Amsterdam Parallel Scientific Computing Group. Note on transputer floating point performance. Based on experience.

[51] G.M. Reed and A.W. Roscoe. A timed model for communicating sequential processes. In *ICALP'86, Lecture Notes on Computer Science*, pages 314–323. Springer-Verlag, LNCS 226, 1987.

[52] G.C. Salzman, J.M Crowell, J.C. Martin, T.T. Trujillo, A. Romero, P.F. Mullaney, and P.M. LaBauve. Cell classification by laser light scattering: Identification and separation of unstained leukocytes. *Acta Cytologica*, 19:374–377, 1975.

[53] Tom C. Bakker Schut, Bart G. De Grooth, and Jan Greve. Cluster analysis of flow cytometric list mode data on a personel computer. *Cytometry*, 14:649–659, 1993.

[54] Ravi Sethi. *Programming Languages (concepts and constructs).* Addison Wesley, 1990. ISBN 0-201-10365-6.

[55] P.M.A. Sloot. *Elastic light scattering from leukocytes in the development of computer assisted cell separation.* PhD thesis, University of Amsterdam, Amsterdam, 1988.

[56] P.M.A. Sloot, M.J. Carels, P. Tensen, and C.G. Figdor. Computer assisted centrifugal elutriation I:1 detection system and data acquisition equipment. *Computer Methods and Programs in Biomedicine*, 8:179, 1987.

[57] P.M.A. Sloot and C.G. Figdor. Elastic light scattering from nucleated bloodcells: Rapid numerical analysis. *Applied Optics*, 25:3559, 1986.

[58] P.M.A. Sloot, P. Tensen, and C.G. Figdor. Spectral analysis of flow cytometric data: Design of a special purpose low-pass digital filter. *Cytometry*, 8:545, 1987.

[59] P.M.A. Sloot, E.H.M. van der Donk, and C.G. Figdor. Computer assisted centrifugal elutriation II: Multiparametric statistical analysis. *Computer Methods and Programs in Biomedicine*, 27:37, 1988.

[60] M.J. Symons. Clustering criteria and multivariate normal mixtures. *Biometrics*, pages 35–43, March 1981.

[61] Andrew S. Tanenbaum. *Operating Systems (design and implementation).* Prentice-Hall International, 1987. ISBN 0-13-637331-3.

[62] L.L.W.M. Terstappen, B.G. De Grooth, K. Visscher, F.A. van Kouterik, and J. Greve. Four-parameter white blood cell differential counting based on light scattering measurements. *Cytometry*, 9(39), 1988.

[63] Themis Computer. *TSVME 440 axis controller board, user's manual*, DOCA246-03 edition.

[64] M. Vyskozil. *TPM-MAC: Starterkit Module for Apple Macintosh II, technical documentation.* Parsytec GmbH, version 1.3 edition, December 1989.

[65] Brent B. Welch. *Practical Programming in Tcl and Tk (second edition).* Prentice-Hall International, 1997. ISBN 0-13-616830-2.

[66] N. Wirth. *Programming in Modula II.* Springer Verlag, NY, 1983.

[67] S.J. Young. *Real time languages: design and development.* Ellis Horwood Limited, UK, 1982.