MASTER

Porting and improving the performance of contactless vital sign monitoring algorithms on constrained multi-core systems

Kruger, M.G.

Award date:
2018

Link to publication
Porting and Improving the Performance of Contactless Vital Sign Monitoring Algorithms on Constrained Multi-Core Systems

Master’s Thesis

Michael Kruger
Master of Science in Embedded Systems

Supervisors:
R. J. Bril (TU/e),
G. M. Kersten (Philips Innovation Services),
R. P. W. Springer (Philips Innovation Services)

Eindhoven, 22nd January 2018
Abstract

Contactless Vital Sign Monitoring is a promising technology that will make vital sign monitoring possible in situations such as neonatal or burn care, where traditional vital sign monitoring is problematic. Research into this technology has produced several algorithms capable of measuring heart rate and respiration using standard cameras. To evaluate and demonstrate these algorithms a Reference Framework has been developed in C++ that embeds the algorithms for execution. This Reference Framework is executed on powerful Windows laptops where performance requirements are easily met.

Now the need arises to execute these algorithms on more constrained systems in order to make embedded commercial use viable. This report presents the porting and optimization of the Reference Framework on a constrained Linux based system. The correctness of the port is validated and the performance of a motion based respiration algorithm is analysed. This analysis showed CPU underutilization and underperformance of the algorithm. To be able to better understand why these issues are seen, as well as guide optimizations, a generic methodology is developed which investigates how these complex parallel applications are scheduled and executed by the operating system.

This methodology called the Extended Thread State Analysis consists of analysing the temporal state of individual threads to identify the bottlenecks and issues preventing efficient utilization of computational resources. The methodology allows visualization and quantification of thread overheads and detection of problematic patterns. By decomposing the lifetime of threads into their states and analysing the state occupation times and transitions issues, such as scheduling priorities, synchronization, context switching overhead, bugs and cache stalls, can be identified. As a use case, this methodology is applied to the respiration algorithm. With the help of the method and its analysis several issues were identified. These issues were fixed and optimizations implemented that increased CPU utilization from 72% to 94% and improved throughput performance by more than 45%.
Acknowledgements

I would like to thank my supervisors Ger Kersten and Rink Springer at Philips Innovation Services for their constant support, feedback and motivation. The 8 months I spent at Philips Innovation Services has been an incredible rewarding and fun experience. I have learned a lot about operating systems (and what it might be like to write your own), C++ and Agile development.

I would like to thank my TU/e supervisor, R. J. Bril, for the quick response to my emails, critical feedback and fruitful discussions. It was a pleasure having him as my supervisor. I would also like to thank R. H. Mak for his initial support in helping me to get started and focused on my assignment.

I would also like to thank the members of the Contactless Monitoring group in Philips Research and Philips Innovation Services, possibly the smartest room of people I have ever been in. Their support and willingness to answer any of my questions is greatly appreciated.

I would like to thank my friends for their support and encouragement.

Finally, I would like to thank my family who have always been there and without them none of this would be possible.
# Contents

Abstract ........................................................................................................... i
Acknowledgements ......................................................................................... ii

1 Introduction .................................................................................................. 1
  1.1 Problem Description ................................................................................. 1
  1.2 Goals ........................................................................................................ 2
  1.3 Research Questions ................................................................................... 2
  1.4 Approach ................................................................................................... 3
  1.5 Contributions ............................................................................................. 4
  1.6 Outline ....................................................................................................... 5

2 Background and Related Work .................................................................... 6
  2.1 Application Domain .................................................................................. 6
  2.1.1 Contactless Vital Sign Monitoring ....................................................... 6
  2.1.2 Motion Based Respiration (AutoROI) ................................................ 7
  2.1.3 Respiration Requirements ................................................................... 9
  2.2 Reference Framework ............................................................................. 9
  2.2.1 Purpose and Functionality ................................................................... 9
  2.2.2 Modelling the Reference Framework .................................................. 10
  2.3 Target Hardware ...................................................................................... 11
  2.4 Related Work ............................................................................................ 13
  2.5 Background Knowledge .......................................................................... 14
  2.5.1 Parallelism Concepts ......................................................................... 14
  2.5.2 Linux Thread Scheduling ................................................................. 15
  2.5.3 Profiling Tools ..................................................................................... 18

3 Porting the Reference Framework to Linux ................................................. 21
  3.1.1 Platform Independence ...................................................................... 21
  3.1.2 Instrumentation .................................................................................... 23
  3.1.3 Building and Compiling the Reference Framework ......................... 25
  3.1.4 Porting Effort ....................................................................................... 25
  3.1.5 Test Driven Development ................................................................... 25

4 Initial Performance Analysis ....................................................................... 27
  4.1 Measurement Setup .................................................................................. 27
  4.2 Results and Analysis ............................................................................... 28
  4.2.1 All Scenarios ....................................................................................... 28
  4.2.2 Scenario of Interest ............................................................................. 29

5 Extended Thread State Analysis ................................................................. 35
  5.1 Expanding Thread States ....................................................................... 35
  5.2 Determining Thread States ..................................................................... 37
List of Figures

Figure 1: Waterfall Model illustrated [4] ......................................................... 3
Figure 2: The Goal Question Metric Approach visualized [3] .......................... 4
Figure 3: Three traditional respiration monitoring algorithms. (a) Spirometry, (b) thoracic impedance measurement [25] and (c) belt with strain gauge .................. 6
Figure 4: The Procor respiration algorithm block diagram [16] .......................... 8
Figure 5: Block diagram of AutoROI [19] ....................................................... 8
Figure 6: Dataflow diagram of Reference Framework (configured with AutoROI) .......................................................... 10
Figure 7: Loop concurrency in steps ............................................................... 11
Figure 8: Raspberry Pi 3 Single Board Computer ........................................... 11
Figure 9: Benchmarks comparing the Raspberry Pi processor (ARMv8) and workstation laptop processor (Intel i7) ................................................................. 12
Figure 10: Thread states typically investigated in the Thread State Analysis method [1] .................................................................................................................. 13
Figure 11: Illustration of functional parallelism where an application is divided in 3 steps allowing input to be pipelined. Coloured numbers indicate individual frames. .................................................. 15
Figure 12: Illustration of data parallelism where the input data is divided into 3 parts that are executed by three separate threads. Coloured numbers indicate individual frames. .................................................. 15
Figure 13: Linux defined thread states. [24] ..................................................... 16
Figure 14: The Red Black tree used to sort threads in the CFS (a, left) [26] and the relation of niceness, priority, to the runtime of a thread (b, right) .................... 17
Figure 15: Illustration of multi-level priority queues used in the Round Robin scheduler ........................................................................................................ 18
Figure 16: Events recorded by Perf [33] ......................................................... 20
Figure 17: Class diagram of Reference Framework. With slightly changed (gray), ported (orange) and added classes (green) .............................................. 21
Figure 18: VisicaAgent class diagram ................................................................ 22
Figure 19: ConcurUtils class diagram ............................................................ 23
Figure 20: UML Class diagram of Instrumentation namespace .......................... 24
Figure 21: Instrumentation output logged to CSV file ..................................... 25
Figure 22: Results of Unit Tests ....................................................................... 26
Figure 23: Measurement setup with boxes presenting the programs and packages, and arrows the flow of data. ........................................................................ 27
Figure 24: Visualization showing the signals determined by the respiration algorithm per block .......................................................................................... 28
Figure 25: Throughput in Frames per Second for different core counts .......... 28
Figure 26: Frame latency for different core counts ......................................... 29
Figure 27: CPU usage, per core and averaged for the first 150 seconds .......... 30
Figure 28: Grasp visualization showing execution of program threads (green), other threads (blue) and idle (orange) on cores for 150ms. Black indicates rapid context switching ............................................................................... 30
Figure 29: The previous figure zoomed in to show the areas of rapid context switching ........................................................................................................ 30
Figure 30: CPU usage per threads. Steps with data parallelism can use more than 100% CPU .......................................................... 31
Figure 31: Buffer occupancy in number of frames for steps (threads) in AutoROI .......................................................... 31
Figure 32: Average time in microseconds for steps (threads) in AutoROI ........ 32
Figure 33: DAG of AutoROI with average step durations on 4 cores .............. 32
Figure 34: Grasp timeline of thread frame execution ...................................... 32
Figure 35: Extending operating system threads states. States surrounded by blue boxes are new sub-states .......................................................... 36
Figure 36: Relating transitions to Perf profiling events ................................. 38
Figure 37: Perf process fork event (A) ............................................................. 38
Figure 76: Processing chain for ObjectPulse with algorithm steps in white

Figure 77: State occupation graph of ObjectPulse

Figure 78: Timeline of ObjectPulse

Figure 79: Processing chain for PulseLCB with algorithm steps in white

Figure 80: State occupation graph of PulseLCB

Figure 81: Timeline of PulseLCB

List of Tables

Table 1: Requirements for Respiration
Table 2: Raspberry Pi 3 hardware specifications
Table 3: Hardware and OS primitives benchmarking results
Table 4: Different Measurement Scenarios
Table 5: The available CPU usage for different core counts
Table 6: Timing for parallel and serial components of CalculateSignalToNoise
Table 7: Comparison of theoretical vs actual speedup in CalculateSignalToNoise
Table 8: Timing for parallel and serial components for steps using data parallelism
Table 9: Speedup for other data parallelized steps
Table 10: Performance of initial ported version
Table 11: Performance with CalculateBlockWeight parallelism removed
Table 12: Performance with ProcessAgent fixed
Table 13: Data parallel load balancing
Table 14: Completely Fair Scheduler results
Table 15: Round Robin Scheduler results
Table 16: Comparison of results for initial and final ported version
Table 17: Procor single region results
Table 18: Procor multiple regions results
Table 19: ObjectPulse results
Table 20: PulseLCB results
## Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVI</td>
<td>Audio Video Interleaved</td>
</tr>
<tr>
<td>C++11</td>
<td>C++ standard 2011</td>
</tr>
<tr>
<td>CFS</td>
<td>Completely Fair Scheduler</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>CSV</td>
<td>Comma Separated Values</td>
</tr>
<tr>
<td>DAG</td>
<td>Directed Acyclic Graph</td>
</tr>
<tr>
<td>EDF</td>
<td>Earliest Deadline First</td>
</tr>
<tr>
<td>ETSA</td>
<td>Extended Thread State Analysis</td>
</tr>
<tr>
<td>FCFS</td>
<td>First-Come-First-Serve</td>
</tr>
<tr>
<td>FIFO</td>
<td>First-In-First-Out</td>
</tr>
<tr>
<td>GPU</td>
<td>Graphical Processing Unit</td>
</tr>
<tr>
<td>IPC</td>
<td>Instructions Per Cycle</td>
</tr>
<tr>
<td>LLC</td>
<td>Last Level Cache</td>
</tr>
<tr>
<td>NUMA</td>
<td>Non Uniform Memory Access</td>
</tr>
<tr>
<td>OS</td>
<td>Operating System</td>
</tr>
<tr>
<td>OSAL</td>
<td>Operating System Abstraction Layer</td>
</tr>
<tr>
<td>PFSPD</td>
<td>Philips File Standard for Pictorial Data</td>
</tr>
<tr>
<td>PMC</td>
<td>Performance Monitoring Counter</td>
</tr>
<tr>
<td>POS</td>
<td>Plane-Orthogonal-to-Skin</td>
</tr>
<tr>
<td>PPG</td>
<td>Photo-Plethysmo-Gramphy</td>
</tr>
<tr>
<td>RAM</td>
<td>Random Access Memory</td>
</tr>
<tr>
<td>ROI</td>
<td>Region Of Interest</td>
</tr>
<tr>
<td>rPPG</td>
<td>remote PhotoPlethysmoGramphy</td>
</tr>
<tr>
<td>RR</td>
<td>Round Robin</td>
</tr>
<tr>
<td>SD card</td>
<td>Secure Digital card</td>
</tr>
<tr>
<td>SNR</td>
<td>Signal to Noise Ratio</td>
</tr>
<tr>
<td>TCP</td>
<td>Transmission Control Protocol</td>
</tr>
<tr>
<td>TSA</td>
<td>Thread State Analysis</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modelling Language</td>
</tr>
<tr>
<td>USE</td>
<td>Utilization Saturation Errors</td>
</tr>
</tbody>
</table>
1 Introduction

Philips Healthcare’s goal is to provide affordable care to more than 3 billion people by 2025 [5]. In order to achieve this goal, healthcare needs to be affordably provided during the entire health continuum, ranging from prevention, to diagnosis, treatment and home care. One of the projects that aim to make this vision a reality is the Contactless Vital Sign Monitoring project initiated by Philips Research and Philips Innovation Services. This project uses commercial off the shelf cameras to remotely monitor vital signs such as respiration and pulse. It enables vital sign monitoring where traditional contact-based vital sign monitoring is prohibitive, such as neonatal and burn care where electrodes and attachments risk patient infection or complications. Additionally, it enables exciting new innovations and use cases to improve healthcare.

To demonstrate the functionality and validate the contactless vital sign monitoring algorithms a Reference Framework has been developed. The purpose of this Reference Framework is to provide a platform for execution of the vital sign algorithms. The framework is implemented on Windows and executes on high-end laptops. It provides input, parallel processing agents and output visualization for the algorithms. The next step in commercializing the project is to prove that the algorithms can be executed on embedded platforms, such as a Raspberry Pi, with the desired throughput and latency. In order to move this framework to a constrained platform it is important to map the algorithms and to maximize the performance that can be achieved.

These days, even low cost embedded platforms consist of multiple cores, heterogeneity and intricate cache and Non Uniform Memory Access architectures. The advent of these embedded systems have made it possible to implement complex applications with intensive computational requirements on low cost platforms. These platforms often run full-fledged operating systems, which allow development to be accelerated with an abundance of packages and libraries. However, these devices are more resource constrained then their desktop counterparts and require that work is effectively scheduled and executed by the operating system in order to maximize performance. It is therefore vital that developers have insight on how their applications are executed. Unfortunately, the complexities and abstraction of the operating system often hides these nitty gritty details. Understanding how the operating system treats the threads of an application can offer great insight in where bottlenecks and issues may lie. There is therefore a need for a methodology to gain this insight and to resolve the bottlenecks and issues.

1.1 Problem Description

To be able to commercially apply the contactless vital sign technology it needs to be proven that the technology is feasible on low cost embedded platforms. The current Reference Framework is implemented on high-end workstations where the desired latency and throughput of the algorithms are easily met. Since the focus has primarily been on evaluating and proving feasibility of the algorithms no extensive performance analysis or profiling has been done. It is therefore not known how these algorithms will perform on constrained systems.

To be able to implement the Reference Framework on a Linux embedded platform several dependencies need to be resolved. The current implementation of the Reference Framework has several platform dependencies that bound it to Windows machines. These dependencies are related to the parallelism, buffering and synchronization; as
well as file I/O and date/time utilities. Improving the platform independence will allow the framework to be used on several architectures and operating systems which improves its versatility.

It is not only desired to execute the Reference Framework on an embedded platform, but it is also desired to do so efficiently. Currently it is not known how the dozens of threads of the Reference Framework are mapped and executed by the operating system. Since the embedded platform is resource constrained it is important to understand how the framework is executed. Understanding this execution can potentially identify issues and allow the efficiency and the performance to be improved.

1.2 Goals

The goals of this project therefore consist of two main parts:

i. Proof of Concept. In order to demonstrate that the algorithms on the Reference Framework can be executed feasibly on low cost embedded platforms a proof of concept needs to be developed. This proof of concept should execute the algorithms of the Reference Framework on a low cost Linux ARM platform such as a Raspberry Pi 3. As a use case the proof of concept will focus on AutoROI, a motion based respiration algorithm.

ii. Performance analysis and improvement. Once the application is ported, it is desired to analyse the performance and identify points of improvement. In an embedded platform where power usage and resource limitations are a concern it is required to execute the target application efficiently. The goal is to develop a structured method that can be used to analyse performance and improve it.

1.3 Research Questions

The following research questions are guiding in determining the direction of this thesis work. These research questions are derived from the goals and addressed in this thesis work.

i. Proof of Concept

Question 1: How is the Reference Framework implemented on Windows and how can it be modelled?

Understanding the paradigms, design and implementation choices of the Reference Framework is important to be able to create an efficient port on an embedded platform. This will require modelling the framework, as well as understanding the dependencies in data and how the processing and parallelism in the framework are constructed.

Question 2: What are the platform dependencies and limitations that currently prevent the platform from being executed on an embedded Linux platform?

In order to be able to execute the algorithms of the Reference Framework on the Linux embedded platform it is important to understand which Windows dependencies are contained in the framework. These dependencies should be replaced with platform independent alternatives, if possible, or with the Linux equivalent. It is also important to understand how the code is structured and built. If possible a solution should be implemented with a single code base that simplifies project maintenance.
ii. Performance analysis and improvement

**Question 3:** Is it possible to analyse the performance of a multithreaded application on an embedded platform and quantify or visualize its behaviour and efficiency?

Once the initial port is implemented on the platform, it needs to be determined how efficient the application is executed. This will require understanding how Linux schedules and executes multithreaded applications as well as which profiling tools and methods can be used to analyse the program execution. Several techniques for performance analysis should be investigated. Based on this a method should be developed and applied to analyse the performance and efficiency.

**Question 4:** How can this analysis be used to identify issues and improve performance?

Once the analysis is complete it is interesting to determine what issues can be identified. Therefore an investigation should be made into the observations that potentially cause inefficiency and underutilization. Based on these observations issues should be identified and addressed. The analysis should then be repeated to determine that the issues are indeed resolved.

1.4 Approach

Due to the nature of the two goals, two separate development approaches where used in the implementation work. The first goal, to provide a proof of concept of the Reference Framework on an embedded platform, is a typical software engineering challenge. The approach used is similar to the waterfall model life cycle. [4] First an analysis of the current implementation and its requirements are performed in order to answer the first and second research questions. Next, a design is created and implemented. Finally the implementation is verified through testing.

![Figure 1: Waterfall Model illustrated [4]](image)

The approach used also incorporated several aspects of Agile development such as Unit Testing, retrospectives and daily stand-up meetings.

The second goal represents the development of an analysis method and using the method to identify issues and their improvement. The Goal-Question-Metric Approach can be used as a structured approach to apply these improvements. [3] The approach is illustrated in [Figure 2], where first the Goal is defined. Questions are derived from the goal, and finally Metrics are given to quantifiably answer the questions. This approach can be used to answer the third and fourth research questions. The goals are to
improve the performance, with the metrics of throughput, CPU usage and latency. Answering the research questions will determine how to measure the Metrics and the observations made during the analysis will introduce the Questions.

![Diagram](image.png)

**Figure 2:** The Goal Question Metric Approach visualized [3]

Goals specify the purpose, issue, process and viewpoint from which the measurement is to be taken. The questions divide the issues into its parts. For each question objective metrics are defined to answer them.

1.5 Contributions

In this thesis we present several contributions that were made to the Contactless Vital Sign Monitoring project at Philips Innovation Services. These contributions are the result of answering the research questions and fulfilling the goals. The method and most of the tools developed are generic and can be applied to other applications on Linux.

The following contributions are specific to the Contactless Vital Sign Monitoring project:

- **Platform Independent implementation.** The platform independence of the Reference Framework is improved through a single code base for both Windows and Linux. This porting is done by improving the utilities providing the Operating System Abstraction Layer. Additionally unified project generation files can generate build files for both Windows and Linux. (Must Have)

- **Instrumentation.** In order to gain better understanding on how the algorithms and their steps perform, instrumentation is added. This instrumentation, implemented for both Windows and Linux can measure the execution times, and information of individual threads, as well as visualize the execution over time. An API is developed that allow algorithms to record instrumentation events. (Should Have)

- **Analysis of algorithms.** The analysis of the contactless vital sign monitoring algorithms using the investigative method presented in this thesis is provided in order to identify the steps of interest and the bottlenecks of the algorithms. (Could Have)

The following contributions are generic and concern the analysis method developed in this thesis work. This analysis can be applied to any Linux application

- **Thread State Analysis tools implementation on Linux.** The Thread State Analysis Method, or TSA method as presented by B. Gregg [1] is implemented for Linux and a set of tools are developed to apply the analysis.
• **Extended Thread State Analysis method.** The basic Thread State Analysis has several shortcomings that hide potential useful state information. In this work several additional states are added to allow a more thorough analysis. These states primarily investigate why a thread is involuntarily removed from a core and how useful the time which a thread is allowed to run on a core is.

• **Thread execution visualization.** The analysis is further extended to allow visualization of thread states over time using Grasp [2] and synchronizes these states with user level events. This allows better understanding of how the application is executed and where potential bottlenecks lie. The visualization is applied to identify issues and implement several improvements.

### 1.6 Outline

The thesis is organized as follows. Chapter 2 describes the background and application domain of the assignment. It models the current Reference Framework and its implementation and investigates the target hardware. A short survey of related work is also provided. Chapter 3 shortly describes the porting effort required to implement the Reference Framework on Linux and improve platform independence through an Operating System Abstraction Layer. The initial performance of the port for AutoROI, a motion based respiration algorithm, is evaluated in Chapter 4 where several performance issues and under-utilization is seen. Chapter 5 introduces the idea of the investigative method which focuses on thread states in order to identify and fix performance and utilization issues. This Extended Thread State Analysis method allows the performance of threads to be quantified and visualized. In Chapter 6 the method is applied to the AutoROI algorithm on the port and the results analysed. In Chapter 7 the issues identified by the analysis method are fixed and the method is reapplied to analyse the performance improvements. Chapter 8, provides a quick overview of the method applied to the other contactless vital sign monitoring algorithms on the Reference Framework. Finally, in Chapter 9 the conclusions are given and possible future work is suggested.

The following Appendices accompany this thesis:

• **Appendix A: Configuring Linux for Development:** Describes how a Linux machine can be configured to be used for development and execution of the Reference Framework.

• **Appendix B: Raspberry Pi 3 Hardware Benchmark Results:** This appendix includes the full benchmark results as provided by `lmbenchmark`. These results are used to estimate the cost of hardware and kernel operations in the analysis. The results of the benchmark are provided in their entirety.

• **Appendix C: Developed Performance Analysis Tools:** Describes the tools that were developed during the course of this thesis work and how to use them in order to perform the analysis.
2 Background and Related Work

In this chapter the background of the project is described and investigated in order to answer some of the research questions posed. First the application domain of Contactless Vital Sign Monitoring is introduced. Next the Reference Framework that currently implements the Contactless Vital Sign Monitoring algorithms on Windows is modelled and investigated. To better understand the potential performance the capabilities of the intended target hardware is briefly investigated. A quick survey of related work investigates techniques to analyse application performance. Finally useful background information and concepts are shortly described.

2.1 Application Domain

This section describes the application domain of Contactless Vital Sign Monitoring and the advantages over traditional vital sign monitoring with a focus on algorithms used for the monitoring of respiration. The section shortly describes AutoROI, the motion based respiration algorithm, which is the focus of the analysis and optimizations in this report.

2.1.1 Contactless Vital Sign Monitoring

Contactless vital sign algorithms is a promising new technology that opens up many new use cases for vital sign monitoring. As the name suggests contactless vital sign monitoring does not require any equipment to be in direct contact with the patient being monitored allowing vital sign monitoring to be done in a non-invasive manner, quickly and remotely. This section briefly describes traditional respiration monitoring, contactless monitoring and the advantages of contactless monitoring over traditional vital sign monitoring.

Traditional respiration monitoring techniques often require expensive equipment to be operated by trained professionals. According to M. Bartula et al, there are three main techniques in use today to measure the respiration rate of patients [16]. The first measures the nasal or oral airflow using a spirometer or mask to monitor respiration (a). The second method measures changes in thoracic impedance using electrodes attached to the chest. When a patient breathes the volume of air in the chest causes changes in the impedance of skin which can be used to determine the respiration rate (b). Finally the third method uses a belt with a built in strain gauge worn by the patient. Breathing causes the belt to stretch which can be measured to monitor respiration (c).

![Figure 3: Three traditional respiration monitoring algorithms. (a) Spirometry, (b) thoracic impedance measurement [25] and (c) belt with strain gauge.](image)

Contactless vital sign monitoring may use many different technologies such as radar [23], infrared cameras to measure heat produced by breath [21] or markers that are tracked [22]. The algorithms developed by Philips Research and implemented on the
Reference Framework by Philips Innovation Services use commercial off the shelf (COTS) cameras. These cameras are commonly available, affordable and often integrated in products where the algorithm might be needed, for instance in baby monitors [19]. The frames captured by these cameras are image processed by the algorithms to monitor vital signs.

Vital sign monitoring is an important factor in diagnosing and monitoring the health of patients [16]. Respiration monitoring is particularly useful in identifying severe respiratory disorders such as sleep apnea. Camera based contactless vital sign monitoring offers several advantages over traditional and other contactless vital sign monitoring techniques. One example of these advantages is the improved suitability of vital sign monitoring in home healthcare monitoring, since no special equipment or training is required. When compared to in-clinic monitoring, monitoring vital signs at home allows coverage of a greater number of patients by caregivers, making it possible to improve the quality of care given. In addition, at home monitoring of at risk patients may allow early diagnosis of problems in order to prevent extended hospital stays. The financial impact of this advantage could be considerable.

Most vital sign monitoring used today requires attaching of sensors or electrodes. For some patients this in itself presents a risk, for instance with neonatal or burn care where electrodes and sensors can result in infections or complications. The discomfort caused from the equipment used to measure respiration may also prevent natural measurements from being taken, for instance with sleep monitoring. Additionally, traditional respiration monitoring technology, as well other non-camera based contactless respiration monitoring techniques, require expensive technology that is not affordable or available to the everyday consumer. For example, pulse rate monitoring using electrodes can have a daily consumable cost of up to 10 Euro per patient [20].

The use of these commercial of the shelf cameras, rather than specialized equipment, introduces several challenges that require complex algorithms and years of continuing research for robust solutions to monitor vital signs. The next section introduces one of these algorithms.

2.1.2 Motion Based Respiration (AutoROI)

In this section a short description of the contactless respiration algorithm that is analysed in this thesis report is given.

The respiration algorithm investigated in this thesis is called AutoROI, or Automatic Region of Interest. The algorithm is based on a precursor algorithm known as Procor, which uses a technique known as Profile Correlation. The complete block diagram for Procor is given in Figure 4. The algorithm starts by creating and filtering a one dimensional vertical profile of a region of the frame of a video input. This profile is cross correlated to a previous profile in order to detect motion changes. Simultaneously a motion detector determines periods where large motion occurs and where respiration cannot be extracted. A classifier uses both the profile correlations and results from the motion detector to determine breaths. More detailed information can be found in [16].
AutoROI is an extension to this algorithm where further research was done by M. Rocque to add automatic region detection [19]. The extension, shown in Figure 5, works by dividing the image into blocks and applying Procor on each block. Candidate blocks that likely represent respiration are then selected for each frame. A persistence computation weights the importance of a block based on how often it is selected as a candidate block. Based on this the blocks that represent the region of interest are selected and the weighted sum of these blocks is used to compute the respiration signal. More detailed information can be found in [19].

Figure 4: The Procor respiration algorithm block diagram [16].

Figure 5: Block diagram of AutoROI [19].
The algorithm described above is implemented on the Reference Framework on Windows and will be the primary algorithm to be ported and analysed on the embedded platform. The stages of these algorithms are mapped by the Reference Framework onto several threads and processed in parallel to take advantage of multicore processing.

2.1.3 Respiration Requirements

In order for the results of the algorithm to be of any use as input for other medical devices or for accurate visualization it is required that the output be delivered regularly, at a certain throughput, and with limited delay from the capture of a frame, latency. Table 1 lists these requirements.

![Table 1: Requirements for Respiration](image)

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Throughput</td>
<td>5Hz minimum</td>
</tr>
<tr>
<td></td>
<td>- Frames processed a second.</td>
</tr>
<tr>
<td>Latency</td>
<td>200ms maximum</td>
</tr>
<tr>
<td></td>
<td>- Delay between a frame capture and results.</td>
</tr>
</tbody>
</table>

2.2 Reference Framework

In order to evaluate the contactless vital sign algorithms and provide a platform for executing these algorithms the Reference Framework has been developed. The Reference Framework contains the Contactless Vital Sign Monitoring algorithms and can be built as a stand-alone application to demonstrate the algorithms. This section shortly describes the Reference Framework and models the Reference Framework as a dataflow graph.

2.2.1 Purpose and Functionality

The Reference Framework embeds the contactless vital sign algorithms and provides functionality to allow evaluation and commercialization of the technology. Once an algorithm has been developed using MatLab it is implemented on the Reference Framework using C++. The Framework provides input capturing, processing utilities and output to the algorithms to simplify their implementation. The functionality of these libraries are shortly described below.

Input

The input library supports various input and reference sources that are formatted and buffered on a frame by frame basis for the processing steps. Input can be captured from the following sources:

- Support for several video file formats to retrieve frames of a video with meta data such as frame numbers and timestamps.
- Support for several cameras such as webcams and IDS uEye industrial cameras.
- Support for several traditional vital sign monitoring devices, such as PPG (Photo-Plethysmo-Gramphy) sensors, used as reference devices to compare the results of the algorithms.
Processing

The processing library provides an array of building blocks, such as threading “agents” and buffering, that allow the implementation steps of an algorithm as C++ classes. Instances of these classes can easily be connected and decoupled using the buffers. The classes provided in the processing library simplifies the implementation of the algorithms from MatLab to C++. The utility library is also linked in the processing library and provides utilities that can be used by the steps of the algorithms.

Output

The output stage allows the capturing, broadcasting and visualization of the output of the algorithms. The output from the individual steps can also be visualized and captured to make analysis of the individual steps possible. The output stage provides the following functionality:

- The output and traces can be visualized on the Graphical User Interface.
- The results of the algorithm and its individual stages can be written to CSV files to allow post run analysis of the accuracy and performance of the algorithms and their steps.
- The results can be transmitted using TCP. This allows other medical devices to potentially use the output of the Reference Framework as input.

2.2.2 Modelling the Reference Framework

This section models the framework as a dataflow graph with AutoROI in order to better understand how the Reference Framework executes the algorithms.

The dataflow graph in Figure 6 models the execution of AutoROI on the Reference Framework, actors shaded in grey are infrastructure steps. The steps of the processing can be modelled as actors that trigger as soon as frames, modelled as tokens in buffers, become available. The dataflow diagram also models the parallelism in the system as each step is implemented as a thread that can execute if a frame becomes available and it is assigned a CPU resource. This diagram can be used to analyse the concurrency and bottlenecks of the framework. The main application thread, not modelled, is responsible for creating the child threads and buffers connecting the threads; once this is done the application thread is idle until termination.

The execution consists of three stages as described in the previous section. First the input is captured. This is modelled as a videoFrame token that can be sourced from a
pre-recorded video or a camera. The InputAgent formats the videoFrame and tokens from a reference device such as a PPG sensor, Nexus sensor device or respiration trigger as a videoImage token that is provided to the processing steps. The processing chain consists of parallel threads that execute the steps of an algorithm and calculate the vital signs. The last step converts the results into a processResults token. The final stage then visualizes this token and/or writes the results to the file or the network.

Further refinement of the diagram reveals the concurrency during several steps where parallel loop execution takes place. These loops take advantage from data independence of individual blocks of a video frame to perform data parallelization. This can be modelled as tokens consumed by several actors, where the number of actors is maximally equal to the number of logical processors. This refinement is given in Figure 7 with the three steps that use parallel loops in AutoROI.

![Diagram of parallel loops in AutoROI](image)

**Figure 7**: Loop concurrency in steps

The Windows only implementation of the Reference Framework is typically capable of executing all the current algorithms ported to the framework in real time, the processing is completed for each frame before the next one arrives. This is due to the use of powerful workstation laptops. However, the intended target hardware will be embedded, low cost and constrained and will likely not be able to execute the algorithms in real time. The next sub-section briefly investigates the target hardware.

### 2.3 Target Hardware

In order to be able to make the application of the algorithms feasible it is necessary to prove that execution is possible on an affordable and constrained platform. This section introduces the target hardware for the embedded implementation of the Reference Framework.

The chosen platform is a Raspberry Pi 3, shown in Figure 8, an affordable single board computer (SBC) that can be purchased for around 40 Euro. The platform is a popular embedded solution due to its small form-factor and low power consumption. Plenty of support, libraries and information is available.

![Raspberry Pi 3](image)

**Figure 8**: Raspberry Pi 3 Single Board Computer
The specifications for the Raspberry Pi is given in Table 2. The Raspberry Pi uses a Broadcom processor with 4 cores based on the ARM Cortex architecture. The cores are clocked at 1.2 GHz and each core has its own L1 data and instruction cache. The L2 cache is shared among all cores. The Raspberry Pi processor includes a GPU, however the lack of support for frameworks such as OpenCL makes it difficult to use.

Table 2: Raspberry Pi 3 hardware specifications

<table>
<thead>
<tr>
<th>Specification</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CPU - Broadcom BCM2837</strong></td>
<td>4 Physical Cores</td>
</tr>
<tr>
<td>(ARM Cortex-A53)</td>
<td>4 Logical Cores</td>
</tr>
<tr>
<td></td>
<td>1.2 GHz</td>
</tr>
<tr>
<td></td>
<td>16 kB Instruction + 16 kB Data L1-cache</td>
</tr>
<tr>
<td></td>
<td>512 kB L2-cache (shared)</td>
</tr>
<tr>
<td><strong>GPU – Broadcom VideoCore IV</strong></td>
<td>12 Cores</td>
</tr>
<tr>
<td></td>
<td>400 MHz</td>
</tr>
<tr>
<td></td>
<td>64 MB LPDDR2 (shared RAM)</td>
</tr>
<tr>
<td><strong>RAM</strong></td>
<td>1GB LPDDR2</td>
</tr>
<tr>
<td><strong>Cost</strong></td>
<td>~ € 40</td>
</tr>
</tbody>
</table>

In order to get an impression on the potential performance difference several benchmarks were ran to compare the Raspberry Pi’s Broadcom ARMv8 processor to the Intel i7 processor currently used in the workstation laptops the Reference Framework executes on. This included the Whetstone and LinPack benchmarks for floating point operations and Dhrystone for integer operations. As well as the Livermore Loops benchmark which is specifically designed to benchmark parallelism.

![Benchmark Performance](image)

**Figure 9:** Benchmarks comparing the Raspberry Pi processor (ARMv8) and workstation laptop processor (Intel i7).

The results are given in Figure 9 showing that the processor on the Raspberry Pi is between 5 to 11 times slower depending on the benchmark used. It is likely that a similar performance difference will be seen with the Reference Framework on the Raspberry Pi.
2.4 Related Work

Ensuring that applications execute efficiently is vital in order to run high demand applications on low cost constrained systems. Therefore much research has been done and many analysis techniques developed to analyse the performance of a system. This section briefly investigates related work in the domain of analysis methodologies.

The need for performance analysis has motivated the development of extensive support for both hardware and software performance counters. The Linux system provides a powerful set of these counters and events, as well as tools to perform analysis based on these counters [28]. The benefit is that such counters are largely supported across a variety of systems. The investigation therefore considers techniques that use these counters and the tools derived from them to apply the analysis, rather than techniques that are specific to a particular CPU architecture. This allows the developed method to be generic and reapplied on a variety of systems.

When deciding on an analysis method it is beneficial to consider the viewpoint of the analysis. Different techniques have different viewpoints from which the analysis is approached. Methods such as the Utilization-Saturation-Error, or USE, method focus on analysis of system resources. The USE method investigates each resource’s usage, saturation and errors. The analysis requires systematically investigating different resources such as CPU, memory and disk to identify underutilization, bottlenecks and potential errors. Because of its focus on resources the analysis needs to be redeveloped for each specific architecture [29], reducing the generic nature of the method. Additionally since the analysis focuses on the system rather than the application limited information can be deduced.

Other methods focus on the analysis of the application itself. One particular common method is Hotspot analysis [31]. Applications are built with debugging symbols and profiled by sampling. A hierarchical visualization is created that can be used to identify parts of code that spend a significant amount of time executing. The analysis primarily focuses on CPU usage.

Another method that focuses on the analysis of the application is the Thread State Analysis method or TSA method. The method focuses on identifying states and determining the amount of time threads spend in states in order to analyse performance issues [1]. The method is typically applied to servers that are tasked with high I/O, computation bursts and network loads and used to pinpoint application threads that cause issues. The method is implicitly used in several tools such as Perf TimeChart where the thread states are visualized [30]. However, the visualization is limited since a single process thread, rather than individual application threads, is visualized and only limited states are visualized.

![Figure 10: Thread states typically investigated in the Thread State Analysis method [1].](image)
The methodologies investigated in this section each have their own benefits and limitations. The USE method focuses specifically on resources and is useful when focusing on architecture rather than application performance. Hotspot analysis can be used to pinpoint exactly where processor cycles are being consumed, however the focus is on excessive CPU usage and limited information can be derived for other issues such as I/O usage and work distribution. The analysis also requires detailed understanding of the application code.

The TSA analysis is an generic alternative that can be applied to an application without any detailed understanding of application code. The use of threads allow problems to be pinpointed on the granularity of threads. However, the current method has several shortcomings that make it less applicable to the analysis for the application domain and embedded platform in this thesis. The analysis is typically applied to more powerful machines with larger memories, cache hierarchies and more powerful CPUs. The analysis presented in [1] therefore fails to answer two questions of interest: why has a thread been moved to the waiting queue for access to the core and how useful is the time a thread spends on the core. The work presented in this thesis aims to answer these questions as a contribution to the method. Additionally the current analysis can benefit from timeline visualization to show interactions between threads, since synchronization is often required for threads due to their parallel nature.

2.5 Background Knowledge

This section introduces several concepts and background knowledge that is useful for the work presented in this thesis.

2.5.1 Parallelism Concepts

Nowadays even embedded platforms often have several powerful cores. In order to be able to exploit these cores applications need to be implemented with parallelism in mind. This section shortly describes parallelism concepts that are used in the Reference Framework to benefit from the multiple cores available on the target hardware.

In an operating system the granularity of parallelism is on thread level. Therefore for an application to exploit parallelism it has to consist of several threads (threads are the unit of concurrency). Two main types of parallelism are exploited on the Reference Framework.

Functional parallelism

In functional parallelism an application is divided into several sequential steps that are implemented on different threads. These threads each work on a different stage of the application and can be executed by different processors at the same time. In the Reference Framework functional parallelism is used to pipeline the processing of the input video frames by the algorithms. An example is given in Figure 11, where an application is functionally decomposed into 3 steps with a thread for each step. This allows the input to be pipelined and all threads to work simultaneously on different frames of data using three processor cores.
Data parallelism can be exploited when the same calculations are to be performed independently on different parts of input data. When data parallelism is exploited several sub-threads are created that during the parallel region execute the same code, but on different parts of data. Several frameworks exist that can be used to manage the creation and work distribution of data parallel threads, an example of this framework is OpenMP [17]. The Reference Framework exploits data parallelism by using several threads to process for loops where data independence is guaranteed, this is sometimes referred to as loop level parallelism. An example is given in Figure 12, where input data is divided into three separate parts and the entire input data is executed by three threads simultaneously using three processor cores.

Hybrid parallelism

Hybrid parallelism is the combination of both functional and data parallelism in an application. The Reference Framework is an example of an application that exploits both functional and data parallelism. The blocks of a frame are processed by various stages of the algorithms through loop concurrency, and frames through functional parallelism.

2.5.2 Linux Thread Scheduling

This section briefly introduces the principles behind operating system thread scheduling and thread management for Linux. The Reference Framework consists of dozens of threads and understanding the principles behind the management and scheduling of these threads is vital in properly optimizing the performance.

Threads

The smallest granularity of work that can be executed by the Linux operating system is a thread. Every single unit of work is done using threads. In Linux threads are implemented as light weight processes, therefore the term process and thread is
analogous for the purpose of this thesis. The operating system scheduler should ensure that threads are divided across the cores fairly and assigned to cores when work needs to be done.

During program execution threads may block waiting on resources, mutexes or sleep and wait when no work needs to be done. The scheduler might also decide to remove a thread from a processor if other threads are waiting to run. In order to manage this efficiently threads are given states which are maintained in the Process Control Block (PCB) of the thread. This PCB is a data structure maintained by the operating system that has all the information necessary to allow a thread to be removed from a core and later restored. These states determine whether a thread is waiting to run, currently executing, finished executing or waiting for an event or signal. In Figure 13 the states of a thread as defined in Linux are given. The Linux kernel maintains a total of five states, the Ready and Executing state are combined as the Running state in the operating system [24]. A thread normally enters the Running state where it waits to execute, is assigned a processor to execute and may be interrupted where it returns in a ready queue waiting for processor assignment to continue execution. If a thread needs to wait for a signal or event it will alert the scheduler and be suspended. The thread can then either enter the Interruptible state, where it can wake on both events or signals, or the Uninterruptible state where it can only be woken on events. The Interruptible state indicates sleeping or waiting on a resource. The Uninterruptible state generally indicates that the process is waiting for events directly from the hardware. A thread may also enter the Stopped state by a debugging signal. After finishing its work a thread will be terminated and enter the Zombie state.

The threads of the Reference Framework will enter the Interruptible state when waiting for frames, once a frame arrive the thread will enter the Ready state to wait for access to a core to perform work. Once a process is assigned the thread will enter the Executing state to start processing the thread.

Figure 13: Linux defined thread states. [24]
Scheduling of Threads

Since all work is done via threads, the scheduling of threads needs to be flexible and capable of scheduling all types of threads from high intensive threads, to real time threads and background tasks. To manage the scheduling of threads several scheduling algorithms are available each with their own advantages and disadvantages and characteristics suitable for different purposes. These scheduling algorithms are modular and categorized into either a Real Time class or Other class scheduling algorithms. Threads with the Real Time scheduling algorithms are allowed to execute before threads with the Other class scheduling algorithms. The Round Robin and FIFO scheduler algorithms are categorized as Real Time and the Completely Fair Scheduler algorithm as Other. These algorithms are shortly described.

The **Completely Fair Scheduler** is the default scheduler used in Linux. The goal of the scheduler is to divide the processors fairly across all threads waiting to run and at the same time ensuring responsiveness of threads. The scheduler dynamically computes a period based on the number of tasks waiting to run and divides the period across the running tasks, according to equation 1.

\[
CFS \ Period = \text{MAX}(\text{schedLatencyNs}, \#\text{ReadyTasks} \times \text{schedMinGranularityNs}) \tag{1}
\]

The schedLatencyNs is a configurable parameter defining the minimum schedule period. To ensure that the time slice does not grow too small, the schedMinGranularityNs defines the minimum time each thread should be allowed to execute. The default values depend on the platform, but are configured at 18ms and 2.25ms for the schedLatencyNs and schedMinGranularityNs respectively on the target hardware.

![Diagram](image)

**Figure 14:** The Red Black tree used to sort threads in the CFS (a, left) [26] and the relation of niceness, priority, to the runtime of a thread (b, right).

To implement priority and to ensure that threads get a bigger slice of this period the niceness value of the thread can be changed. A lower niceness value can be used to exponentially, **Figure 14** (b), increase the weight of a thread in calculating the time slice as seen in equation 2. The time slice a thread will be given in the period is calculated from equation 3.

\[
\text{threadWeight} = \frac{1024}{1.25^{\text{niceness}}} \tag{2}
\]

\[
\text{Thread Timeslice} = CFS \ Period \times \frac{\text{threadWeight}}{\text{totalThreadWeight}} \tag{3}
\]
To pick threads fairly the scheduler maintains a total sum of the runtime of each thread in a red-black binary tree. Figure 14 (a), that is ordered by this runtime. The scheduler always picks the left most node to execute [27]. This ensures that tasks that rarely run, with lower total runtimes, will be the first task picked to run. Similarly the virtual runtimes for higher priority tasks increments slower when running making them more likely to be the left most node to be picked to run. This also allows a thread to only use part of its time slice and when it becomes ready to run again it will be picked immediately since its run time was only partially incremented.

The **Round Robin scheduler** is a real-time scheduler that provides threads waiting to run pre-determined time slices or quantum during which the thread will be given access to a core. A thread can execute until it exhausts its quantum after which the next thread waiting to run will be assigned a quantum. The scheduler allows the priority of threads to be set. A higher priority thread will immediately pre-empt a running lower priority thread if it becomes ready to run. This is illustrated in **Figure 15** where high priority thread A is allowed to run on the CPU until it yields. After this middle priority threads B, C and D will execute in a round-robin fashion. After a thread exhausts its time slice it is placed in the back of the queue. Only after threads A, B, C and D all yield will thread E be able to run. If thread E is running and any of the other threads become available it will be immediately pre-empted, provided that no other core is available. This ensures that higher priority threads are responsive, however the disadvantage is that lower priority threads can be delayed from executing for a non-deterministic period of time.

The **First-In-First-Out algorithm**, sometimes referred to First-Come-First-Serve (FCFS), is similar to the Round Robin algorithm, but threads do not have a quantum or time-slice. Threads are allowed to execute until they voluntarily yield. The disadvantage is that this can potentially result in less responsive threads if there are threads waiting in the run queue with the same priority and all cores are busy [24]. The priority of the threads determine which run queue a thread will be placed in, as the Round Robin scheduler. Threads entering the Run state in higher priority queues will immediately pre-empt running lower priority tasks, as illustrated **Figure 15**.

### 2.5.3 Profiling Tools

In order to understand how an application is executed by the operating system it is necessary to profile the application and benchmark the target hardware. This section briefly investigates the profiling tools and benchmarks available in Linux to be able to profile the application.

Overheads such as context switching, page faults and cache misses can greatly influence the performance of an application. In order to understand the impact these overheads may have on an application it must first be determined how expensive these operations
are. The *lmbenchmark* benchmark measures the performance of common hardware operations and operating system primitives for any Linux machine. This includes cache and main memory access times, file I/O bandwidth and context switching overhead. The benchmark has been compiled and run on the Raspberry Pi 3. The relevant results that were obtained are listed in Table 3. The author of the benchmark requests that results are published in their entirety to prevent cherry picking, the complete results are included in Appendix B: Raspberry Pi 3 Hardware Benchmark Results.

To measure I/O read and write performance the utility *dd* can be used to write and read files with measurement results enabled. Before reading the file the cache is flushed using */proc/sys/vm/drop_caches*. On a Raspberry Pi the results are highly dependent on the SD card that is used as the storage medium. The results are listed in Table 3. A comparison of different SD cards as well as instructions on these measurements can be found on the Raspberry Pi Wiki [7].

To be able to analyse how the tasks are scheduled it is necessary to determine how the scheduler is configured. The time quantum applied for the different scheduling policies can be read from the kernel configuration files in */proc/sys/kernel*. The quantum assigned for real time Round Robin scheduling can be read from *sched_rr_timeslice_ms*. The quantum for the Completely Fair Scheduler is dependent on the system load, but the minimum and maximum bound can be read from *sched_min_granularity_ns* and *sched_latency_ns* respectively. For the FIFO schedule policy a quantum is not applicable. Parameters are listed in Table 3.

Table 3: Hardware and OS primitives benchmarking results

<table>
<thead>
<tr>
<th>Measurement</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory Access Times (<em>lmbenchmark</em>)</td>
<td></td>
</tr>
<tr>
<td>L1 cache access time</td>
<td>2.53 ns</td>
</tr>
<tr>
<td>L2 cache access time</td>
<td>6.54 ns</td>
</tr>
<tr>
<td>Main Memory (RAM)</td>
<td>29.4 ns</td>
</tr>
<tr>
<td>Page Fault</td>
<td>1431 ns</td>
</tr>
<tr>
<td>Context Switching (<em>lmbenchmark</em>)</td>
<td></td>
</tr>
<tr>
<td>2 processes / 0 Kbyte cache data</td>
<td>4690 ns</td>
</tr>
<tr>
<td>I/O Bandwidth (<em>dd</em>)</td>
<td></td>
</tr>
<tr>
<td>File Write</td>
<td>6.3 MB/s</td>
</tr>
<tr>
<td>File Read</td>
<td>20.4 MB/s</td>
</tr>
<tr>
<td>OS Scheduling Quantum (<em>proc</em>)</td>
<td></td>
</tr>
<tr>
<td>RR Policy Quantum</td>
<td>10ms</td>
</tr>
<tr>
<td>CFS Policy Quantum Min</td>
<td>2.25ms</td>
</tr>
<tr>
<td>CFS Policy Quantum Max</td>
<td>18ms</td>
</tr>
<tr>
<td>FIFO Policy Quantum</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Once these metrics are known it is necessary to profile the application to capture these events. The comprehensive Perf utility can be used to record various hardware and software events. This includes scheduling, context switching, page faults, cache and I/O events. An overview of the events that Perf can capture is given in Figure 16.
The utility captures an enormous amount of data. This data, while comprehensive, can be hard to visualize and analyse. A minute of profiling can easily generate a data file of several hundred megabytes. The benefit of using Perf is that it can be applied on any program without requiring code instrumentation or recompilation. However, this complicates the matching of the application code to the runtime threads unless threads are properly identified. It is therefore common to add instrumentation to the code to mark events and allow identification of threads. This instrumentation can then be used for visualization by tools such as Grasp [2].
3 Porting the Reference Framework to Linux

The Reference Framework investigated in chapter 2.2 is the current solution specifically developed to demonstrate and evaluate the Contactless Vital Sign Monitoring algorithms. The Reference Framework is a stand-alone application which includes the algorithms. But before being able to execute the Reference Framework on the embedded Linux platform the current operating system and platform dependencies need to be addressed. The goal in this porting effort is to maintain a single code base that can be used to build and run the framework and its algorithms for both Windows and Linux. Additionally choices should be made to use as much platform independent code and standard functions available in C++ to improve maintainability. While this report focuses on AutoROI, the porting efforts included porting of all the algorithms currently on the Reference Framework. This section describes some of the porting effort required to execute the framework on Linux.

3.1.1 Platform Independence

In this sub-section the Reference Framework architecture is modelled as a class diagram with some of the dependencies modelled and the porting effort is shortly described. The Reference Framework is cleanly divided into interface and implementation. The Reference Framework uses factories that can be expanded to support additional input sources, visualizations and algorithms [14].

![Figure 17: Class diagram of Reference Framework. With slightly changed (gray), ported (orange) and added classes (green).](image-url)
In Figure 17 the Reference Framework architecture is modelled as a class diagram. In order to keep the diagram readable only some of the dependencies are indicated. The classes that spawn threads are stereotyped as well as the buffers that decouple the frames, indicated by blue dashed lines. The stereotyping provides a better understanding on the behaviour of the Framework. The threads of the steps of an algorithm that process a frame are created by the ProcessAgentFactory.

The Reference Framework builds several libraries and executables, the classes that belong to these libraries and executables are indicated within containers in the diagram. The class structure is also reflected in the directory organization and structure. The framework provides two executables: Core that runs the framework with algorithms, and UnitTests that tests the system and individual classes. The Operating System Abstraction Layer, or OSAL, is maintained in the Utility library that provides functions for threading, file handling and date/time utilities. The classes marked in grey required minimal porting and changes such as removal of operating system specific header files or adapting to changes in function signatures required by changes in the OSAL. Classes in orange required re-structuring or significant code changes. Classes in green were added during the porting in order to improve the platform independence or add new functionality. The porting efforts for the classes that required significant changes are described shortly.

VisicaAgent

The VisicaAgent class is responsible for providing an abstraction of threading and buffering to the steps of the algorithm. The steps of an algorithm extend a VisicaAgent instance and are required to implement an ExecuteLoop function where the step performs its work. Each algorithm has several steps that derive from the VisicaAgent class. A class derived from the ProcessAgentBase class maintains a list of these agents and ensures that they are configured and that their threads are started and stopped correctly. The VisicaAgent class is useful because it provides a single derived class that encapsulates the threading concepts in the Reference Framework.

The Windows specific threads were replaced with standard C++11, the C++ 2011 standard, threads to improve the platform independence. During the porting this class was extended to improve the flexibility and management of threading in the Reference Framework. Functionality was added to identify threads as well as to instantiate a thread with a given priority and scheduling policy. The class diagram is given in Figure 18.

Figure 18: VisicaAgent class diagram
ConcurUtils

The ConcurUtils namespace provides various constructs that are used to handle synchronization and atomicity between threads. The classes in this namespace use features introduced in the C++11 standard library to ensure platform independence. The class diagram is given in Figure 19. It provides a OS-agnostic C++ implementation of buffering between threads to replace the Windows specific concurrency buffers. The class also implements an event class that threads can use to signal events such as key presses.

Figure 19: ConcurUtils class diagram

The ConcurUtils namespace was added as a dependency for the VisicaAgent. It implements the buffering and synchronization of threads provided by the VisicaAgent class.

RaspiCam

The RaspiCam class is a VideoSource that can be used as an input source for the Reference Framework. It uses the RaspiCam C++ API library [15] and provides an adapter that implements the IVideoSource interface. The class allows the camera on the Raspberry Pi to be used as video input for the algorithms. A guide on how to enable the camera and build the libraries is available in Appendix A: Configuring Linux for Development.

Instrumentation

The Instrumentation class adds profiling that allows the timing of steps as well as the throughput and latency to be recorded, the class is described in detail in 3.1.2 Instrumentation. The instrumentation is supported for both Windows and Linux.

3.1.2 Instrumentation

In order to better understand how the Reference Framework, and operating system, execute an algorithm instrumentation code is added. This instrumentation is capable of measuring the execution and synchronization times, as well as buffer occupancy, of each individual frame processed by the steps. The instrumentation profiles the threads that are part of the algorithm steps, as this is the focus of the evaluation. Additionally information and statistics of all the program threads are captured. This section briefly describes the implementation of the instrumentation class.
Figure 20 illustrates the implementation of the instrumentation code as a class diagram. The instrumentation code is integrated into the Reference Framework and an instance of the Instrumentation::Trace class is created for each step in the algorithm. This is done automatically by the VisicaAgent thread abstraction layer that creates the threads for each step. This dependency is illustrated in Figure 17 where the structure of the Reference Framework is described. In order to add additional instrumentation traces, the algorithms being executed simply need to call functions to trace the points of synchronization and execution.

To maintain accurate timing the std::chrono class introduced in C++11 is used to capture the time points. In addition several other metrics such as thread IDs, priority, CPU usage and memory usage are recorded.

Upon termination of the application detailed thread run information and statistics are transparently read and logged from the kernel for each thread. Following this the captured trace points added in the code are formatted and written to a file (Figure 21). This reduces the overhead introduced by the instrumentation during program execution since no formatting or file I/O is done during runtime. However, this does come at the cost of an increase in RAM usage. The instrumentation therefore captures both information from the kernel and from the user space allowing detailed post analysis and visualization with Grasp.
The output from the user code instrumentation can be interpreted by several Python scripts. These scripts can calculate and graph the framerate, latency, buffer occupancy and individual step times. The scripts can also parse the output and generate input for frame visualization using Grasp. See Appendix C: Developed Performance Analysis Tools for more information.

3.1.3 Building and Compiling the Reference Framework

To improve project maintainability, project files and make files are generated using CMake [34]. This allows project files to be generated and compiled using the same workflow for both Linux and Windows. The result is that a single code based can be used. CMake will automatically configure the project files and resolve dependencies based on the platform it is run on. The switch from Linux to Windows is therefore nearly effortless.

3.1.4 Porting Effort

The code of the Reference Framework, excluding external libraries is contained in 816 C++ files with a total of more than sixty thousand lines of code. The changes during the course of the porting effort took place over 44 commits with more than 4700 lines of code, about 8% of the total code base, either inserted or deleted.

3.1.5 Test Driven Development

The unit tests consists around 500 tests that test the individual classes and their exposed functions. The existing tests cover the threading agents, buffering and concurrency utilities that were added. Therefore no additional tests were added. Additionally the system tests ensure that algorithms still produce the correct output when exercised with the given input videos. The tests of the modified Reference Framework were re-run on Windows with all the tests passing.

On Linux all the unit tests pass except for several system tests that test the algorithms themselves, these tests fail due to missing dependencies, in particular the missing proprietary face tracker library for which no source code is available or changes in the latest version of OpenCV. A screenshot of the 6 failing tests and the 501 tests that pass is given in Figure 22.
The Reference Framework supports several different algorithms for respiration and pulse monitoring. The results of the system tests for the individual algorithms are given below.

1. AutoROI: Passes all system tests and unit tests.
2. Procor: Passes all system tests and unit tests.
3. ObjectPulse: System test fails due to changes in the OpenCV 3.2 resulting in slightly different output, this issue also occurs on the mainline development branch.
4. PulseLCB and PulsePos: The PulseLCB and PulsePos algorithms use a face tracking library to identify patches of skin on which to apply the algorithm. The library is proprietary and source code has not been made available. The algorithm therefore uses a fixed region and produces different output then expected causing the system test to fail.
4 Initial Performance Analysis

After porting is complete and the validity proven via the unit tests the performance of the algorithm on the target hardware can be reliably evaluated. This analysis focuses on AutoROI the motion based respiration algorithm described in previous sections. First the scenarios to be measured are described, followed by the results. These results are analysed and conclusions on the performance are drawn.

4.1 Measurement Setup

This section describes the measurement setup and scenarios that have been defined in order to evaluate the performance of the AutoROI algorithm on the target hardware.

![Figure 23: Measurement setup with boxes presenting the programs and packages, and arrows the flow of data.](image)

The measurements are taken on a Raspberry Pi 3 using pre-recorded videos. The setup of the tests are illustrated in Figure 23. The AutoROI algorithm is run on the Reference Framework and the instrumentation added profiles the execution times of the steps. Additionally profiling tools are run to log the core and threads CPU usage, as well as monitoring tools to monitor the clock frequency and core temperature of the ARM processor. This monitoring ensures that the results are relatively consistent and that frequency scaling does not influence results. Above 80 °C the clock frequency is reduced to prevent thermal damage, heatsinks provided cooling allowing the clock frequency to be kept consistent at the maximum of 1.2 GHz. Additionally the results are averaged over the first 500 video frames. To reduce the system load unnecessary services such as the Graphical User Interface are disabled.

Since AutoROI applies the respiration algorithm on each block in the video stream, by increasing the resolution of the input video and decreasing the block size the computational load can be increased substantially. Three input scenarios are identified; see Table 4.

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Video Input</th>
<th>Block Sizes</th>
</tr>
</thead>
<tbody>
<tr>
<td>#1 – Low</td>
<td>320x240 px @ 31 FPS</td>
<td>80x80 px – 12 blocks</td>
</tr>
<tr>
<td>#2 – Medium</td>
<td>768x576 px @ 60 FPS</td>
<td>24x24 px – 768 blocks</td>
</tr>
<tr>
<td>#3 – High</td>
<td>768x576 px @ 60 FPS</td>
<td>16x16 px – 1728 blocks</td>
</tr>
</tbody>
</table>
The Medium scenario is chosen for the detailed measurement and analysis, representing a fairly intensive work load. The original Windows workstation used for development is capable of executing the algorithm and processing the 60 FPS video in real-time for the given scenario (#2 - Medium).

![Visualization showing the signals determined by the respiration algorithm per block.](image)

Additionally the profiling was repeated with a variation of the number of cores made available to the Reference Framework. Linux allows the threads of a process to be pinned to cores using the `taskset` command. The Raspberry Pi 3 has homogeneous cores with identical private data and instruction cache that together share an L2 cache. Therefore the combination of cores used is irrelevant and only the amount is varied.

### 4.2 Results and Analysis

In this section the results of the profiling and analysis are given for the measurement scenarios described in the previous section. First, basic analysis is given of all the scenarios, followed by a more in depth analysis of the Medium load scenario.

#### 4.2.1 All Scenarios

In the next figures the measured throughput and latency, as well as the available CPU usage, for the different scenarios are given. The algorithm is running using the default Linux scheduling algorithm (Completely Fair Scheduler) and priority. The true available CPU usage depends on the core affinity and the cores the process is allowed to execute on. The total CPU usage from the available CPUs is therefore a fraction of the total CPU resources if the number of cores are reduced, calculated as follows:

\[
\text{CPU usage from Available} = \text{CPU Usage} \% \times \frac{4}{\text{#used CPUs}^+},
\]

\[(4)\]

![Throughput in Frames per Second for different core counts.](image)
The CPU usage is summarized in the table below:

<table>
<thead>
<tr>
<th>Scenario</th>
<th>1 Core</th>
<th>2 Cores</th>
<th>3 Cores</th>
<th>4 Cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>#1 – Low</td>
<td>39.7 %</td>
<td>32.1 %</td>
<td>22.6 %</td>
<td>18.1 %</td>
</tr>
<tr>
<td>#2 – Medium</td>
<td>99.8 %</td>
<td>94.6 %</td>
<td>76.4 %</td>
<td>72.2 %</td>
</tr>
<tr>
<td>#3 – High</td>
<td>99.3 %</td>
<td>88.0 %</td>
<td>77.2 %</td>
<td>71.6 %</td>
</tr>
</tbody>
</table>

As expected, increasing the number of cores improves the throughput. The latency of the frames are also decreased, likely due to the possibility to better exploit data parallelism. As can be seen in Figure 25, only in the case of the Low load scenario is the video processed in real time. This is possible even if only 1 core is made available. The load is significantly increased in the Medium and High load scenarios. In all core configurations for these loads the application is unable to process frames in real time. In both the Medium and High load scenarios the CPU is not fully utilized, with all 4 cores more than a quarter of the CPU power is unused. If the cores are reduced to 3 the available CPU usage only slightly increases. Only when the cores are reduced to 2 or less is high CPU usage in the range of 90 to 100 percent seen.

For the scenario of interest, Medium load with 4 cores, the minimum throughput and maximum latency requirements are met. The target requirements are at least 5 frames per second with a latency of 200ms. The achieved performance is 9.6 frames per second with a latency of 174.3ms.

4.2.2 Scenario of Interest

The investigation now focuses on the scenario of interest. As can be seen in Table 5, the average CPU usage is only 72.2% of the available 4 cores, more than a quarter of the processing power is unused. Which seems to indicate that there is underutilization of the CPU resources on the target system. In order to determine how the work is distributed, the CPU usage is measured during the execution per core. The result, seen in Figure 27, shows that the work is evenly distributed across all cores. While there are significant jumps in CPU usage for the individual cores, the average stays relatively consistent with a standard deviation of 2.5%.
To be able to analyse in more detail how the execution is mapped to 4 cores the program is profiled with all scheduling switches made by the operating system. A Python script (see Appendix C: Developed Performance Analysis Tools) has been written to parse the output and generate a visualization with Grasp [2]. The result is shown in **Figure 28**, where the execution is visualized for 150ms after the Reference Framework has processed more than a hundred frames. The green represents the program threads, orange represents idle time and blue other threads. The black areas indicate short periods of execution with many context switches. Zooming in on this area shows that the context switching is between threads belonging to the Reference Framework, **Figure 29**. Therefore useful work is done in the black areas. The graph is consistent with the observed CPU usage in **Table 5** and **Figure 27** as approximately 75% of the timeline is green.

The instrumentation code added during the porting allows the threads and steps of the application to be profiled in more detail. These results are given below.
Average CPU usage per thread

**Figure 30**: CPU usage per thread. Steps with data parallelism can use more than 100% CPU

*Figure 30* shows the CPU usage per thread. The steps exploiting data parallelism can have a CPU usage greater than 100% as work is divided across multiple cores. The majority of the work takes place in *CalculateSignalToNoise* and *CalculateShiftAndMotion* steps. Other threads only do a negligible amount of work.

Buffer occupancy at given frame

**Figure 31**: Buffer occupancy in number of frames for steps (threads) in AutoROI

The buffer occupancy graph shows the number of frames that are buffered at the input of the algorithm steps. The number of frames that can be buffered are limited to 30 frames in order to prevent saturation of the framework. In the graph above it can be seen that two steps experience a build-up in frames in their input buffer. Other steps are omitted from the graph since their buffers never have more than 1 frame occupancy during the execution. For *CalculateShiftAndMotion* the buffer initially fills up due to longer initial start-up execution, but the step is able to catch up due to the Reference Framework slowing down the input of frames. The *CalculateSignalToNoise* step on the other hand is unable to process frames quickly enough and its input buffer saturates.
Average step durations

**Figure 32:** Average time in microseconds for steps (threads) in AutoROI. 

**Figure 32** shows the average duration of the individual steps. It can be seen that the \textit{CalculateSignalToNoise} and \textit{CalculateShiftAndMotion} are the two most intensive steps of the algorithm as expected from the buffer occupancy. The Directed Acyclic Graph can now be annotated, **Figure 33**, by adding the average time each actor spent processing a frame as actor firing times.

**Figure 33:** DAG of AutoROI with average step durations on 4 cores.

**Scheduling of Threads**

To determine the order that the steps are scheduled, the instrumentation also monitors when steps are actively working on a frame. This allows visualizing when a step starts processing a frame and completes processing a frame. **Figure 34** shows the execution of 4 frames, where colours are matched to the frame being executed. The graph has the steps in order, with the bottom step being the first and the top step last. The first two steps are 30 frames ahead of the remaining steps due to the buffer build up in the \textit{CalculateSignalToNoise} step. Frames are executed in order due to the properties of the Completely Fair Scheduler that attempts to give threads fair access to the CPU.

**Figure 34:** Grasp timeline of thread frame execution.

**Figure 34** shows that the throughput of the algorithm is completely limited by the \textit{CalculateSignalToNoise} step. This step is constantly executing, as soon as it completes a frame the remaining steps can immediately execute generating the output. The analysis efforts now focus on this step.
Limits of Parallelism

If this step does not exhibit data parallelism then it is simply not possible to improve the performance. The analysis would end here. But, the step uses OpenMP to exploit the data parallelism, where the motion is extracted for each block in a frame. Now the analysis needs to determine how much parallelism can be exploited in this step.

Amdahl’s law states that the maximum speedup of a program that can be achieved through parallelism is limited by the proportion of the program that can be parallelized. [6] The law is often formulated as follows,

$$\text{Speedup}_N = \frac{1}{(1-p) + \frac{p}{N}},$$  \hspace{1cm} (5)

where \( p \) is the proportion of parallel work and \( N \) the number of cores.

The proportion of parallel work can be calculated by dividing the amount of parallel work by the sum of the parallel and serial work:

$$p = \frac{T_{\text{parallel}}}{T_{\text{parallel}} + T_{\text{serial}}}$$  \hspace{1cm} (6)

In order to apply Amdahl’s law to calculate the possible theoretical speedup, the \textit{CalculateSignalToNoise} step is profiled on a single core and the timing of the serial and parallelizable components are instrumented in order to calculate the ratio. The results are given in Table 6, note that this is the maximum expected execution time since the step is executed on a single core and therefore no parallelism is exploited. The results previously obtained were executed on multiple cores, where some speed up is seen.

\textbf{Table 6:} Timing for parallel and serial components of CalculateSignalToNoise

<table>
<thead>
<tr>
<th>Step</th>
<th>( T_{\text{serial}} )</th>
<th>( T_{\text{parallel}} )</th>
<th>Parallel Ratio (gCSN)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CalculateSignalToNoise</td>
<td>23 283\mu s</td>
<td>132 160\mu s</td>
<td>0.850</td>
</tr>
</tbody>
</table>

With all 4 cores the expected speed up is 2.76 times as calculated below. Assuming one core needs to be dedicated to the remaining tasks the speed up is limited to 2.31.

$$\text{Speedup}^4 = \frac{1}{(1-0.85) + \frac{0.85}{4}} = \frac{1}{0.3625} = 2.76$$

The actual measured speedup can be calculated as follows:

$$\text{Actual Speedup}_N = \frac{T_{\text{exec,1core}}}{T_{\text{exec,Ncores}}}$$  \hspace{1cm} (7)

\textbf{Table 7:} Comparison of theoretical vs actual speedup in CalculateSignalToNoise.

<table>
<thead>
<tr>
<th></th>
<th>Theoretical</th>
<th>Actual</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time Measured</td>
<td>3 cores</td>
<td>4 cores</td>
</tr>
<tr>
<td>Speedup</td>
<td>2.31</td>
<td>2.76</td>
</tr>
</tbody>
</table>

Besides the \textit{CalculateSignalToNoise}, two other steps make use of data parallelism. In order to understand whether these steps can benefit from the parallelism, the same calculations are repeated for these steps. Similarly to the \textit{CalculateSignalToNoise} step.
the time of the serial and parallel parts of these steps are measured on a single core to determine the parallel ratio.

Table 8: Timing for parallel and serial components for steps using data parallelism

<table>
<thead>
<tr>
<th>Step</th>
<th>$T_{serial}$</th>
<th>$T_{parallel}$</th>
<th>Parallel Ratio (p)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CalculateShiftAndMotion</td>
<td>263μs</td>
<td>79 382μs</td>
<td>0.997</td>
</tr>
<tr>
<td>CalculateBlockWeight</td>
<td>311μs</td>
<td>1 001μs</td>
<td>0.760</td>
</tr>
</tbody>
</table>

The current speedups obtained on the Reference Framework for these steps are given below and compared to the theoretical speedup.

Table 9: Speedup for other data parallelized steps

<table>
<thead>
<tr>
<th>Step</th>
<th>Theoretical</th>
<th>Actual</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3 cores</td>
<td>4 cores</td>
</tr>
<tr>
<td>CalculateShiftAndMotion</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time Measured</td>
<td></td>
<td>82 964μs</td>
</tr>
<tr>
<td>Speedup</td>
<td>2.98</td>
<td>3.96</td>
</tr>
<tr>
<td></td>
<td></td>
<td>64 553μs</td>
</tr>
<tr>
<td>Speedup</td>
<td>0.96</td>
<td>1.23</td>
</tr>
<tr>
<td>CalculateBlockWeight</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time Measured</td>
<td></td>
<td>2 733μs</td>
</tr>
<tr>
<td>Speedup</td>
<td>2.03</td>
<td>2.33</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3 124μs</td>
</tr>
<tr>
<td>Speedup</td>
<td>0.48</td>
<td>0.42</td>
</tr>
</tbody>
</table>

None of the steps with data parallelism is able to achieve speedups close to their theoretical maximum. This is due to the fact that cores cannot solely be dedicated to the parallel threads. The performance gain for some steps is even negative, possibly indicating that thread overhead and scheduling decreases performance.

By applying Amdahl’s law it has been shown that several steps have a high degree of potential parallelism, which is currently not exploited. The bottleneck step has frames continually available in the buffer for processing. The question now arises why CPU underutilization is seen. The analysis so far is only able to determine whether a thread is active on the CPU or idle. It cannot determine why a thread is idle or how efficiently work is done. In order to gain this understanding a more detailed method needs to be developed and applied to investigate the issues seen in this section.
5 Extended Thread State Analysis

The initial analysis has shown that the ported version of the Reference Framework achieves a moderate throughput and latency, but under-utilizes the CPU. In order to identify the reason for the under-utilization and identify possible changes that can improve the latency and throughput a more detailed and structural analysis is required. This chapter introduces and develops the idea of a generic investigative method that can be used to analyse and quantify the performance of application threads on Linux. A thread is the basic unit of work that can be scheduled by the operating system [8] and a great deal of information can be derived by observing the way the operating system schedules and manages threads.

The method focuses on determining the states of threads that indicate the availability of the thread for work, whether it is blocked or when work is executed. Based on the time a thread spends in these states several potential issues, such as bugs, overheads and unbalanced work distribution can be identified. Building on the existing Thread State Analysis method [1], the Extended Thread State Analysis method introduces several additional states that can be used to analyse threads as well as visualize the execution. The analysis is also quantified by introducing metrics to assess the efficiency and overhead of threads.

While the existing Thread State Analysis method has been partially documented for operating systems such as Solaris, no documentation on how to apply this method to Linux is available [1]. This section first develops and extends the thread state analysis and explains how the states can be deduced and profiled on Linux. Following this the analysis is further extended to include hardware and kernel states such as context switching, page faults and cache stall overheads. To be able to better interpret the analysis user level events are introduced that allow the kernel profiling events to be synchronized with application events such as the arrival of frames or the start of parallel regions. The results of this analysis is given as charts decomposing the percentage of time threads spent in each state as well as tools to visualize the timeline of thread execution. Finally, the steps to apply this analysis is given in the last subsection of this chapter.

5.1 Expanding Thread States

In order to be able to benefit from the analysis it is desired to derive as much useful state information as possible. The current TSA method has limited state information. In particular there is no distinction for the reason a thread is removed from the processor and placed on the run-queue. In this section the automaton that models the states of the threads is introduced and expanded with several additional states. This expansion answers the question on why a thread is waiting for access to a core.
Figure 35: Extending operating system threads states. States surrounded by blue boxes are new sub-states.

In Figure 35 the states that a thread may exhibit during its lifetime, as defined by this analysis, are given as a finite state automaton. These states are based on the states defined in the Thread State Analysis, see 2.4, as well as the states defined by the Linux kernel, see 2.5.2. The automaton in the Thread State Analysis is extended by decomposing the runnable state into three sub-states, indicated by blue borders.

In a typical lifetime a thread will be created, put on a run-queue to be scheduled to run, given a time slice on a processor after which it may voluntarily or involuntarily yield and be rescheduled. The process repeats as long as the thread has work to do. After the thread has completed its work it will return and be terminated after which it enters a zombie state where the parent thread will join on it. The purpose of the states as well as the transitions that cause state entry and exit are explained below.

The **NEW** state is entered once at the creation of the thread when the parent spawns a new thread. Once the thread is created it will be woken and placed on the run-queue and enter the runnable state.

The **RUNNABLE** state is entered once the current thread wakes up from a voluntary yield, where either the thread yielded its time-slice on the processor to sleep, or previously blocked itself due to an I/O or blocking kernel call. Threads that enter the runnable state are considered to have previously voluntarily yielded the processor, rather than involuntarily, in order to wait for an event or sleep. A voluntary yield implies that the thread, instead of the scheduler, decided to switch the thread off the CPU. In contrast to the existing thread state analysis the Extended Thread State Analysis makes this distinction explicit.

The **EXECUTING** state is entered once the thread is assigned to one of the processors and allowed to do meaningful work: The instructions belonging to the thread are executed. During execution the thread can either voluntarily yield or block or be involuntarily removed from the processor by the scheduler. The time-slice and influence of priority depends on the scheduling policy of the thread as investigated in the background info section of this thesis, see 2.5.2.

The **ZOMBIE** state is entered when the thread has been terminated. This occurs when the thread has completed its work or aborts. The thread state will exist in the data
structures until the parent thread has acknowledged the termination by joining on the thread.

The **SLEEPING** state is entered when a thread yields and waits for a signal. This could either be a timed yield, where a thread asked to be rescheduled at some future time point, or blocking, waiting for a mutex to become available or conditional variable to be satisfied. Typically mutexes are used as locks to protect multithreaded access to data or as barriers to synchronize threads [9]. These events will therefore place a thread in the sleep state.

The **BLOCKED** state is entered when a thread yields and marks itself as uninterruptible. In this state the thread cannot be woken by signals. This typically occurs when a thread calls a kernel system call that waits directly on hardware events [10].

The **I/O WAIT** state is entered whenever the thread makes a request to data that is not currently available in main memory or cache memory and initiates an I/O request. The thread will remain in this state until the I/O request is resolved after which a signal is generated and the thread is scheduled to react on it. I/O requests are typically some of the slowest operations in a system.

The states described so far are typically applied in Thread State Analysis. To extend the analysis two additional states are defined that indicate involuntary, i.e. scheduler induced, yielding. These states make a distinction on why threads are involuntarily removed from the processor. And answer the question of why a thread is waiting for access to a core.

The **READY QUANTUM** state is entered when a thread is removed from the processor by the scheduler because it has exhausted its time-slice, or quantum, and other threads are waiting in the runnable state. The duration of the time-slice depends on the configuration of the scheduler and the policy.

The **READY PRE-EMPT** state is entered when a thread is removed from the processor by the scheduler because a higher priority thread has become available to pre-empt the current running thread. The order of pre-emption and when pre-emption occurs depends on the scheduling policy of the threads.

The addition of the last two states make it possible to analyse the scheduling configuration, priorities and policies in more detail to determine why a thread is waiting for processor time. These states make it possible to determine if a thread was replaced due to its priority and the properties of the scheduler.

### 5.2 Determining Thread States

In order to be able to practically apply the extended thread state analysis it needs to be investigated how the thread states can be derived from the event profiling that is recorded by the Perf tools. This section investigates this profiling data and how the states can be derived based on it.

The Linux kernel and scheduler only maintain the minimum amount of states needed in order to do their work efficiently. As a result interpretation needs to be performed to identify the states of interest. The Perf profiling can be used to record the scheduling decisions made by the operating system over time. The result of this profiling session is hundreds of megabytes of binary data that can be parsed into a human readable
format. In order to derive the states an experimental approach is taken where a simple program is written to exercise these states. In Figure 36 the different transitions that can be deduced from the Perf events are annotated. The events that indicate these transitions are explained in this section.

Figure 36: Relating transitions to Perf profiling events

The transition marked by A indicates the creation of a new thread by a parent thread. Perf generates a sched_process_fork event whenever a new thread is created. This event contains the name and unique thread ID of the newly created child thread as well as the parent ID, see Figure 37. Once this event is generated the child thread will remain in the NEW state until it is woken up.

Figure 37: Perf process fork event (A)

Once a thread has been created it will be moved to the run-queue and the RUNNABLE state when woken up, indicated by B. The first time this happens a sched_wakeup_new event is recorded, which contains the name and ID of the thread, its priority, whether the thread was successfully woken and which CPU queue the thread is placed on, see Figure 38.

Figure 38: Perf wakeup new event (B)
The main event that is recorded by the Perf tools is the *sched_switch* event. This event is recorded whenever a thread is removed from a processor and replaced with another thread. Based on this it can be determined when a thread is moved to a processor, transition C as well as the reason that the outgoing thread is removed from the processor, transition E, F, G, H and I.

![Diagram of Perf sched switch event](image.png)

**Figure 39:** Perf sched switch event (C, E, F, G, H, I)

The incoming thread will transition from either the runnable, ready quantum or ready pre-empt to the executing state.

For the outgoing thread there are several states that the thread may transition to. The decision tree to determine the next state transition is given in **Figure 40**. The next state of a thread mainly depends on the Linux process state. These flags determine whether the thread is entering the sleeping, blocked or I/O wait states. Through experimentation it was found that the Linux process state enters the uninterruptible state with the waking flag set for I/O requests. Similarly it was found that a process enters the interruptible state when sleeping or waiting for a mutex. A thread that leaves with the thread state defined as running can enter the run-queue as one of two states in the Extended Thread State Analysis. If the priority of the incoming thread is equal or lower than the outgoing thread than the thread is removed because it has exhausted its time-slice. If the priority of the thread that is replacing the current thread is higher, then the thread is pre-empted by the incoming thread. This is the case for the FIFO and Round Robin policies. For the Completely Fair Scheduler, this indicates that the thread is replaced by a thread with a lower nice value.
When a thread completes the profiling records a `sched_process_exit` event. The thread transitions to the ZOMBIE state, transition D, for the remainder of the program execution. This event includes the name and unique thread ID of the terminating thread, see *Figure 41*. 

*Figure 40*: Decision tree to determine next state of outgoing thread.
The next transition, J, occurs when a thread that voluntarily yielded is returned onto a run-queue and the RUNNABLE state. Similar to transition B a `sched_wakeup` event is recorded when a thread wakes up for the signal or event it was waiting for, see Figure 42.

If the scheduler decides due to, for instance load balancing, that a thread on the run-queue needs to be migrated to a different processor a `sched_migrate_task` event is recorded. This event does not cause a state transition in the model, but may cause significant cache stalls since the thread is moved to a cold cache. The event records the name and ID of the thread as well as the run-queue of the previous core and the new core, see Figure 43.

Based on these events, and their relation to the transitions in the state machine, scripts were written that can parse the files generated by Perf. These scripts can visualize the states of threads during the execution of the application using Grasp, as well as calculate the total time threads spend in the states for analysis and identification of issues.
5.3 Derived Hardware and Kernel States

So far the analysis has focused on how the scheduler perceives and schedules threads. As far as the operating system is concerned once a thread is scheduled and moved to a processor it consumes 100% of the processor cycles to execute its instructions. In reality hardware events and other kernel events can prevent efficient use of processor time by a thread. This leaves the question on how useful on core time is. This section investigates several of these issues and extends the thread state model to include these issues in the analysis.

![Figure 44: Thread state model expanded with, in orange boxes, the executing state split into sub-states for context switching, page faults and cache stalls.](image)

In Figure 44 the thread state model is expanded to consider additional states that can be derived from kernel and hardware level profiling. These states are added to allow investigation into how useful the time that a thread spends on a processor is. Several issues such as excessive cache stalls, page faults or context switching can reduce the work a thread performs on the core. These issues are more of a concern in embedded platforms with fewer cores that may require more context switching, have smaller page tables and smaller cache memories with simpler hierarchies. The Extended Thread State Analysis therefore considers these events as states a thread will experience during execution. The EXECUTING state is decomposed and 3 additional states are introduced. The description of these states as well as the formulas to calculate the overhead these events introduce are given below.
The **CACHE STALL** state represents the approximate amount of time that a thread spends with the processor stalled due to cache misses. According to Moore’s law the amount of processors on a chip double approximately every two years, but the speed of memory has increased by only about 7% per year [12]. This has introduced a processor-memory gap where access to memory outside the processor registers is slower than the instruction cycle time. To reduce the effect of this gap many processors now have hierarchical cache memories with smaller and quicker memories closer to the processor. The cache hierarchy as well as the sizes and timing for the target hardware used in this project is given in **Figure 45**. The timing is derived from the benchmarks run on the Raspberry Pi.

<table>
<thead>
<tr>
<th>Processor Registers</th>
<th>Size</th>
<th>Access Speed</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 data + instruction</td>
<td>16 KB + 16 KB</td>
<td>~2.5 ns</td>
</tr>
<tr>
<td>L2 (Last Level Cache)</td>
<td>512 KB</td>
<td>~6.5 ns</td>
</tr>
<tr>
<td>RAM (Main Memory)</td>
<td>1 GB</td>
<td>~30 ns</td>
</tr>
<tr>
<td>SD card (Disc)</td>
<td>8 GB</td>
<td>~5.5 ms</td>
</tr>
</tbody>
</table>

**Figure 45**: Cache hierarchy with sizes and access times for the Raspberry Pi 3.

If data is not in the processor registers it is hierarchically fetched from the next level caches or main memory. If the requested data is not available in the main memory an I/O request will be performed and the thread will be context switched off the core and enter the I/O WAIT state.

Caches typically make use of spatial and temporal locality of data to reduce the miss rate. Spatial locality takes advantage of the fact that if memory is accessed at a certain location the neighbouring memory is likely to be accessed next. This is done by for example prefetching data when a cache miss occurs. The amount of data that is fetched is determined by the cache line width, for the ARM processor in the Raspberry Pi the line width is 64 bytes [11]. The cache replacement policy can also take advantage of temporal locality, i.e. data that was recently accessed is likely to be accessed again, by replacing the Least Recently Used cache entry. The ARM processor in the Raspberry Pi, however, uses a pseudo-random policy that does not take advantage of the temporal locality [11].

The approximate amount of time that a thread spends stalled waiting for the cache can be calculated as follows, where \( L1i \) and \( L1d \) represents the Level 1 instruction and data caches respectively and LLC represent the shared Last Level Cache:

\[
T_{\text{CACHE STALL}} = (#L1i_{\text{misses}} + #L1d_{\text{misses}}) \times t_{L2\text{latency}} + #LLC_{\text{misses}} \times t_{\text{RAM latency}} \tag{8}
\]

The latencies for the L2 cache and RAM access are based on benchmarks run on the Raspberry Pi, see Appendix B: Raspberry Pi 3 Hardware Benchmark Results. The amount of cache misses for the different levels of cache can be recorded for each thread using the Perf Stat utility. This utility records the hardware counters for cache misses, loads and stores.
The **PAGING** state represents the amount of time a thread spends resolving minor page faults. Linux uses a virtual memory system that assigns processes a large virtual memory space. This virtual memory space abstracts the physical memory from the process. To correctly address memory the virtual memory addresses used by a process need to be translated to the physical address. In order to do this efficiently a page table is used that acts as a cache of these translations. A minor page fault occurs when the requested page exists in the physical memory, but has been removed from the page table [13]. The kernel will translate the address and cache the result. A minor page fault does not result in an I/O request to disk and the process will not be removed from the processor. When the requested page does not exist in physical memory a major page fault is generated and an I/O request will be initiated. In this case the process will be removed from the processor and enter the I/O WAIT state.

The formula to calculate the time a thread spends resolving minor page faults is given below:

\[ T_{PAGING} = \#\text{MinorPageFaults} \times t_{\text{MinorPageFault}} \]  

The time to resolve a minor page fault is based on the benchmarks that were ran on the Raspberry Pi, see Appendix B: Raspberry Pi 3 Hardware Benchmark Results. The amount of page faults are recorded for each thread using the Perf Stat utility.

The **CONTEXT SWITCH** state represents the time that is spent on persisting the context of the thread when it is pre-empted from running on a processor and replaced with another thread. This includes the time required to persist the CPU registers, the stack pointer and thread program counter. The thread state analysis so far cannot determine whether context switching overhead is a significant issue, as a thread that experiences many context switches can spend the same amount of time in the EXECUTING state as a thread that experiences barely any context switches. An example of this situation is given in **Figure 46**, where Thread B has a significant amount more context switches than Thread A, yet the total CPU usage is 50% for both threads. The more a thread is context switched, the more it will interfere with the scheduling of other threads.

![Figure 46](image)

**Figure 46**: Example of how threads with that is context switched many times (Thread B) can report the same CPU usage as a thread that is barely context switched (Thread A).

The approximate amount of time the scheduler will spend on context switching a thread can be calculated as follows, where \( #CS_{\text{voluntary}} \) and \( #CS_{\text{involuntary}} \) represents the number of voluntary, yielding, context switches and involuntary, pre-empted, context switches respectively:
\[
T_{\text{context switch}} = (\#CS_{\text{voluntary}} + \#CS_{\text{involuntary}}) \times t_{\text{context switch}}
\] 

The time for a single context switch, \(T_{\text{context switch}}\), is based on the benchmarks run on the Raspberry Pi, see Appendix B: Raspberry Pi 3 Hardware Benchmark Results. The amount of voluntary and involuntary context switches are measured per thread using the Perf Stat utility.

Unlike the scheduling decisions profiled by Perf, these events are not individually recorded but are accumulative counters captured by the profiling tools. As such it is not possible to visualize these events, simply due to the sheer volume and profiling overhead this would introduce. With the extension now complete, all the Perf events that are recorded during the Extended Thread State Analysis are now given in Figure 47, annotated in orange. The analysis records both the kernel trace points for scheduling decisions, context switches, core migrations and page faults. As well as hardware Performance Monitoring Counters, or PMCs, for references and misses of different levels of cache memory.

![Linux perf_events Event Sources](http://www.brendangregg.com/perf.html 2016)

**Figure 47:** Linux system with used events in the Extended Thread State Analysis annotated [33].

### 5.4 Synchronizing User and Kernel Events

The method presented so far is generic and can be applied without any code modification or recompilation. However, in order to better understand how the application threads perform their work it is beneficial to add user level events that are synchronized to the events profiled by Perf. This section describes the addition of these user level synchronization points to the Reference Framework.
To be able to support user level events the instrumentation developed during the porting phase is extended to allow profiling of user level events. This is illustrated in Figure 48 where the synchronization data structures are indicated in red. The user events are merged with the events generated by Perf.

**Figure 48:** UML class diagram of Instrumentation with Perf event recording synchronization in red

The synchronization code marks 4 synchronization points with kernel level event recording.

- **PERF SYNCH FRAME START**
  - This event marks the start of processing of a frame.

- **PERF SYNCH DATA PARALLEL START**
  - This event marks the start of a parallel region during processing. All the children threads of the thread generating this event will wake up and process blocks of the data in parallel.

- **PERF SYNCH DATA PARALLEL END**
  - This event marks the end of a parallel region during processing. All the children threads of the thread generating this event will synchronize and sleep.

- **PERF SYNCH FRAME END**
  - This event marks the end of processing of the current frame. The thread generating this event will return to idle, SLEEPING state, until arrival of the next frame.
Figure 49: Pseudo-code of frame processing on left (a) and timeline on right (b) of user annotated events

Figure 49 provides a pseudo-code example (a) of how these synchronization points are added to mark these events for a thread. The timeline of the figure (b) shows where the thread should be idle, in the SLEEPING state, (white) or processing a frame (green and blue) based on the synchronization points. The synchronization points are also used to indicate the serial (green, squares) and data parallelized (blue, vertical lines) regions of a thread. The instrumentation generates a log file that records the event type, the frame being processed, the thread it belongs to and a timestamp based on the same monotonic clock used by Perf. This timestamp can then be synchronized with the events recorded by Perf to annotate events during visualization.

With the addition of user level events it is now possible to accurately determine when a thread has useful work to do. Based on the events threads can be in one of two states from a user perspective:

- Idle (white). During the idle period a thread does not have any pending data to process. It is expected that the thread will sleep and be triggered to wakeup upon the arrival of a frame. Spending a significant time in either the RUNNABLE state or EXECUTING state indicates that the thread is doing work outside its main concern of processing frames.
- Work Pending (green and blue). During the work pending period a frame has arrived and is being processed. It is expected that the thread will either be executing or placed on a run-queue to be scheduled for execution until processing of the frame is complete.

The user level events can be annotated and overlaid in the visualization of the thread states to simplify analysis.
5.5 Quantifying Thread State Occupation

During the lifetime of a thread, it may enter some states more than several million times. In order to be able to simplify the analysis a state occupation graph is created. This graph shows the accumulated percentage of time each thread spends in the individual states, which allows problematic states and potential issues to be identified with a simple glance. The scripts in the analysis calculate these percentages as follows.

The duration of time a thread spends in a state is calculated as follows. The state exit and entry timestamps for the \( i \)th time thread \( p \) enters and exits state \( x \) are denoted as \( t_{exit_{xp_i}} \) and \( t_{entry_{xp_i}} \) respectively. These timestamps are derived from the Perf events described earlier in this section. The total time thread \( p \) spends in state \( x \) can therefore be calculated by:

\[
T_{xp} = \sum_{i=1}^{\text{#state\_entries}} (t_{exit_{xp_i}} - t_{entry_{xp_i}})
\]  

(11)

For calculating the duration of the executing state the occupation time is not solely determined by the scheduler events, but also considers overheads outside of the scheduler’s control. The time a thread therefore spends executing its user-land program instructions can be calculated as follows where \( T_{\text{CACHE\_STALL}} \), \( T_{\text{PAGING}} \) and \( T_{\text{CONTEXT\_SWITCH}} \) are from equations 8, 9 and 10 respectively.

\[
T_{\text{EXECUTING}_i} = \sum_{i=1}^{\text{#state\_entries}} (t_{exit_{EXECUTING_{pi}}} - t_{entry_{EXECUTING_{pi}}}) - T_{\text{CACHE\_STALL}_i} - T_{\text{PAGING}_i} - T_{\text{CONTEXT\_SWITCH}_i}
\]  

(12)

The derived hardware and kernel states described in 5.3 do not have associated Perf timestamps. The time a thread spends in the CACHE STALL, PAGING and CONTEXT SWITCH states are therefore calculated from Equation 8, 9 and 10 respectively.

The state occupation percentage for state \( x \) of thread \( p \) is given in Equation 13, where \( T_{total} \) is the total program execution time.

\[
\%_{x_p} = \frac{T_{xp}}{T_{total}} \times 100\%
\]  

(13)

The state occupation percentage is visualized using a stacked bar graph. This bar graph plots threads of a program together in order to identify anomalies.
5.6 Visualizing Thread States

The quantification of thread states allows anomalies, such as excessive time spent in an unexpected state, to be identified at a glance. But in order to be able to identify patterns and interference between threads the analysis also allows interactive visualization of the thread states over time. This section describes the Grasp script that is created to allow this visualization.

The Grasp script graphs the state entry and exit timestamps captured with Perf. In addition to this the user level events captured by the instrumentation in the framework and described in 5.4 are annotated on the visualization.

Each thread is a separate task in the Grasp script. This allows Grasp to plot the task as a separate line. A task for thread \( p \) is declared as follows:

```
newTask p -name "p"
```

To indicate the start of state for a thread a new job is created and started for each state entry. The \( i^{th} \) time thread \( p \) enters state \( x \), defined as \( t_{entry_{sp}} \) in the previous section, is therefore annotated as a Grasp event below.

```
newJob p.x_i p -name "p i"
plot t_{entry_{sp}} jobStarted p.x_i
plot t_{entry_{sp}} jobResumed p.x_i
```

At the state exit the job is marked as completed. The \( i^{th} \) time thread \( p \) exits a state \( x \), defined as \( t_{exit_{sp}} \) in the previous section, is annotated as a Grasp event below.

```
plot t_{exit_{sp}} jobCompleted p.x_i
```

The analysis now visualizes the thread states of the individual threads over time, but in order to gain better understanding the user level events can be annotated. These events indicate the arrival and processing of frames, as well as the parallel regions as described in 5.4.

The arrival of a new frame for processing by a thread is indicated by an arrow on the visualization and is recorded as a job arrival in Grasp. The arrival of the \( n^{th} \) frame for processing by thread \( p \) at time \( t_{arrival_{pn}} \) is annotated as the Grasp event below.

```
plot t_{arrival_{pn}} jobArrived p_n p
```

The parallel regions are indicated on the visualization by vertical red lines that connect the tasks that process data in parallel. The start of a parallel region by thread \( p \), involving both thread \( p \) and \( p' \), on frame \( n \) at time \( t_{parStart_{pn}} \) is annotated as a Grasp event below.

```
plot t_{parStart_{pn}} line p p' -color red -stem solid
```
The end of a parallel region is similarly visualized using vertical red lines connecting tasks. The end of a parallel region by thread \( p \), involving both thread \( p \) and \( p' \), on frame \( n \) at time \( t_{\text{parEnd}} p_n \) is annotated as a Grasp event below.

\[
\text{plot } t_{\text{parEnd}} p_n \text{ line } p p' \text{ -color red -stem solid}
\]

The duration of the processing of a frame by a thread, the work pending period, is indicated by a blue line above the task during the entire duration of processing. The duration for processing of frame \( n \) by thread \( p \) is annotated using the Grasp event below, where the start time and end time are given by \( t_{\text{arrival}} p_n \) and \( t_{\text{end}} p_n \) respectively.

\[
\text{plot } t_{\text{arrival}} p_n, p t_{\text{end}} p_n \text{ -arrow none -color blue -stem solid}
\]

An example of the user level annotated events are given in Figure 50 where the arrow indicates job arrival, red vertical lines the parallel regions and blue horizontal lines the frame processing time.

![Figure 50: Example of Grasp visualization showing frame arrival (arrow), parallel regions (red vertical lines) and frame processing duration (blue horizontal line).](image)

The derived hardware and kernel states described in 5.3 do not have associated Perf timestamps, but rather an accumulative count, and can therefore not be visualized.

### 5.7 Methodology Workflow

This section presents the workflow that is used to apply the Extended Thread State Analysis described in this chapter. In order to apply the analysis the application has to be run with Perf profiling and the data processed to create the output described in the previous sub sections. The analysis can be used to identify anomalies and make observations regarding the execution of the application threads. Based on these observations improvements to the target application can potentially be made. This process is repeated to systematically improve the performance.

The workflow is illustrated in Figure 51 and consists of three stages. First, the application is implemented and the correctness validated. Once this is completed in the second stage the application is analysed and if performance or efficiency issues are seen the Extended Thread State Analysis is applied. Based on the analysis observations are made for further investigation. In the third stage these observations are investigated and potential improvements and optimizations are applied and the correctness is once again validated. The second and third stages are repeated until the desired performance is achieved or no more observations can be made that can be used for further investigation.
Figure 51: Flowchart of Extended Thread State Analysis method. Grey shaded boxes apply the Extended Thread State Analysis Method.

The three different stages are also reflected in this thesis where the implementation and correctness is described in Chapter 3. The application performance is analysed, in Chapter 4, where the CPU usage was seen to be underutilized. The Extended Thread State Analysis is therefore applied in the next chapter, Chapter 6, where several observations are made. These observations are investigated and optimizations/improvements are repetitively applied, validated and analysed until the CPU usage is no longer significantly underutilized, Chapter 7.
6 Applying the Analysis to AutoROI

The method developed in the previous section is well-suited for use on the Reference Framework where typically dozens of threads are forked by the framework and algorithms. In this section the Extended Thread State Analysis is applied to the ported Reference Framework and AutoROI, the motion based respiration algorithm that was analysed in previous sections. In the initial performance analysis it was determined that the CPU was underutilized and that parallel work was available but unexploited. Based on the analysis in this section several points of improvement are identified and implemented in subsequent chapters.

To be able to comprehensibly apply the method it is necessary to include the threads belonging to the Reference Framework in the analysis. The Directed Acyclic Graph is updated in Figure 52 with these threads included. The two threads that are added, *InputAgent* and *ProcessAgent*, are responsible for reading input from file and managing the algorithm threads respectively. Additionally two threads, the *Core* and *ApplicationApp* threads, are created by Linux. These threads simply start the other threads and perform no work during algorithm execution. The threads that belong to the framework infrastructure are shaded in grey in the image.

The initial ported application is re-run with profiling and the analysis is applied. The experiment profiles the entire processing of a video consisting of more than 1800 frames from the start of the application thread until all threads and the application are terminated. The results for the throughput, latency and CPU usage that was achieved in the initial analysis with the default scheduler configuration is given in Table 10.

**Table 10: Performance of initial ported version**

<table>
<thead>
<tr>
<th>Initial ported version</th>
<th>Throughput</th>
<th>Latency</th>
<th>CPU usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.58 FPS</td>
<td>174.30ms</td>
<td>72.2 %</td>
<td></td>
</tr>
</tbody>
</table>

The output of the Extended Thread State analysis provides a graph with the state occupation percentages as well as an interactive timeline visualization using Grasp. In this section the output of the method is provided and analysed to identify issues and points of improvement. In Figure 53 the state occupation percentages for each thread is given when AutoROI is executed. Some steps have up to four threads due to the additional threads created by OpenMP that exploit data parallelism in the parallel for loops, these threads are indicated with a post-fixed number.
The figure shows the most intensive threads on the right and more idle threads on the left. The threads on the left sleep most of the time and barely execute or enter the runnable queue or any potentially undesired states. The only thread that spends any time in the I/O wait state is the InputAgent that is responsible for reading the input video from disk. The most overheads are experienced by the ProcessAgent and the threads belonging to the CalculateSignalToNoise and the CalculateShiftAndMotion algorithm steps. These algorithm steps were shown to be the most intensive threads in the initial analysis.

The data parallelized threads of these steps spend approximately the same percentage of time in the executing state. This is also true for the main CalculateShiftAndMotion thread, but the main CalculateSignalToNoise thread spends significantly more time executing than its children threads. This reflects the ratio of parallel work seen in previous chapters where the amount of parallelism was calculated to be 99% for CalculateShiftAndMotion and 85% for CalculateSignalToNoise. These threads are mainly removed from the CPU and placed in the runnable queue because they exhaust their time slices and are not pre-empted by other higher priority threads. They also spend time in the run queue due to waking up from sleep in order to process newly arrived frames or to start new parallel regions.

All threads experience cache stalls, for the CalculateShiftAndMotion this is about 6% of the total time and for CalculateSignalToNoise only around 2.5%. The relatively high cache stalls of the CalculateShiftAndMotion is attributed by Perf to instruction cache misses, optimizing code for size in this step could reduce this. The thread that uses the CPU least efficiently is the ProcessAgent. This thread spends more than 12% either context switching or cache stalling. The high context switch overhead and the low percentage of time slice exhaustion indicates that the thread mainly executes for very short periods. This overhead and percentage of execution is also not expected for the ProcessAgent since it is expected to periodically wake up, prepare and route video frames to the algorithm.

The state occupation graphs provide an overview of where threads spend their time during the lifetime of the application execution. It is useful at identifying problematic states at a glance that may require further investigation. In order to better understand how threads interact and influence one another the application threads and their states are visualized using an interactive Grasp timeline. This timeline is given in Figure 54.
Figure 54: Timeline visualization of thread states for AutoROI; arrows indicate frame arrival, vertical red lines indicate parallel regions and blue lines indicate active frame processing. Black regions indicate rapid state switching.

The timeline shows the application threads processing two video frames. Frames are processed through the framework in order from bottom, InputAgent, to the top, ResultsWriter. The arrows indicate when frames arrive and red vertical lines indicate parallel regions. The timeline in combination with the state occupation graph allows better analysis to identify the cause of issues. Figure 55 gives the timeline annotated with observations worthy of further investigation. The remainder of the analysis shortly describes these annotations.

Figure 55: Grasp timeline of thread states with identified issues annotated.
Observation A: Unnecessary data parallelism in `CalculateBlockWeight`

The first observation derived from the timeline, annotation A, is the inefficiency of the data parallelism of the `CalculateBlockWeight` threads. These threads spend 0.7% of the time executing during the entire run, but over 2% waiting to run. The timeline shows how the relatively little work performed by the `CalculateBlockWeight` step is delayed by the data parallel threads waiting to run. This delay slows down the delivery of output of individual frames and increases the latency. This was also observed in Table 9 where the speedup was less than 1 when data parallelism was introduced. A simple fix to potentially reduce this latency is to remove the data parallelism of the `CalculateBlockWeight` step, this will eliminate the synchronization of these threads.

Observation B: `ProcessAgent` excessive wakeup and interference

As previously seen in the state occupation graphs the `ProcessAgent` experienced high cache stalls and context switch overhead. In the timeline it can be seen that the thread wakes up excessively for very short periods, annotation B. Zooming in on the black areas shows the thread sleeping, waking up, shortly executing and yielding to sleep again, Figure 56. The `ProcessAgent` thread causes interference with several other threads. These are the threads that are placed in the same processor run queue. The scheduler and load balancer try to keep threads assigned to the same processor during its execution. Migrating a thread to a different processor incurs an overhead cost due to the cache misses that may occur on the cold cache. A thread is only allowed to be migrated after the thread could not be executed on a specific core for a period equal to the migration cost period, defined as 500us, after which the cache would be considered cold by the load balancer [32]. This explains why different threads are influenced by the `ProcessAgent` over time. Since the performance of the `ProcessAgent` is unexpected behaviour, further investigation needs to be performed.

Observation C: Parallel regions synchronization and idle threads

Another observation from the timeline analysis shows how the data parallel threads do not finish their work at the same time, but end up sleeping and waiting for the slowest thread to complete its work, annotation C. The main thread cannot continue execution until all the data parallelized threads complete their work. The execution is delayed by the slowest data parallelized thread, the thread that experiences the most interference. This interference is caused by the data and functional parallelism in the framework and the CPU limitations of the platform. The data parallel threads are interrupted due to exhaustion of their time slices and threads of other steps are allowed to run. This observation applies to all the steps that use data parallelism; `CalculateSignalToNoise`, `CalculateShiftAndMotion` and `CalculateBlockWeight`. Eliminating this idle time will improve the latency and throughput of the output.

Observation D: Work distribution

Lastly it can be observed, annotation D, that there are periods where all the intensive threads try to perform work, the threads belonging to the `CalculateSignalToNoise` and, `CalculateShiftAndMotion` steps. The scheduler attempts to divide these threads across the 4 cores and the threads are scheduled out due to exhaustion of their time slices.
Furthermore there are idle periods where not all of the CPU cores are active, because the *CalculateShiftAndMotion* completes its work during this intensive period.

It appears that restructuring the work distribution, in order to reduce this interference, will likely improve the latency and throughput.

**Summary of observations:**

A. Unnecessary data parallelism in *CalculateBlockWeight*
   Solution: Remove data parallelism in *CalculateBlockWeight.*

B. Excessive wakeups and interference in *ProcessAgent.*
   Solution: Investigate code for implementation error or bugs.

C. Idle thread time and synchronization in parallel regions.
   Solution: Improve parallel work distribution and flexibility.

D. Scheduling of threads results in interference and idle periods.
   Solution: Improve work distribution through scheduling priorities and algorithms to make use of idle periods

This section applied the Extended Thread State Analysis to the Reference Framework and AutoROI. With the help of this analysis insight was gained into the execution and behaviour of the individual threads. It is now clear why the initial analysis showed CPU underutilization and underperformance. The state occupation graph and interactive timeline allowed identification of several potential issues that warrant further investigation. These observations have introduced several questions that can be investigated to improve performance according to the GQM-approach. The next section investigates these observations and applies optimizations and fixes to improve the performance.
7 Improving Performance

In the previous section the analysis presented in thesis was applied to the implementation of AutoROI on the embedded platform. Several observations were made based on the state occupation percentages and the thread state timeline. In this section these observations are investigated and optimizations are implemented to improve the throughput and latency of the AutoROI algorithm. After each optimization the analysis is repeated to verify that the optimization is correctly applied.

7.1 Remove CalculateBlockWeight Parallelism

The first observation made in the previous section is the unnecessary use of data parallelism in the CalculateBlockWeight step of AutoROI. The analysis showed that the threads of the step perform relatively little work, yet incur delays due to the synchronization and scheduler interference as indicated by the time these threads spent in the run-queue. This optimization can easily be applied by removing the parallel directive that allows OpenMP to divide the loop iterations across all the data parallel threads. This optimization is expected to improve the latency since the synchronization delay is removed. The results are given in Table 11 where the latency is seen to improve by 2.5%.

<table>
<thead>
<tr>
<th>Removing CalculateBlockWeight parallelism</th>
</tr>
</thead>
<tbody>
<tr>
<td>Throughput</td>
</tr>
<tr>
<td>9.64 FPS (+0.6%)</td>
</tr>
</tbody>
</table>

The thread state analysis is rerun to verify that the optimization is correctly applied. The state occupation graph and timeline is given in Figure 57 and Figure 58 respectively.

![State Occupation Percentages](image)

Figure 57: State occupation percentages with CalculateBlockWeight data parallelism removed.

The analysis now shows that the data parallel threads of the CalculateBlockWeight are no longer created. The remaining CalculateBlockWeight thread now exhausts some of its time slices and spends about 3% of the time in the ready quantum state. This is due to the increased computational requirements of the thread since the work is no
longer spread across the data parallel threads. The thread now spends more than 3 % of the time executing compared to 0.9 % previously.

![Timeline with CalculateBlockWeight data parallelism removed.](image)

**Figure 58**: Timeline with CalculateBlockWeight data parallelism removed.

In the timeline it can also be seen that the single thread, with the synchronization overhead removed, is able to complete its work quicker than in the previous analysis with the individual data parallel threads. The result is that the latency is improved as the total processing chain completes its work earlier.
7.2 Fix ProcessAgent

The next optimization is to resolve the issue experienced in the ProcessAgent. In the analysis this thread was seen to continually wake up and sleep resulting in interference with other application threads. During the lifetime of the application the thread would wake up several million times.

Investigating the code revealed that the timeout delay for the function that waits for frames from the InputAgent was incorrectly type casted. The function expected an double with the number of seconds, rather than an integer. The result of the bug was that a timeout of zero seconds was applied and the thread would yield and be woken up immediately. The offending line of source code is given Figure 59 with the required change shown.

```c
ptrVideoImage = (source.try_receive(success, static_cast<unsigned int>(imageReceiveTimeout)));
ptrVideoImage = (source.try_receive(success, imageReceiveTimeout));
```

Figure 59: Line of code in ProcessAgent with the bug (top) and the fix (bottom)

With the fix implemented the thread now correctly waits for the next frame with the correct timeout value. The results of the fix can be seen in Table 12 where the latency and throughput improved by about 10 % and the CPU usage decreased slightly.

<table>
<thead>
<tr>
<th>ProcessAgent fixed</th>
<th>Throughput</th>
<th>Latency</th>
<th>CPU usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.66 FPS (+10.6 %)</td>
<td>153.07ms (-9.3 %)</td>
<td>72.2 % (-1.7 %)</td>
<td></td>
</tr>
</tbody>
</table>

Table 12: Performance with ProcessAgent fixed

The analysis is rerun to and the state occupation graph and timeline is given in Figure 60 and Figure 61 respectively.

Figure 60: State occupation percentages with ProcessAgent fixed.

The analysis now shows that the ProcessAgent causes very little interference and is now one of the threads with minimal overhead. The thread sleeps for more than 97 % and spends barely any time, less than 0.07 % waiting to run. The thread executes for only 0.2 % compared to 21% previously. The overheads in cache stalls and context switching is reduced from nearly 12 % to less than 0.1 %.
Figure 61: Timeline with ProcessAgent fixed.

The timeline shows that the *ProcessAgent* thread wakes up once per frame, does some work and delivers the frame to the next step before returning to sleep. The thread now only context switches a total of 1911 times instead of more than 2 million times. The black lines, which previously indicated rapid context switching, are no longer present in the *ProcessAgent* or the other threads, indicating that the interference is significantly reduced.

The improved latency and throughput has shown that the issue is resolved and via the analysis confirmed that this is attributed to the *ProcessAgent*. 
7.3 Data Parallelization Load Balancing

In the analysis it was observed, Observation C, that even though the amount of work performed by the data parallel threads were equal, some threads would be delayed in executing their work. The slowest thread would keep executing while the other threads start sleeping waiting for synchronization to allow the main thread to continue execution. In this section a solution for this problem is given, implemented and analysed.

By default OpenMP uses predetermined scheduling that divides the work equally across all threads at compile time and assumes that threads will be able to process their work without interference. In reality interference by the operating system scheduler will result in data parallel threads being delayed in processing of their work as they are switched off the processor. This results in the data parallel threads sleeping and waiting to synchronize on the slowest thread, since the end of parallel regions require synchronization, delaying the output and resulting in CPU underutilization.

To improve the distribution of parallel work it is possible to implement dynamic allocation of work where the parallel work is divided into small chunks and processed by threads as they become available to process chunks. This allows idle data parallel threads to continue processing new chunks of data if other data parallel threads are pre-empted. In Figure 62 the difference between static and dynamic scheduling is illustrated showing how with dynamic scheduling previously sleeping threads can continue processing chunks when other threads are pre-empted allowing frame processing to be completed sooner. This solution does however introduce some overhead since work assignment is dynamically allocated at runtime instead of statically at compile time. By changing the chunk size it is possible to measure this overhead. These measurements showed that the overhead is negligible, no impact on latency or throughput, and therefore the smallest chunk size of one block is used.

![Figure 62: Illustrating the effect of thread pre-emption on static scheduling (left) and dynamic scheduling (right).](image)

The result of using dynamic scheduling is given in Table 13 where it is seen that the throughput and latency is improved by nearly 11 % and 6 % respectively. The CPU usage also increases by more than 11 %.

<table>
<thead>
<tr>
<th>Throughput</th>
<th>Latency</th>
<th>CPU usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>11.81 FPS (+10.8 %)</td>
<td>144.03ms (-5.9 %)</td>
<td>80.6 % (+11.6 %)</td>
</tr>
</tbody>
</table>

Table 13: Data parallel load balancing
The analysis is once again rerun to verify that the optimization results in the improvement. The state occupation graph and timeline is given in Figure 63 and Figure 64 respectively.

**Figure 63:** State Occupation Percentages with Data parallelization load balancing.

The analysis now shows, in Figure 63, that the data parallel and main threads of the intensive steps spend less time sleeping and more time executing. The main `CalculateSignalToNoise` thread now executes for more than 60% of the time and sleeps for only 2.5% of the time compared to 55% execution and sleeping for 16% of the time. The thread now experiences more switches due to exhaustion of its time slices, from 17% to 21%. Since the thread no longer has to wait for its children threads it can spend more time executing. Additionally, it is seen that the data parallel threads no longer do the same amount of work, this indicates that the threads are dynamically performing work and the effect of scheduler interference is reduced.

**Figure 64:** Timeline with Data parallelization load balancing.

The timeline, in Figure 64, shows that the data parallel threads finish at approximately the same time. In some cases there is a minor delay in synchronization, due to one of the threads being delayed in processing the last chunk of data. The increase in throughput and CPU usage can be attributed to the data parallel threads no longer sleeping during the parallel regions. However the interference between the data parallel threads still remain and the idle periods are still present. The optimizations now focus on improving this work distribution.
7.4 Thread Scheduling

The final optimization is to improve the scheduling of the threads. As observed in the analysis, Observation D, there are periods where the scheduler attempts to schedule more threads than available processors, as well as periods where multiple processors are idle. This indicates that work distribution can be improved. The scheduling of threads can be configured by changing both the scheduling policy and the priority of the threads. The priority of threads later in the processing chain are given a higher priority to "pull" frames toward the output. This strategy is similar to the approach taken by Weffers-Albu M. A. et al., however the focus is on improving the latency and throughput rather than reducing the buffer sizes [35]. Two scheduling policies are investigated in this section and alternatively applied to analyse the performance improvements.

7.4.1 Completely Fair Scheduler

The Completely Fair Scheduler is the default scheduler used in Linux. As investigated in 2.5.2 the CFS attempts to divide the processors equally among all tasks waiting to run. The scheduler defines a dynamic period where the length depends on the number of tasks waiting to run and divides the period across all runnable tasks. In order to give certain tasks more processor time during this period the priority, niceness value, can be changed.

The goal in changing this scheduling is to improve the work distribution and to "pull" frames towards the output. In order to do this steps later in the processing chain are given higher priorities than the earlier steps. Once a step completes processing a frame the next step can pre-empt the previous step and start processing a frame. However, the CFS is not completely pre-emptive; the priority can be used to ensure a thread is assigned a larger slice of the period and more likely to be selected to run first. But the scheduler still ensures fairness by giving tasks that rarely run, or run for very short periods, an opportunity to run first. As seen in Table 14, with this policy and priority scheme applied the framerate improves by around 4 % and the latency by less than 2 %. The CPU usage also increases by nearly 6 %.

<table>
<thead>
<tr>
<th>Completely Fair Scheduler results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Throughput</td>
</tr>
<tr>
<td>12.28 FPS (+4.0 %)</td>
</tr>
</tbody>
</table>

The results of the thread state analysis is given in Figure 65 and Figure 66.
The state occupation graphs now show that overall the threads spend more time in the ready pre-empt state, indicating that threads are now pre-empted from running by higher priority threads. The CalculateShiftAndMotion step appears to be the only step that is severely affected, spending about 15% in the ready pre-emption state.

The timeline shows that these threads are pre-empted by the CalculateSignalToNoise threads that succeed the step and therefore have a higher priority. The effect is that the majority of the work done by the CalculateShiftAndMotion threads are moved to the end of the CalculateSignalToNoise parallel region where only the main thread is active and three processors are idle. This area was previously idle because the CalculateShiftAndMotion threads completed their work during the CalculateSignalToNoise parallel region.
However, the improvement in latency and throughput is only minor. This is likely due to the fairness of the Completely Fair Scheduler that guarantees that all tasks will be allowed to run eventually and be given a slice of the period if they are waiting to run. In order to do this the scheduler maintains the virtual runtime for each thread. This virtual runtime increments slower for higher priority threads, allowing them more time on the processor and likely to be selected first to run. But threads that run less will increment their virtual runtime less often and may also end up with the lowest virtual runtime and be selected to run [26]. This explains why in the timeline the lower priority InputAgent, InputSplitter and ProcessAgent threads are allowed to run directly when they wake up. It also explains why the CalculateShiftAndMotion threads are allowed to do some work in the period where the CalculateSignalToNoise threads have work pending. This interference explains why no significant latency or throughput improvement is seen.

### 7.4.2 Round Robin Scheduler

Round Robin, a fixed priority pre-emptive scheduler, is an alternative scheduler that is intended for threads with more real time requirements. As investigated in 2.5.2, the scheduler gives tasks a fixed time slice for which they are allowed to run on a processor. At the end of this time slice other waiting tasks with the same priority are given a time slice to run on the processor. This repeats in a round-robin fashion, until no more tasks with the same priority are waiting to run and lower priority tasks are allowed to run. If a higher priority task becomes available it will immediately pre-empt a running task.

Similarly to the previous scheduler, the goal in changing this scheduling is to improve the work distribution and to "pull" frames towards the output. In order to do this, the steps later in the processing chain are given higher priorities than the earlier steps. Once a step completes processing a frame the next step can pre-empt the previous step and start processing a frame. The Round Robin scheduling is completely pre-emptive and higher priority threads will be allowed to run immediately if ready to run and are allowed to complete their work, given that the waiting queue for its priority and higher priority threads remain empty. Round Robin behaviour is only expected to occur with the data parallel threads of a step since these threads have the same priority. Table 15 shows applying this scheduling policy and priority scheme sees large improvements in the latency of more than 35 % and an improvement of nearly 18 % in the throughput. The CPU usage increases by more than 16 %. Enforcing the priority scheme also reduces the buffer sizes and the buffer build-up seen in the initial analysis, Chapter 4.2.2, no longer occurs.

Table 15: Round Robin Scheduler results

<table>
<thead>
<tr>
<th>Throughput</th>
<th>Latency</th>
<th>CPU usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>13.90 FPS (+17.7 %)</td>
<td>93.01ms (-35.4 %)</td>
<td>93.8 % (+16.3 %)</td>
</tr>
</tbody>
</table>

The thread state analysis with the Round Robin priority scheduling is given in Figure 67 and Figure 68.
Figure 67: State occupation percentages for Round Robin with priority scheduling

The state occupation graph now shows that the CalculateSignalToNoise threads and all the threads belonging to the later steps experience no pre-emptions or quantum exhaustion. These threads sleep and are allowed to run when they wake up without any involuntary interruptions. The only threads that experience interference are the CalculateShiftAndMotion and the InputAgent steps.

The CalculateShiftAndMotion threads spend time in the runnable state waiting to run. These threads briefly sleep, due to the yield that occurs when synchronizing with OpenMP to fetch the next chunk of data [17]. When yielding, the thread’s time slice is forfeited and another of this step’s thread is allowed to run. The threads also spend some time in the pre-emption state when the end steps start processing a frame. The InputAgent no longer spends any time in the sleep state, but spends about 40% in the ready pre-empt state and 55% in the I/O wait state. The massive increase in the I/O wait state is due to the lower priority of the thread, which does not allow it to be scheduled and move from the I/O wait state to the runnable state until there is an idle core. The pre-emption likely occurs when the InputAgent is executing to preparing a frame after I/O and is interrupted by a higher priority thread.

Figure 68: Timeline of Round Robin with priority scheduling
Analysing the timeline, *Figure 68*, confirms that the higher priority threads can complete their work without involuntary interrupts. Unlike the Completely Fair Scheduler investigated in the previous section where the `CalculateShiftAndMotion` threads were allowed to do some work, the threads are now completely disallowed from working in the `CalculateSignalToNoise` parallel region and can only start when the serial region begins. The timeline also shows that `InputAgent` spends a significant amount of time interrupted by the `CalculateSignalToNoise` threads, and only at the end of the parallel region can the `InputAgent` complete processing and deliver the next frame. As a result of this the buffer build up is no longer seen. If a camera is used as input, this may lead to frames being dropped.

Since the work is now strictly ordered and the threads experience less interference the latency is greatly reduced as frames are “pulled” towards the output. The idle areas are now better exploited, since lower priority threads are completely disallowed from work until higher priority threads release a core. The result is a significant increase in the CPU usage and a nearly equivalent increase in the throughput.
7.5 Final Results

The improvements in this section has shown how the analysis can be used to systematically improve the performance of an application using the Extended Thread State analysis. The observations made in the previous sections were investigated and the improvements were applied. As a result the latency of AutoROI running on the Reference Framework improved by more than 46% and the throughput improved by 45%. The CPU is better exploited and CPU usage increased by 30% to more than 90%.

<table>
<thead>
<tr>
<th></th>
<th>Initial ported version</th>
<th>Final optimized version</th>
<th>Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>Throughput</td>
<td>9.58 FPS</td>
<td>13.90 FPS</td>
<td>+45.1 %</td>
</tr>
<tr>
<td>Latency</td>
<td>174.30ms</td>
<td>93.01ms</td>
<td>-46.6 %</td>
</tr>
<tr>
<td>CPU usage</td>
<td>72.2 %</td>
<td>93.8 %</td>
<td>+29.9 %</td>
</tr>
</tbody>
</table>

In Figure 69 the output of the Extended Thread State Analysis is given. The diagrams on the left show the original analysis and the diagrams on the right the analysis of the final version. In the final version the improvements of this section are applied with Round Robin scheduling. As can be seen in the state occupation graph the overhead that threads experience has largely been reduced and the majority of the threads only sleep and execute. The timeline shows a much clearer picture, where the black areas of context switching is eliminated and the interference between threads is reduced.

The result of the improvements is that the application behaves more consistently and predictably and is less perceptible to the state of the operating system or interference from other programs. This can also be seen when viewing the core usage, Figure 70, where cores now perform more consistent bursts of work.
8 Analysis of Other Contactless Monitoring Algorithms

The Reference Framework supports a growing collection of respiration and pulse algorithms. In this section the Extended Thread State Analysis method presented in this thesis is applied to the other algorithms currently implemented on the Reference Framework. Some brief analysis is given as well as possible points of investigation that might lead to improvements.

8.1 Procor

Procor is a respiration algorithm that uses motion to monitor breathing of a patient [16]. The algorithm is executed on fixed regions of an image that is manually configured, whereas AutoROI, the algorithm investigated in previous sections, is based on Procor and applies Procor on blocks of a frame to automatically select regions where breathing occurs. For this analysis a single region, spanning the entire frame, is declared and the default scheduling used. The results are given in Table 17.

![Processing chain for Procor with algorithm steps in white.](image)

Table 17: Procor single region results

<table>
<thead>
<tr>
<th>Video info</th>
<th>768x576 60 FPS (single region)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frame rate</td>
<td>37.56 FPS</td>
</tr>
<tr>
<td>Latency</td>
<td>28.60ms</td>
</tr>
<tr>
<td>CPU usage</td>
<td>29.30 %</td>
</tr>
</tbody>
</table>

The algorithm itself consists of two steps. The state occupation graph and timeline is given in Figure 72 and Figure 73 respectively.

![State occupation percentages for Procor single region.](image)
Figure 73: Timeline of Procor single region.

Most of the work for Procor is performed in the ExtractMulti (single) step by a single thread, this can be seen in both the state occupation graph and timeline. The thread is allowed to execute without any interference, when other threads need to execute idle cores are used. This algorithm is relatively lightweight and achieves a good performance of nearly 38 frames per second and a latency of less than 30ms. No data parallelism is currently exploited in this algorithm with a single region. But the threads are still created and incur some minor synchronization overhead, a recommendation would be to completely remove these threads if a single region is used. Nonetheless, the performance is sufficient and the platform is powerful enough to execute the algorithm with the desired performance.
8.2 Procor Multiple Regions

In this analysis Procor is once again analysed, but four equally sized regions are used instead of one. Each region is assigned a quadrant of the video frame. The results are given in Table 18.

Table 18: Procor multiple regions results

<table>
<thead>
<tr>
<th>Video info</th>
<th>768x576 60 FPS (4 equal sized regions)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frame rate</td>
<td>57.50 FPS</td>
</tr>
<tr>
<td>Latency</td>
<td>19.78ms</td>
</tr>
<tr>
<td>CPU usage</td>
<td>48.90 %</td>
</tr>
</tbody>
</table>

The state occupation graph and the timeline is given in Figure 74 and Figure 75.

The Procor algorithm now appears to become I/O bound when using multiple regions. This can be seen in the state occupation graph where the InputAgent is in the I/O wait state for nearly 80% of the time and never sleeps. The use of all 4 data parallel threads, a thread per region, allows greater utilization of the CPU, at nearly 50%, and reduces the computation time per frame resulting in the algorithm becoming I/O bound. The
video is recorded at 60 frames per second at a resolution of 768x576 using 8 bits monochrome value. The bandwidth that is used can be approximated as follows.

\[
usedBandwidth = \text{resolution}_X \times \text{resolution}_Y \times \frac{\text{bits per pixel}}{8} \times \text{frameRate}
\]

\[
usedBandwidth = 768 \times 576 \times \frac{8}{8} \times 57.5 = 24.3 \text{ MB/s}
\]

The benchmark previously determined that the read speed of the SD card was 20.4 MB/s. However a higher read speed of 24.3 MB/s is achieved with the 57.5 frames per second output above. This is most likely due to the sequential reading and prefetching of the file which allows faster access. A recommendation would be to use the faster USB for storing and accessing the video file, rather than the SD card. However a framerate of 60 frames per second is rarely used in practice.
8.3 **ObjectPulse**

ObjectPulse is a contactless vital sign monitoring algorithm that uses remote photoplethysmography (rPPG) to monitor the pulse of a person. The technique determines a pulse signal using a camera to capture the subtle colour variations caused by the blood flow in light reflected from a person’s skin. ObjectPulse uses object tracking to detect and follow skin on which to apply rPPG and implements an algorithm known as Plane-Orthogonal-to-Skin (POS) as described in [18].

The processing chain for the algorithm is given in **Figure 76**. The algorithm is run with the default scheduler. As can be seen in **Table 19** the algorithm performs relatively well with a throughput of more than 13 FPS using more than half of the CPU power.

**Table 19**: ObjectPulse results

<table>
<thead>
<tr>
<th>Video info</th>
<th>768x576 30 FPS (24 bit RGB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frame rate</td>
<td>13.26 FPS</td>
</tr>
<tr>
<td>Latency</td>
<td>164.69 ms</td>
</tr>
<tr>
<td>CPU usage</td>
<td>53.55 %</td>
</tr>
</tbody>
</table>

The state occupation graph and the timeline is given in **Figure 77** and **Figure 78**.

![State Occupation Graph](image)

**Figure 77**: State occupation graph of ObjectPulse.
Figure 78: Timeline of ObjectPulse.

The analysis shows that the majority of the work performed by the ObjectPulse algorithm is due to the skin detection and tracking rather than the pulse signal extraction and rate calculations. Most of the work is done in the SkinMask and OnlineTracker threads. The InputAgent now consists of several threads, these threads are spawned by the library, and are likely the result of processing a colour video, rather than a grayscale video.

The analysis shows that the SkinMask spends a significant amount of time, more than 20%, resolving cache stalls. The majority of this overhead spent is attributed to cache misses in the instruction cache. As a recommendation it could be investigated if the SkinMask thread can be functionally decomposed to use several threads. This will reduce instruction complexity allowing better Level 1 instruction cache memory usage and attempt to divide computation across all cores increasing CPU utilization. The actual frame colour averaging used by the algorithm is done in the ObjectPulseAverageRGB step and pulse calculation in the ObjectPulseExtractPulse and ObjectPulseImprovement steps.
8.4 PulseLCB

PulseLCB is another pulse algorithm that uses rPPG to monitor pulse. The algorithm uses biological properties of the blood volume pulse signal in different wavelengths to detect the pulse signal in a camera image as described in [18]. Instead of object tracking PulseLCB uses a face detection library to identify face patches with skin on which to apply the rPPG algorithm.

However, since the face detector library is proprietary and no source code was provided, the face detector is not included in the current algorithm on the Linux port. In the implementation and analysis below the library has been stubbed to “detect” a face in a square in the centre of the image filling one 9th of the frame.

The processing chain for PulseLCB is given in Figure 79. The algorithm performs poorly and a low frame rate and high latency is achieved, as seen in Table 20.

![Processing chain for PulseLCB](image)

**Figure 79:** Processing chain for PulseLCB white algorithm steps in white.

*Table 20: PulseLCB results*

<table>
<thead>
<tr>
<th>Video info</th>
<th>768x576 30 FPS (24 bit RGB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frame rate</td>
<td>2.75 FPS</td>
</tr>
<tr>
<td>Latency</td>
<td>370.05ms</td>
</tr>
<tr>
<td>CPU usage</td>
<td>26.69 %</td>
</tr>
</tbody>
</table>

The state occupation graph and the timeline is given in Figure 80 and Figure 81.

![State occupation graph](image)

**Figure 80:** State occupation graph of PulseLCB.
Figure 81: Timeline of PulseLCB

The analysis shows that PulseLCB is completely CPU bound. The majority of the work is done in the PulseSignalExtraction thread which calculates the average colour of the face patches. The thread spends barely any time waiting to run and experiences very little context switching and cache stall overheads.

Analysing the PulseSignalExtraction step reveals that it has several nested loops that are data independent and can therefore benefit from data parallelism to increase CPU usage. The step can also be functionally decomposed into two separate parallel steps. This will allow multiple threads to use the CPU. The actual pulse signal is calculated in the PulseSignalAnalysis and PulseImprovement steps which only execute for 0.1% of the time.

The analysis excludes the face tracker library. The addition of the library will likely change the results of the analysis.

The analysis of the other algorithms on the Reference Framework shows how the Extended Thread State Analysis and its visualization can be used to improve the understanding of an application and identify bottlenecks that can be used to propose improvements of the algorithms.
9 Conclusion and Future Work

This thesis has presented the work done in porting and improving the performance of a Reference Framework developed to run contactless vital sign monitoring algorithms on an embedded Linux system. Contactless vital signs monitoring is an active research area and a promising technology that will open up many new use cases for vital sign monitoring in healthcare. The initial performance requirements in latency and throughput of the AutoROI algorithm were met. In order to improve the performance an analysis method has been developed and applied to analyse the performance and guide the optimizations. The optimized port showed that the Reference Framework and the contactless vital sign monitoring algorithms can be run on constrained systems within the specified requirements.

Answering the research questions provided insight on the Reference Framework and how a Proof of Concept could be developed on the embedded platform. The questions further guided the profiling and analysis work that was used to identify improvements.

9.1 Contributions

The main contribution of this thesis work is the development of the Extended Thread State Analysis method used to guide the optimizations of the Proof of Concept that demonstrates the viability of the Reference Framework on a constrained system. The optimizations focused on the AutoROI respiration monitoring algorithm as a use case. Both the goals of the thesis were fulfilled and the contributions for each goal are shortly described below.

As a Proof of Concept the Reference Framework that implements the contactless vital sign algorithms has been ported to Linux using a single code base. This code base can be used to compile the framework for both Windows and Linux and allows additional algorithms to be supported for both operating systems with minimal effort. The Proof of Concept is demonstrated on a Raspberry Pi 3, as well as a virtual machine on the workstation laptop.

Since initial analysis showed underperformance of the application and underutilization of platform resources a more detailed analysis, the Extended Thread State Analysis method, has been developed based on an existing analysis method. This analysis is further expanded to answer two questions that are more of a concern in constrained systems: mainly why is a thread waiting for an available core and how well is the time spent by a thread executing on a core. This Extended Thread State Analysis is suitable for any Linux program and can largely be applied without any code modifications or recompilation. It allows post-run interactive visualization and quantification of thread states. To demonstrate the validity and usefulness of the analysis, several of the contactless vital sign monitoring algorithms were analysed using the method where multiple issues, bottlenecks and optimizations have been identified. Focusing on AutoROI it has been shown how the analysis can guide optimizations, identify problematic threads and be used to gain insight as well as verify that optimizations are correctly applied. These optimizations have resulted in a performance improvement of more than 45% and increased CPU utilization to more than 90%. Additionally, the method was further validated by applying it to the other contactless vital sign monitoring algorithms on the Reference Framework.
9.2 Future Work

In this section several recommendations for future work are given.

- **Reference Framework Improvements.** The ported version of the Reference Framework, described in Chapter 3, has focused on AutoROI. Functionality that is irrelevant for AutoROI has not been considered in this thesis. Therefore, some functionality of the Reference Framework is currently not implemented for the Linux version. An example is the missing face tracker library used by several algorithms. In addition the recording application has not been validated.

- **Reference Framework Maintenance.** The Reference Framework is currently maintained on a separate branch on the version management system. Changes to the mainline development will need to be periodically merged to keep the branch up to date. This merging process is simplified by the single code base and existing abstractions.

- **AutoROI Improvements.** Chapter 7 investigated and applied numerous fixes and optimizations to the AutoROI algorithm to improve the performance. However, several anomalies discovered during the analysis could potentially still be investigated; in particular the excessive page faults for the *CalculateBlockWeight* thread and the context switching by the *ResultsConverter*, can still be investigated, even though these points will likely only result in minor improvements.

- **Other Algorithms.** In Chapter 8 the Extended Thread State Analysis was applied and briefly analysed for the other algorithms currently implemented on the Reference Framework. This analysis identified several points worthy of further investigation and optimization for the algorithms.

- **Automatic Priority Assignment.** The Extended Thread State Analysis quantifies the amount of time threads spent in a particular state. In future work, it could be investigated how this could be used to mathematically map threads to I/O, as Linux also allows priorities to be assigned to disk access, or CPU priority assignments. This is particularly interesting for applications that are not Directed Acyclic Graphs where simple priority schemes to “pull” data to the output may not be effective.
References


Appendix A: Configuring Linux for Development

**VSCREF development on Linux**

This guide describes in detail how to configure a Linux machine to develop and execute the vital signs camera reference framework (VSCREF). The platform independent code base and project generation files are in git version control under origin/students/michael.

Commands or deviations that are specific for a Raspberry Pi are indicated using [Raspberry Pi].

For support or questions: michaelgkruger@yahoo.com.

This guide, and VSCREF, has been tested on the following platforms:

1. Raspberry Pi 3 (ARM, 32-bit)
2. Debian 9 Virtual Machine (x86, 64-bit)

The guide consists of the following sub-sections:

1. Linux configuration and setup
2. Building and executing the Core and UnitTests projects
3. Useful Linux commands
4. Limitations and pending work

1. **Linux configuration and setup.**

1.1 Install a Linux image

The first step is installing the desired Linux image. A wide selection of distributions are available, it makes sense to download a widely supported and used distribution unless a specific distribution is required. The initial development and porting has been done using Debian builds (Ubuntu has also been used).

[Raspberry Pi]

The rest of this section describes how to install the distribution used on the Raspberry Pi 3. A micro SD card of at least 8GB is required.


Unzip the image. To burn the image to the SD card, Etcher can be used; available at [https://etcher.io/](https://etcher.io/). To use Etcher simply select the image and SD card and press Flash. Once the process is done the drive should be remounted.

There are two ways to access the Raspberry Pi. If it is possible to determine the IP address of the Pi SSH can be used. In recent releases due to security concerns, SSH is disabled by default. To enable SSH it is required to create a file on the /boot directory called ssh. This will enable SSH the first time the Pi boots. Connect an Ethernet cable and insert the SD card, boot the Pi.
Use a SSH client (such as TeraTerm, available from https://ttssh2.osdn.jp/index.html.en) to connect to the Pi. After connecting the Pi will ask for the credentials.

If it is not possible to determine the IP address, and a HDMI screen and USB keyboard is available, then this can be used to log-on to the Pi. After inserting the SD card and booting the log-on will be shown.

Log-on to the Pi using the default credentials:

**Username:** pi  
**Password:** raspberry

To setup the Pi for future access run the following command:

```
sudo raspi-config
```

It will present the following:

```
Appendix A-1 Raspi-Config Screen

Using the interface change the password (option 1) to something more secure, since the Pi can now be accessed on the network. In the interfacing options (option 5) enable the camera (P1), SSH (P2) and VNC (P3)
```

```
Appendix A-2 Enabling Camera, SSH and VNC

The Pi is now ready to be configured for development with VSCREF.
```
1.2 Configuring prerequisites

In order to develop for the VSCREF project several libraries and dependencies need to be installed and in some cases built from source (some of the prebuilt libraries are outdated or minimal on the Pi).

Several libraries need to be built from source this will require package such as git and cmake, as well as Linux build tools to be installed. Run the commands below:

<table>
<thead>
<tr>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>apt-get update</td>
</tr>
<tr>
<td>apt-get install sudo</td>
</tr>
<tr>
<td>apt-get install git</td>
</tr>
<tr>
<td>apt-get install cmake</td>
</tr>
<tr>
<td>apt-get install build-essential</td>
</tr>
<tr>
<td>apt-get install uuid-dev</td>
</tr>
</tbody>
</table>

Install GCC

Since VSCREF is written in C++ it will require a C++ compiler and linker. The GNU Compiler Collection can be installed to compile VSCREF. The framework uses features from C++11 that are available in GCC 5 and later. Install the GCC version from the package sources and determine whether the version is new enough:

<table>
<thead>
<tr>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>apt-get install gcc</td>
</tr>
<tr>
<td>gcc --version</td>
</tr>
</tbody>
</table>

If the version is lower than 5 it is required to build the latest version of GCC from source. Otherwise it is possible to skip the rest of this step. This will take several hours to complete. 

[Raspberry Pi] Before starting, the swapfile size may need to be increased due to the RAM limitations. Change the CONF_SWAPSIZE to 1024 in /etc/dphsys-swapfile and reboot.

Run the following sequence of commands to build GCC:

<table>
<thead>
<tr>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>cd</td>
</tr>
<tr>
<td>wget <a href="http://gcc.parentingamerica.com/releases/gcc-6.3.0/gcc-6.3.0.tar.bz2">http://gcc.parentingamerica.com/releases/gcc-6.3.0/gcc-6.3.0.tar.bz2</a></td>
</tr>
<tr>
<td>tar xf gcc-6.3.0.tar.bz2 -v</td>
</tr>
<tr>
<td>cd gcc-6.3.0</td>
</tr>
<tr>
<td>apt-get install libmpc-dev</td>
</tr>
<tr>
<td>mkdir download_prerequisites</td>
</tr>
<tr>
<td>cd obj</td>
</tr>
<tr>
<td>../configure --enable-languages=c,c++ --prefix=/usr/local/gcc-6.3 --program-suffix=-6 --with-cpu=cortex-a53 --with-fpu=neon-fp-armv8 --with-float=hard --build=arm-linux-gnueabihf --host=arm-linux-gnueabihf --target=arm-linux-gnueabihf</td>
</tr>
</tbody>
</table>
make -j5
sudo make install

```
sudo update-alternatives --install /usr/bin/gcc gcc /usr/local/gcc-6.3/bin/gcc-6 60 --slave /usr/bin/g++ g++ /usr/local/gcc-6.3/bin/g++-6
```

```
sudo update-alternatives --install /usr/bin/gcc gcc /usr/bin/gcc-4.9 40 --slave /usr/bin/g++ g++ /usr/bin/g++-4.9
```

```
sudo update-alternatives --config gcc
```

To make sure that the correct standard runtime libraries are dynamically loaded, it is necessary to export the new library path. This can be done by adding “export LD_LIBRARY_PATH=/usr/local/gcc-6.3/lib:$LD_LIBRARY_PATH” to ~/.bashrc. Additionally create a file called /etc/ld.so.conf.d/gcc-6.3.conf and add “/usr/local/gcc-6.3/lib” to it.

GCC should now be built and loaded correctly. [Raspberry Pi] After GCC is compiled remember to restore the CONF_SWAPSIZE in /etc/dphsys-swapfile and reboot.

Install OpenCV

VSCREF requires that OpenCV version 3.3 is available. Currently some package managers only have libraries for version 2. First determine what version is available in the package manager:

```
sudo apt-cache policy libopencv-dev python-opencv
```

If the version is 3.3 or greater simply install OpenCV using apt-get install and skip the rest of this section. If the version that is to be installed is not version 3.3 or greater it is required to download and build OpenCV. Luckily a shell script can be used to simplify this process.

```
```

```
install-opencv.sh
```

The script will run and install OpenCV. This may also take several hours to complete. During the development issues were experienced with installing QT. Follow the instructions in the GUI subsection in the shell script to remove QT support and use GTK instead.

Install MatIO

Some of the tests use the MatIO library. This library may not be available in the package managers. To install it run the commands below:

```
apt-get install libmatio-dev
```

```
wget https://github.com/tbeu/matio/releases/download/v1.5.8/matio-1.5.8.tar.gz
```
Install RaspiCam C++ library

The RaspiCam C++ library is required to capture images from the Raspberry Pi camera to be used in VSCREF. The library can be built by downloading the source from [http://www.uco.es/investiga/grupos/ava/node/40](http://www.uco.es/investiga/grupos/ava/node/40) and running the commands below:

```bash
tar -xvf matio-1.5.8.tar.gz
./autogen.sh
./configure
make
make check
make install
```

To test the camera and its OpenCV binding run the commands below:
```
./raspicam_test
./raspicam_cv_test
```

Install uEye camera drivers

The uEye is an industrial camera that is used for image capture in VSCREF. Libraries are provided for ARM. First determine whether the CPU supports hardware or software float. This can be done by running the following command, and checking if libraries are in the form of `arm-linux-gnueabihf` where the `hf` indicates hardware float support.

```
ldd /bin/ls
```

Additionally confirm that the correct versions of GNU C (>= 2.7) and GNU C++ (>= 3.4.17) are available.

```
strings /lib/arm-linux-gnueabihf/libc.so.6 | grep GLIBC
strings /usr/lib/arm-linux-gnueabihf/libstdc++.so.6 | grep GLIBCXX
```

Download the correct version of the library at [https://en.ids-imaging.com/download-ueye-emb-hardfloat.html](https://en.ids-imaging.com/download-ueye-emb-hardfloat.html) and setup the library.
```
tar -xvf uEye_SDK_driver -C /
rm uEye_SDK_driver
sudo sh /usr/local/share/ueye/bin/ueyesdk-setup.sh
```

Once this is done add the uEye USB daemon to be started at boot. Reboot to apply the changes.
```
update-rc.d ueyeusbdrc defaults
```
The Raspbian Stretch Lite distribution does not come with a Graphical User Interface installed. In order to include a minimal desktop that can be enabled and disabled when required it is possible to install Pixel using the commands below:

```
sudo apt-get install --no-install-recommends xserver-xorg xserver-xorg-video-fbdev xinit
```

```
sudo apt-get install raspberrypi-ui-mods
sudo apt-get install lightdm
```

After the GUI is installed it needs to be started. This will cause output to be rendered via HDMI and VNC. It also makes X-11 tunnelling possible. To make this permanent the boot options in the raspi-config can be changed to boot to Desktop. The GUI does however introduce a memory and CPU load.

```
Startx
```

2. **Building and executing the Core and UnitTest projects**

   2.1 Generating build files and compilation

   First check out the source code to vscref using git or use an FTP file client to transfer the folder to the Linux machine (this is useful to develop and test simultaneously on both Windows and Linux)

   To build the project first the project files need to be generated. This is done using CMake. To keep the project clean create a new folder in the main directory of the project:

   ```
   mkdir build
   cd build
   ```

   Now using CMake the project files can be generated:

   ```
   cmake .
   cmake-gui
   ```
CMake GUI can be used to easily configure the project settings.

Choose the source as the main folder and the build directory as the build directory created.

To build the complete project select all the algorithms. To enable support for the Raspberry Pi camera that was added, define the WITH_RASPI_CAMERA option.

Click Configure and Generate to generate the project files.

The projects are now ready to be compiled. Run the following commands in the build folder:

```
make clean
make -j4
```

After several minutes the binaries will be created in the sub directories Applications/Core and Applications/UnitTests

2.2 Executing the projects

The executables are now ready to be run. For the Core project move to the directory of the executable and run the command with the arguments. These arguments are identical to the Windows arguments, with the exception that forward slashes are used for paths. Make sure that the ini files are available.

As an example, to run AutoRoi with the Raspberry Pi camera:

```
cd Applications/Core
./Core -v raspicam --videoformat mono -lo 1 -lc 1 -vs 1 -o autoroi -i core_exe.ini
```
To run the UnitTests the test data needs to be copied to the executable folder. This can be done by copying the files in the Applications/UnitTests/TestData folder. The UnitTests can be run as follows:

``` bash
cd Applications/UnitTests
./UnitTests
```

![UnitTests output](image)

**Appendix A**

- **5 Output of UnitTests application**

3. **Useful Linux commands**

The following table contains a list of commands and utilities that were useful during the development:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>watch -n 1 /opt/vc/bin/vcgencmd measure_temp</code></td>
<td>[Raspberry Pi] Prints the temperature of the processor every second.</td>
</tr>
<tr>
<td><code>watch -n 1 /opt/vc/bin/vcgencmd measure_clock arm</code></td>
<td>[Raspberry Pi] Prints the dynamic clock frequency of the processor every second.</td>
</tr>
<tr>
<td><code>apt-get install hwloc lstopo lscpu</code></td>
<td>Can be used to see how the operating system perceives the processor and memory topology.</td>
</tr>
<tr>
<td><code>getconf LONG_BIT</code></td>
<td>Returns whether the operating system is 32 or 64 bit.</td>
</tr>
<tr>
<td><code>uname -r</code></td>
<td>Returns the Kernel version.</td>
</tr>
<tr>
<td><code>top -H</code></td>
<td>Prints the CPU and memory usage of all threads.</td>
</tr>
<tr>
<td><code>iostat -x &lt;int&gt;</code></td>
<td>Prints the I/O usage of the I/O devices at interval &lt;int&gt; seconds.</td>
</tr>
<tr>
<td><code>sudo apt-get install iotop iotop</code></td>
<td>Prints the I/O usage per process.</td>
</tr>
<tr>
<td><code>ls -l /dev/disk/by-uuid/</code></td>
<td>[Raspberry Pi] Used to mount a USB drive.</td>
</tr>
<tr>
<td><code>sudo mount /dev/sda1 /media/usb -o uid=pi,gid=pi</code></td>
<td></td>
</tr>
<tr>
<td><code>sync; echo 3 &gt; /proc/sys/vm/drop_caches</code></td>
<td>Can be used to clear all memory caches.</td>
</tr>
</tbody>
</table>
https://sourceforge.net/projects/lmbench/
make results see

Runs the *lmbenchmark* to measure several metrics.

**perf stat** `<command>`

Prints information on the execution of `<command>` such as instructions, context switches, IPC, page faults and cache-misses.

**taskset** `<mask>` `<command>`

Set the processor affinity of `<command>` where `<mask>` is a hexadecimal representation of the cores.

**renice** `-n` `<nice>` `-p` `<processId>`

Used to change the niceness of `<process>` to `<nice>`.

**chrt** `-m`

Lists the supported scheduling policies and their priority ranges.

**chrt** `-r` `-p` `<priority>` `<process>`

Used to set the `<process>` to Round Robin scheduling policy with priority `<priority>`.

**chrt** `-f` `-p` `<priority>` `<threadid>`

Used to set the `<process>` to FIFO scheduling policy with priority `<priority>`.

4. **Limitations and pending work**

The following limitations and work is still pending:

1. Record application. The record application that is used to record reference and video files has not been explicitly ported to Linux.
2. Face tracker library. The face tracker library is currently stubbed due to the fact that the source code has not been made available. As a result the algorithms that rely on it will not perform as intended and their unit tests will fail.
3. Dependency libraries are only compiled for x86 64-bit and ARM architectures. This includes the gtest and gmock libraries used for unit testing as well as the CPFSPD library used for video files. The source code is available and these libraries can easily be recompiled.
## Appendix B: Raspberry Pi 3 Hardware Benchmark Results

<table>
<thead>
<tr>
<th>Host</th>
<th>OS Description</th>
<th>MHz</th>
<th>lat cache mem scal pages</th>
<th>load</th>
</tr>
</thead>
<tbody>
<tr>
<td>raspberry Linux 4.9.36-</td>
<td>armv71-linux-gnu 1198</td>
<td>10</td>
<td>64 2.7600 1</td>
<td></td>
</tr>
</tbody>
</table>

**Processor, Processes - times in microseconds - smaller is better**

| Host               | OS | MHz | null null | open alt sig sig fork exec sh call I/O stat close TCP inst hndl proc proc proc |
|--------------------|----|-----|-----------|-----------------------------|---------------------------------|
| raspberry Linux 4.9.36- |    | 1198| 0.25 0.56 3.33 11.3 21.3 0.79 3.67 591.1984 4889 |                               |                                 |

**Basic integer operations - times in nanoseconds - smaller is better**

<table>
<thead>
<tr>
<th>Host</th>
<th>OS</th>
<th>integr integr integr integr integr integr integr</th>
<th>bit add mul div mod</th>
</tr>
</thead>
<tbody>
<tr>
<td>raspberry Linux 4.9.36-</td>
<td></td>
<td>0.8300 0.8300 5.0100 5.0100</td>
<td></td>
</tr>
</tbody>
</table>

**Basic uint64 operations - times in nanoseconds - smaller is better**

<table>
<thead>
<tr>
<th>Host</th>
<th>OS</th>
<th>uint64 uint64 uint64 uint64 uint64</th>
<th>bit add mul div mod</th>
</tr>
</thead>
<tbody>
<tr>
<td>raspberry Linux 4.9.36-</td>
<td></td>
<td>0.840 66.5 62.6</td>
<td></td>
</tr>
</tbody>
</table>

**Basic float operations - times in nanoseconds - smaller is better**

<table>
<thead>
<tr>
<th>Host</th>
<th>OS</th>
<th>float float float float</th>
<th>add mul div bogo</th>
</tr>
</thead>
<tbody>
<tr>
<td>raspberry Linux 4.9.36-</td>
<td></td>
<td>5.3400 5.3500 10.9 20.2</td>
<td></td>
</tr>
</tbody>
</table>

**Basic double operations - times in nanoseconds - smaller is better**

<table>
<thead>
<tr>
<th>Host</th>
<th>OS</th>
<th>double double double double</th>
<th>add mul div bogo</th>
</tr>
</thead>
<tbody>
<tr>
<td>raspberry Linux 4.9.36-</td>
<td></td>
<td>3.3400 3.3400 10.4 27.8</td>
<td></td>
</tr>
</tbody>
</table>

**Context switching - times in microseconds - smaller is better**

| Host               | OS | 2p/OK 2p/16K 2p/64K 8p/16K 8p/64K 16p/16K 16p/64K oxksw oxksw oxksw oxksw oxksw oxksw oxksw oxksw |
|--------------------|----|-----------------------------|------------------|
| raspberry Linux 4.9.36- |    | 4.6900 4.8900 5.2700 5.6500 6.6100 6.41000 12.2 |                   |                                 |
### Local Communication latencies in microseconds - smaller is better

<table>
<thead>
<tr>
<th>Host</th>
<th>OS</th>
<th>2p/0K</th>
<th>Pipe AF</th>
<th>UDP</th>
<th>RPC/ TCP</th>
<th>TCP</th>
<th>RPC/ TCP</th>
<th>conn</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>raspberry Linux 4.9.36- 4.690</td>
<td>17.9</td>
<td>15.7</td>
<td>37.3</td>
<td>48.3</td>
<td>78.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Remote Communication latencies in microseconds - smaller is better

<table>
<thead>
<tr>
<th>Host</th>
<th>OS</th>
<th>UDP</th>
<th>RPC/ TCP</th>
<th>TCP</th>
<th>RPC/ TCP</th>
<th>conn</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>raspberry Linux 4.9.36-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### File & VM system latencies in microseconds - smaller is better

<table>
<thead>
<tr>
<th>Host</th>
<th>OS</th>
<th>OK File</th>
<th>10K File</th>
<th>Mmap</th>
<th>Prot</th>
<th>Page</th>
<th>100Kd</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Create</td>
<td>Delete</td>
<td>Create</td>
<td>Delete</td>
<td>Latency</td>
<td>Fault</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>raspberry Linux 4.9.36-</td>
<td>111.6</td>
<td>54.8</td>
<td>108.1</td>
<td>75.3</td>
<td>16.6K</td>
<td>1.43120</td>
<td>7.546</td>
</tr>
</tbody>
</table>

### Local Communication bandwidths in MB/s - bigger is better

<table>
<thead>
<tr>
<th>Host</th>
<th>OS</th>
<th>Pipe AF</th>
<th>TCP</th>
<th>File</th>
<th>Mmap</th>
<th>Bcopy</th>
<th>Bcopy</th>
<th>Mem</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>raspberry Linux 4.9.36-</td>
<td>608.</td>
<td>1853</td>
<td>767.</td>
<td>841.9</td>
<td>1768.7</td>
<td>968.7</td>
<td>992.5</td>
<td>2070</td>
<td>1728.9</td>
</tr>
</tbody>
</table>

### Memory latencies in nanoseconds - smaller is better

(WARNING: may not be correct, check graphs)

<table>
<thead>
<tr>
<th>Host</th>
<th>OS</th>
<th>Mhz</th>
<th>L1 $</th>
<th>L2 $</th>
<th>Main mem</th>
<th>Rand mem</th>
<th>Guesses</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>raspberry Linux 4.9.36-</td>
<td>1198</td>
<td>2.5290</td>
<td>6.5440</td>
<td></td>
<td>38.4</td>
<td>241.6</td>
<td></td>
</tr>
</tbody>
</table>

92
Appendix C: Developed Performance Analysis Tools

This appendix describes the tools that were developed during the course of this work and a guide on how to use them. The tools are divided into two categories, Reference Framework specific and generic tools. The Reference Framework specific tools are developed for use with the instrumentation added to the framework during porting efforts, it can be used for both Windows and Linux. The generic tools, including the Extended Thread State Analysis tools are developed for Linux and can be used for any program on Linux without requiring modification or recompilation.

Reference Framework Tools

StepStats.py

**Purpose:**
This Python script takes as input the CSV file that is generated by the instrumentation code, Chapter 3.1.2, and calculates several statistics such as the throughput, latency and frame time of the frames. It also calculates the average execution time and buffer occupancy per step. The script supports all the algorithms currently on the Reference Framework and can be used for profiling on both Linux and Windows.

**Usage:**
StepStats.py [-h] [-i INPUT] [-s START] [-e END] algorithm

<table>
<thead>
<tr>
<th>algorithm</th>
<th>The name of the algorithm to calculate statistics on (procor, autoroi, objectpulse, pulselcb, pulsepos, pulsegeneric)</th>
</tr>
</thead>
<tbody>
<tr>
<td>-h, --help</td>
<td>show this help message and exit</td>
</tr>
<tr>
<td>-i, --input</td>
<td>The input file with the CSV output generated by instrumentation in VSCREF (default = processUserTraceLog.txt)</td>
</tr>
<tr>
<td>-s, --start</td>
<td>The frame number to start with (default = 1)</td>
</tr>
<tr>
<td>-e, --end</td>
<td>The frame number to end with (default = 500)</td>
</tr>
</tbody>
</table>

**Output:**
A screenshot of the output is given below showing the FPS, latency and times per step.

![Output of StepStats.py](image)
Used to collect the statistics and create *Figure 25, Figure 26, Figure 31* and *Figure 32*

**StepVisualize.py**

**Purpose:**
This Python script takes as input the CSV file that is generated by the instrumentation code and generates a Grasp file that can be used to analyse the scheduling and execution of the frames over time. The script supports all the algorithms currently on the Reference Framework and can be used for profiling on both Linux and Windows.

**Usage:**
StepsVisualize.py [-h] [-i INPUT] [-s START] [-e END] outputfile algorithm

<table>
<thead>
<tr>
<th>parameter</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>outputfile</td>
<td>The name of the Grasp file to generate.</td>
</tr>
<tr>
<td>algorithm</td>
<td>The name of the algorithm to calculate statistics on (procor, autoroi, objectpulse, pulselcb, pulsepos, pulsegeneric)</td>
</tr>
<tr>
<td>-h, --help</td>
<td>show this help message and exit</td>
</tr>
<tr>
<td>-i, --input</td>
<td>The input file with the CSV output generated by instrumentation in VSCREF (default = processUserTraceLog.txt)</td>
</tr>
<tr>
<td>-s, --start</td>
<td>The frame number to start with (default = 1)</td>
</tr>
<tr>
<td>-e, --end</td>
<td>The frame number to end with (default = 500)</td>
</tr>
</tbody>
</table>

**Output:**
A screenshot of the interactive timeline plotted using Grasp is given below.

![Timeline](image)

*Appendix C - 2* Output of StepVisualize.py plotted using Grasp.

The output is seen in *Figure 34.*
Generic Tools

The generic tools consist of tools to analyse the performance as well as tools to apply the Extended Thread State Analysis. These tools can be used on Linux for any program, even without recompilation.

Tools to apply the Extended Thread State Analysis:

ProfileThreadStates.sh

Purpose:
This bash script starts and runs a program with Perf event and counters recording. The output is parsed and two files are generated. The first is a script file with all the scheduling decisions during the execution. The second is a stat file with the counters of hardware and kernel events for each thread. These files are used as input for the Python scripts in the thread state analysis.

Usage:
./ProfileThreadStates output-directory command-with-arguments

<table>
<thead>
<tr>
<th>output-directory</th>
<th>The directory to generate the output files.</th>
</tr>
</thead>
<tbody>
<tr>
<td>command-with-arguments</td>
<td>The command as it would usually be run with arguments if needed</td>
</tr>
</tbody>
</table>

Ensure that the script is run with root permissions and that the output file is saved on external USB media to reduce I/O interference when Perf stores its event buffers.

Output:
The output generates two files:

1. perf_script.txt: Contains all the operating system scheduler decisions for the application during execution
2. perf_stat.txt: Contains the counters for hardware and kernel events per thread.
ThreadStateMetrics.py

Purpose:
This Python script takes as input the perf script and stat files that are generated using the profiling bash script. It calculates the state occupation times and percentages and outputs a CSV file to allow bar graphs to be created.

Usage:
ThreadStateMetrics.py [-h] [-i1 SCRIPTINPUT] [-i2 STATINPUT] [-i3 BENCHINPUT]

| outputfile | The name of the CSV file to generate |
| process    | The name of the process and its threads to generate metrics from |
| -h, --help | show this help message and exit |
| -i1, --scriptinput | The input file with the Perf schedule events (default = perf_script.txt) |
| -i2, --statinput | The input file with the Perf statistics (default = perf_stat.txt) |
| -i3, --benchinput | The input file with the benchmark results (default = metric_benchmark.csv) |

Output:
An example screenshot of the thread state occupation times for a thread is given below.

Appendix C - 3: Output of ThreadStateMetrics.py

A CSV file is generated as output. The output is used to create the bar plots in chapters 6, 7 and Error! Reference source not found.
ThreadStateVisualize.py

**Purpose:**
This Python script takes as input the Perf output that is generated by the bash script, ProfileThreadStates.sh; and the processPerfSynchLog.txt file that is generated by the Reference Framework to synchronize user events. It generates a Grasp output file for visualization.

**Usage:**
ThreadStateVisualize.py [-h] [-i1 SCRIPTINPUT] [-i2 SYNCHINPUT] [-s START] [-e END] [-m MAXEVENTS] [-l SYNCOFFSET]

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>outputfile</td>
<td>The name of the output Grasp file to generate.</td>
</tr>
<tr>
<td>process</td>
<td>The name of the process and its threads to plot.</td>
</tr>
<tr>
<td>-h --help</td>
<td>show this help message and exit</td>
</tr>
<tr>
<td>-i1, --scriptinput</td>
<td>The input file with the Perf schedule events (default = perf_script.txt)</td>
</tr>
<tr>
<td>-i2, --synchinput</td>
<td>The input file with the user synchronization events (default = processPerfSynchLog.txt)</td>
</tr>
<tr>
<td>-s, --start</td>
<td>The relative time to start from in seconds (default = -1, beginning)</td>
</tr>
<tr>
<td>-e, --end</td>
<td>The relative time to end with in seconds (default = -1, end)</td>
</tr>
<tr>
<td>-m, --maxevents</td>
<td>The maximum number of switch events to process (default = 10000)</td>
</tr>
<tr>
<td>-l, --synchoffset</td>
<td>The offset in seconds between Perf timestamps and user timestamps (default = 0.013)</td>
</tr>
<tr>
<td>-u, --usesynch</td>
<td>Use the synchronization points (default = 1, true)</td>
</tr>
</tbody>
</table>

**Output:**
A screenshot of the interactive timeline plotted using Grasp is given below.
The output generates the timelines visualized with Grasp in chapters 6, 7 and Error!

Reference source not found.

The following tools are used for additional analysis.

CoreVisualize.py

**Purpose:**
This Python script takes a Perf script file containing the scheduling decisions and generates a Grasp file that can be used to plot the CPU usage of the different cores during the execution of a program. It also calculates the average CPU usage.

**Usage:**
CoreVisualize.py [-h] [-i INPUT] [-s START] [-e END] [-m MAXEVENTS] [-c NUMCPUS]

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>outputfile</td>
<td>The name of the Grasp file to generate.</td>
</tr>
<tr>
<td>process</td>
<td>The name of the process and its threads to plot CPU usage of</td>
</tr>
<tr>
<td>-h, --help</td>
<td>show this help message and exit</td>
</tr>
<tr>
<td>-i, --input</td>
<td>The input file with the Perf events (default = perf_script.txt)</td>
</tr>
<tr>
<td>-s, --start</td>
<td>The relative time to start from in seconds (default = -1, beginning)</td>
</tr>
<tr>
<td>-e, --end</td>
<td>The relative time to end with in seconds (default = -1, end)</td>
</tr>
<tr>
<td>-m, --maxevents</td>
<td>The maximum number of switch events to process (default = 10000)</td>
</tr>
<tr>
<td>-c, --numcpus</td>
<td>The number of CPU cores on the system (default = 4)</td>
</tr>
</tbody>
</table>

**Output:**
A screenshot of the interactive timeline plotted using Grasp is given below.

Appendix C - 5 Output of CoresVisualize.py plotted using Grasp

The output is used to create Figure 30 and record CPU usage.
Workflows

The scripts are located in the AnalysisScripts folder of the Git repository. The following guides shortly describe the workflow to perform the different analysis.

Analyzing Algorithm Steps:

This workflow is used to analyse the performance of the algorithms on the framework from a higher level. It can be used to determine the throughput and latency as well as visualize how steps of an algorithm is scheduled and where the intensive steps are. The analysis is a prelude to the Extended Thread State Analysis. It focuses on the algorithm steps rather than the threads.

1. Run the algorithm of interest using the Reference Framework and allow it to terminate by itself. At termination a log file will be created. This log file is called processUserTraceLog.txt and located in the working directory.
2. Copy the processUserTraceLog.txt file to the folder with the scripts.
3. Run StepStats.py with the algorithm as a command line parameter, i.e. to analyse AutoROI for the first 100 frames run “StepStats.py autoroi –s 1 –e 100”. The script supports all current algorithms. To add an additional algorithm define the order of the steps using their VisicaAgent names in the agentList array.
4. The output of StepStats will provide the FPS, latency and frame time; as well as step times and buffer occupancies for the individual steps.
5. To visualize how frames are executed run StepsVisualize.py and provide an output file and the algorithm of interest, i.e. to visualize AutoROI for the first 100 frames and generate a Grasp file called out.grasp run “StepsVisualize.py out.grasp autoroi –s 1 –e 100”.
6. Open the Grasp executable and select the output file generated in the previous step. It is now possible to zoom and drag the timeline, the frame numbers are annotated on the time line.

Analyzing CPU Cores Usage

This workflow can be used to accurately monitor CPU usage and visualize how a program is scheduled on a system. Other CPU monitoring tools use sampling or averaging, this workflow will give an exact value of CPU usage.

1. In order to see detailed core information the Perf scheduling decisions must be known. This can be profiled by running the ProfileThreadStates.sh bash script with the output directory and command with arguments as the parameters.
2. After this completes copy the perf_script.txt file to the folder with the scripts.
3. Run CoresVisualize.py with the process name and number of CPUs as parameter, i.e. to view the CPU usage of Core during the first second, with a maximum of 1000 events, on a machine with 4 cores run ‘CoresVisualize.py out.grasp ./Core –s 0 –e 1 –m 1000 – c 4’. The script works on any program on Linux.
4. The output on the terminal will provide the exact CPU usage during the period. A file that can be parsed by Grasp will be created called out.grasp.
5. Open the Grasp executable and select the output file generated in the previous step. It is now possible to zoom and drag the timeline. Green represents tasks belonging to the program, orange are idle periods and blue periods where other programs are running.

**Extended Thread State Analysis**

---

**Appendix C - 6 Workflow for applying the Extended Thread State Analysis**

This workflow explains how to run the Extended Thread State Analysis as presented in this thesis work. The analysis is generic and can be applied to any Linux program. It provides a quantitative analysis of thread state occupation time as well as visualization to identify issues and points of improvement.

1. First the application must be run with the ProfileThreadStates bash script using the output directory and command with arguments as the parameters.
2. After the script completes copy the perf_script.txt and perf_stat.txt file to the directory with the scripts. Also copy the processPerfSynchLog.txt that is generated by the framework. This file contains all the user events that can be synchronized to the Perf events.
3. The script tries to estimate the cost of cache stalls, context switching and page faults. To do this a metric_benchmark.csv file is used that contains benchmark results with the cost for a cache miss, context switch or page fault. If the analysis is run on the Raspberry Pi 3 then the existing file can be used. Otherwise the lmbenchmark must be
run and the values updated in the file. See the useful commands in Appendix A: Configuring Linux for Development for the command to run the benchmarks.

4. Run the script to calculate the metrics and state occupation times. To execute the script with the profiling data in the previous steps if the program is called Core run “ThreadstateMetrics.py metric_plots.csv ./Core”. A CSV file called metric_plots.csv is created. The script works with any program on Linux.

5. This script creates a CSV file with the state occupation times. Open the CSV file in Excel select all and convert it to columns (Data -> Text to Columns). Select Delimited and use Comma as the separator. In the Data Preview on the next screen select all the columns and click Advanced with “.” as the Decimal separator and none as the Thousands separator. Click Finish.

6. Copy the formatted data and paste it into the metric_plots.xlsx sheet. The graphs will update and can be analysed.

7. To visualize the thread states run ThreadStateVisualize.py with the process as parameter, i.e. to view the thread states of Core during the first second with a maximum of 1000 events run ‘ThreadStateVisualize.py out.grasp ./Core –s 0 –e 1 –m 1000’. The script works with any program on Linux.

8. Open the Grasp executable and select the output file generated in the previous step. It is now possible to zoom and drag the timeline. The different colours represent the different states. Arrows indicate the arrival of frames, and red lines the parallel regions. The same colours are used as in the graphs. If the arrows and parallel regions are not correctly aligned, adjust the synchronization offset in the previous step.