Towards a Generic Scan Analysis Framework

Dennis Hendrik Henkes (0760594)
d.h.henkes@student.tue.nl

April 25, 2014
“We can’t solve problems by using the same kind of thinking we used when we created them.”

Albert Einstein
Abstract/
Management Summary

Océ printers uses calibration procedures to keep their print qualities to the fullest. As part of such a procedure, the scan of a printed test chart is evaluated. These charts include patterns and each of them must be processed by a set of Scan Analysis Algorithms. Another class of algorithms are those used for printing sheets. Recently, a multiprocessor Image Processing Framework (IPF) has been developed that is limited to the execution of these Print Algorithms. Advantages of the IPF such as flexibility and predictability gave rise to further development. In this project, a solution is developed that allows the IPF to also process the test charts.

The solution consists of a Scan Analysis Framework (SAF) that includes a variable number of identical workers. The workers run in parallel and are used to process patterns on a test chart. This processing is done by arbitrary Scan Analysis Algorithms which can easily be added and removed. A vital component of the SAF is the scheduler, which attempts to equally distribute the loads induced by the patterns over the workers. The observation that many patterns have fairly similar processing times was crucial and has been exploited by packing these into bags. It has also led to the introduction of the Bag Scheduling Problem in which a set of bags must be scheduled onto a set of identical workers, where each bag has a set of jobs that have similar processing times. Furthermore, it is assumed that each worker can process any job from the set. To solve this type of problem, the Vector-Scheduler is proposed. Experiments using synthetic, realistic bag sets revealed that this scheduler has a remarkably fast running time for large job sets when the number of bags is low. It also became evident that the quality of the resulting schedules were comparable to those of the widely accepted LPT-Scheduler.

Three Scan Analysis Algorithms are successfully implemented in the framework, showing that it works properly. Furthermore, several quality aspects of the SAF are investigated experimentally. This reveals that the overhead induced by the SAF is negligible and that it processes test charts time efficiently.
Preface

This thesis marks the end of my rather turbulent study career and was written for acquiring a master’s degree in Embedded Systems from Eindhoven University of Technology. It elaborates on the graduation assignment and presents the developed solutions. Moreover, it is a reflection of both my professional and academic skills, allowing for assessment of the performed work.

The project is conducted at Océ-Technologies, a leading company in developing and manufacturing printing systems. They offered me the right tools and equipment to successfully complete the challenging and multifaceted assignment. Working out this assignment has brought me to a new level of thinking, developing and working, which will hopefully serve me well in the future.

Finally, I turn to those who have helped me performing this graduation project. First of all, I would like to acknowledge the active involvement and inspiring guidance of my Océ supervisor A. Lint, whose insights has led to a significant improvement of both the developed software and this thesis. Moreover, he has introduced me to his close colleagues early on, making me feel comfortable from the start. Another significant player has been R. Mak, my TU/e supervisor, who not only provided me with excellent technical advice, but also offered valuable feedback that has strengthened this thesis in significant ways. Also, I would like to thank S. Stuijk for participating my graduation committee. The already mentioned colleagues have provided me with crucial insights during meetings and casual chats. Finally, there are my parents, sister and brother, who have always supported me in every way, for which I am deeply grateful. And not to be forgotten are my friends, who have been of great help before and during this project.

Dennis Henkes
Eindhoven, 2014
# List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AP</td>
<td>Algorithm Processor</td>
</tr>
<tr>
<td>CMYK</td>
<td>Cyan, Magenta, Yellow and Key (black)</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>DAG</td>
<td>Directed Acyclic Graph</td>
</tr>
<tr>
<td>DLL</td>
<td>Dynamic Link Libraries</td>
</tr>
<tr>
<td>GPU</td>
<td>Graphical Processing Unit</td>
</tr>
<tr>
<td>IPF</td>
<td>Image Processing Framework</td>
</tr>
<tr>
<td>KPN</td>
<td>Kahn Process Network</td>
</tr>
<tr>
<td>OLS</td>
<td>Ordinary Least Squares</td>
</tr>
<tr>
<td>OPT</td>
<td>Optimal result</td>
</tr>
<tr>
<td>RAS</td>
<td>Sun Raster Filer</td>
</tr>
<tr>
<td>RGB</td>
<td>Red, Green and Blue</td>
</tr>
<tr>
<td>SAA</td>
<td>Scan Analysis Algorithm</td>
</tr>
<tr>
<td>SAF</td>
<td>Scan Analysis Framework</td>
</tr>
<tr>
<td>SDF</td>
<td>Synchronous Data Flow</td>
</tr>
<tr>
<td>TSF</td>
<td>Training Set File</td>
</tr>
</tbody>
</table>
# List of Symbols

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td>Algorithm Processor</td>
</tr>
<tr>
<td>$B$</td>
<td>Set of bags</td>
</tr>
<tr>
<td>$</td>
<td>B</td>
</tr>
<tr>
<td>$l_i$</td>
<td>Load of worker $w_i$</td>
</tr>
<tr>
<td>$n_{bj}$</td>
<td>Number of jobs in bag $b_j$</td>
</tr>
<tr>
<td>$\mathbf{n}$</td>
<td>Vector with the numbers of jobs in a set of bags</td>
</tr>
<tr>
<td>$S$</td>
<td>A schedule</td>
</tr>
<tr>
<td>$s_i$</td>
<td>Schedule for worker $w_i$</td>
</tr>
<tr>
<td>$s_{i,j}$</td>
<td>Number of jobs worker $w_i$ processes from bag $b_j$</td>
</tr>
<tr>
<td>$t_{a,\varphi}$</td>
<td>Processing time of a job $\varphi$ in Algorithm Processor $a$</td>
</tr>
<tr>
<td>$t_{b_j}$</td>
<td>Processing time of any job in bag $b_j$</td>
</tr>
<tr>
<td>$\mathbf{t}$</td>
<td>Vector with processing times of jobs in a set of bags</td>
</tr>
<tr>
<td>$w_i$</td>
<td>Worker $i$</td>
</tr>
<tr>
<td>$W$</td>
<td>Set of identical workers</td>
</tr>
<tr>
<td>$</td>
<td>W</td>
</tr>
</tbody>
</table>
# Contents

1 Introduction .................................. 1

2 Problem Description .......................... 3

3 The Scan Analysis Framework ................. 7  
   3.1 Job Manager .................................. 8  
   3.2 A Worker .................................... 13  
   3.3 Job Accumulator ............................. 14  
   3.4 Broker ...................................... 14

4 The Bag Scheduling Problem ................. 15  
   4.1 The Vector-Scheduler ........................ 16

5 The Scan Analysis Algorithms ............... 25  
   5.1 Find Markers ................................ 26  
   5.2 Vertical Averages ............................ 27  
   5.3 Find Stripes ................................ 28  
   5.4 Find Edges .................................. 28  
   5.5 Performance Evaluation ..................... 28

6 Analysis .................................... 31  
   6.1 Overhead of the Scan Analysis Framework 31  
   6.2 Scheduler ................................... 34  
   6.3 Processing Time Prediction ................. 39  
   6.4 A Total Test ................................ 41

7 Conclusion .................................. 43

A Profiler Tool ................................ 45

Bibliography ................................... 47

Index ......................................... 48
Chapter 1

Introduction

Océ-Technologies is a Netherlands-based company that develops, manufactures and sells printing and copying systems. This graduation project is conducted at the “Research and Development” department located in Venlo. This division develops and designs printers for large format and cut sheet markets. Figure 1.1 presents an Océ printer.

![Figure 1.1: Océ printer.](image)

The act of printing sheets includes variables that need to be controlled, keeping the quality of each individual printed sheet to its full potential. This is done via calibration procedures, which are regularly executed. During such a procedure, a special test chart is printed and subsequently scanned. This scan is then evaluated by Scan Analysis Algorithms (SAAs) that are designed to identify flaws in a printing system.

Until recently, Océ-Technologies mainly realizes image processing algorithms in dedicated hardware. However, in the light of saving design time and costs as well as improving flexibility, there is a shift towards implementing these algorithms in software. To facilitate this, an Image Processing Framework (IPF), which can be deployed on a multiprocessor system, is being developed. This framework realizes a multicore pipeline that offers steady resource allocations, making predictable processing possible. At the start of this project, this framework was able to execute print algorithms time efficiently, which serves the process of printing sheets. This graduation project is concerned with extending the usability of the IPF in such a way that it can also process arbitrary test charts by using existing and forthcoming SAAs. This extension is realized through a generic Scan Analysis Framework (SAF), which is developed and tested throughout this project.
This thesis is organized as follows:

- The first two chapters are essentially preparatory, which describe the context (this chapter) and present an elaborative problem description (chapter two).
- Chapter three presents the architecture of the developed SAF, discusses its components and their relations.
- Chapter four introduces the Bag Scheduling Problem, which raises in the SAF, and proposes the Vector-Scheduler.
- Chapter five deals with the implementation of a set of SAAs in the SAF.
- Finally, chapters six and seven deal with the evaluation of the SAF and the quality of the Vector-Scheduler.
Chapter 2

Problem Description

This graduation project focusses on the processing of test charts by using the existing Image Processing Framework (IPF). This processing is part of a calibration procedure which is used to determine the quality of the printing system. Such a procedure starts with creating a descriptor of the test chart. Using this descriptor, the test chart is printed once and the printed sheet is subsequently scanned. Both the descriptor and the scan data are stored in the main memory of the computer that performs the image processing. The image processing must be performed by using the IPF.

The IPF is designed to support the execution of image processing algorithms that are implemented in so called Processors (not to be confused with CPUs); these Processors must be compiled as Dynamic Link Libraries (DLLs). It creates a replicated parallel pipeline of these Processors, controls the execution of this pipeline and manages the dataflow in the pipeline. In a replicated parallel pipeline, multiple instances (replicas) of a processor are executed concurrently. Figure 2.1 presents the structure of such a pipeline; in this example, Processor two and three (P2 and P3) have several replicas running in parallel.

![Figure 2.1: Replicated parallel pipeline of Processors.](image)

The setup of the pipeline is controlled by a script file, which specifies the Processors to be included and the connections among them. Moreover, the script file allows for setting parameters for the Processors—these can be requested in the Processor. The resulting pipeline is executed in a single process\(^1\) and may have multiple threads assigned to it. The IPF attempts to map these threads onto the processing system such that each core executes one thread at most. If there are more threads than cores, at least one core must execute more than one thread (the pigeonhole principle). In this case, the threads on the

---

\(^1\)Hence, the algorithms share an address space.
cores are scheduled using the round robin approach. Data is communicated by means of pointers to shared memory. More precisely, a Processor instantiates and initializes a data structure and sends a pointer to the target Processor.

Test charts ultimately consist of primitives, such as circles and rectangles, and each primitive must be processed by a set of Scan Analysis Algorithms (SAAs). The processing can lead to three types of results, namely A, B and C. An A-type result comprises the outcome of an evaluation and is used to examine functionalities of the printer. A B-type result is used by successive algorithms, i.e. as a preparation for upcoming primitives. A C-type result is a modification of the scan data. The processing of primitives may require surrounding pixels. Figure 2.2 shows a primitive (thick vertical line) and the required pixels to process it (dotted box).

![Figure 2.2: Processing a primitive.](image)

The processing of primitives may be subjected to precedence constraints, which can be modeled by a set of independent Directed Acyclic Graphs (DAGs). Each DAG models the relationship among a set of primitives, and since not all primitives are related (in any way), it results in a set of DAGs that does not have any relationship. A typical DAG is presented in Figure 2.3, where first a set of primitives is used to perform multiple initializations. Then, other primitives are used to prepare the scan data and finally, a set of primitives is used to evaluate a certain functionality. The nodes are labeled with the type of result its processing leads to.

![Figure 2.3: A typical precedence graph.](image)

As is illustrated in Figure 2.4, adjacent primitives may have overlapping pixels (these are indicated by a gray shaded area) that are needed to process them. Some algorithms are optimized to take advantage of this phenomenon, and are able to process sets of adjacent primitives. Consequently, the data of overlapping pixels are reused, attempting to exploit the CPU’s cache. Such a set of primitives is referred to as a pattern, hence any subset of adjacent primitives in a pattern can be processed jointly. Note that a pattern only indicates what adjacent primitives are suitable to be processed together.
The exact composition of a test chart as well as information on how it must be processed is specified in a test chart descriptor. Listing 2.1 shows an example of a descriptor that has three rectangular shaped primitives which must be processed by the SAA called findStripes.

```
push pattern
push saa | name findStripes | modifyScan 0 | processingParameters 1 |
          | parameter PA 42 7     | parameter PB 10 45
draw rectangle | parameter position 10 20 | parameter dimension 8 4
draw rectangle | parameter position 10 22 | parameter dimension 8 4
draw rectangle | parameter position 10 24 | parameter dimension 8 4
pop saa
pop pattern
```

Listing 2.1: Example of a test chart descriptor.

The descriptors have push-pop structures, allowing for the use of sections. A section is opened (pushed) with a set of parameters; these hold till the section is closed (popped). Pushing a section is done with the following syntax

```
push <type> | <attributes> | <list of parameters>
```

There are two types of sections, namely pattern and saa. A pattern is used to group primitives and other patterns together and has no attributes nor parameters. An saa defines an SAA; all primitives in such a section are processed by this algorithm. This type has the attributes name, modifyScan and processingParameters. The latter two can either be zero or one, expressing false and true respectively. For instance, in Listing 2.1, the three primitives must be processed by an SAA called findStripes, which does not modify the scan data but does generate processing parameters. An saa also includes a parameter set. Each saa can have an arbitrary number of parameters, and multiple parameters are separated by ‘|’. A parameter has a key (name) and a set of value, and is listed as follows

```
parameter <key> <val> <val> ... <val>
```

The saa in Listing 2.1 has two parameters, PA and PB, both having two values. saa sections can be nested and cannot contain patterns. A primitive is listed using the following syntax

```
draw <shape> | <list of parameters>
```

The shape can have an arbitrary name, since they are solely used by SAAs. The syntax of a parameter is equal to the one specified above.
Assignment

The main objective of this graduation assignment is to develop a generic Scan Analysis Framework (SAF) to which arbitrary SAAs can easily be added. It has to be able to identify primitives and patterns in an arbitrary test chart descriptor and let them be processed by the added SAAs. The extracted work (from the descriptor) must be scheduled in such a way that the makespan—total time required to perform all of the processing—is minimized while the precedence constraints are preserved. Finally, the results of the processing must be accumulated to one processing result.

The framework must be generic in the sense that it is able to exploit any number of available cores, can include arbitrary SAAs—given that the interface is standardized—and can run on arbitrary homogeneous x86 CPUs. The latter implies that the algorithms cannot be profiled on processing times and the SAF must have a self-learning method to predict the processing times of patterns and primitives. These predictions are required to derive processing schedules in which the makespans are minimized.

The assignment is extended with the development of C++ implementations of the SAAs that are used to evaluate the operational conditions of printing elements. There are Matlab implementations available of these algorithms, which can be taken as a starting point and used to verify the developed implementations. Moreover, the C++ implementations serve as experimental evidence that the developed framework functions properly.

To summarize, the deliverables of this graduation project are:

- A generic Scan Analysis Framework.
- C++ implementations of SAAs that evaluate printing elements.
Chapter 3

The Scan Analysis Framework

The developed Scan Analysis Framework (SAF) consists of a fork-join pipeline that has three generic components and a set of identical workers $W$, which run in parallel. Figure 3.1 depicts this setup. The SAF is executed by the existing Image Processing Framework (IPF).

![Figure 3.1: Architecture of the solution.](image)

The first component is generic and is called the Job Manager. This manager reads the Processing Request, which contains a scan and a test chart descriptor, and derives a set of jobs accordingly. Each job is related to a set of primitives and describes how they must be processed. Depending on the number of primitives and the required processing, a job will require a certain amount of processing time. These times are predicted based on statistics of previously processed jobs; these statistics are reported by the Job Accumulator. Next, using these predictions, the jobs are balanced over the available workers. The workers are identical and consist of a series of Algorithm Processors (APs), which are adapted IPF Processors. Each AP implements one Scan Analysis Algorithm (SAA). At startup time, each AP registers the name of the SAA it implements by the Broker. (The Job Manager and Job Accumulator uses this Broker to discover what SAAs are present in the workers and in which order.) The jobs traverse the complete series of APs, and only the specified APs perform the scan analysis on it. Processing Parameters as well as Processing Results are stored in the job. A Processing Parameter is a parameter that is produced by an SAA for successive SAAs. Hence, SAAs communicate parameters to successive SAAs via jobs.
A Processing Result is an outcome that must be saved in the (final) Processing Result. This result is created by the last component in the pipeline—the generic Job Accumulator. This component accumulates all processing results that are stored in the jobs and forwards these as a Processing Result. Successive IPF Processors use the results to complete the calibration procedure. As hinted above, the Job Accumulator also registers processing time statistics. These are saved in Training Set Files. Figure 3.2 depicts the deployment of a SAF that include $W$ workers.

### 3.1 Job Manager

The Job Manager consists of four components, namely the Job Chain Creator, Scheduler, Dispatcher and Regressor. The Regressor is only executed at start-up time. Figure 3.3 depicts the setup of these components—the off-line Regressor is visualized by a framed box.

The Job Chain Creator receives a Processing Request, which contains one or two scan(s) (Sun Raster files) of a test chart and the corresponding test chart descriptor. First, the creator uses the descriptor to derive a set of jobs chains. Job chains are independent of each other and each encompasses a set of jobs which must be processed by a single worker in the order as specified by the chain, without the intervention of any other job. Next, the processing time of each chain is predicted using prediction formulas. These formulas are
constructed by the Regressor and are based on statistics of previously processed jobs that are stored in Training Set Files. Then, the job chains are packed into bags such that each bag has chains with similar processing times. The bags are sent to the Dispatcher and definitions of the bags are sent to the Scheduler. The Scheduler calculates a processing schedule in which the makespan is minimized and communicates this to the Dispatcher. Once the Dispatcher has all the job chains and the processing schedule, it loads the jobs (not the chains) into the worker queues in compliance with the schedule while respecting the order enforced by the chains.

3.1.1 Job Chain Creator

The Job Chain Creator first initializes an Image Descriptor from which SAAs can request image properties as well as snippets (fragments of the sheet). These snippets are rectangular and include one of the following color channels: red, green, blue, cyan, magenta, yellow or black. The position (upper left corner) of the snippet is given with respect to the upper left corner of the sheet. Since the scans include a view of the environment, the edges of the sheet are detected, leading to the position of the sheet’s upper left corner. The algorithm to find these edges is based on an existing Matlab implementation and is described in Section 5.4.

When the sheet is scanned by means of two scanners, a left scan and a right scan is received. It is presumed that these scans are aligned in the vertical direction and have an overlap in the horizontal direction. The Creator determines where the right image starts in the left image—now it can be determined from which scan a snippet must be taken.

When the image descriptor is instantiated and initialized, jobs are created. Each job corresponds to a pattern that has saa sections and primitives (as defined in the test chart descriptor). For each job \( \varphi \), a processing plan is created that specifies by which APs \( \varphi \) must be processed. The plan ensures that \( \varphi \) is processed by the specified SAAs in the specified order. This mapping is based on the structure of the workers, which is requested from the Broker. In this most general form, each worker has an arbitrary structure and a plan must be derived for each worker, as it is unknown in advance which worker processes \( \varphi \). However, in this project it is assumed that all workers are identical and only one processing plan is derived. If no valid plan exists, the SAF gives an error message and exits. When a valid schedule is derived, an Algorithm Descriptor is created and added to \( \varphi \) for each SAA that processes \( \varphi \). This descriptor contains all parameters that the test chart descriptor specified for that SAA. Finally, for each primitive in the pattern, a Primitive Descriptor is created that has all parameters that are listed for that primitive.

Jobs inherit all precedence relations from the patterns. Jobs that have a precedence relation, are put into a chain in such a way that all these constraints are met. Jobs that have no precedence relations are put into a “private” chain, hence such a chain has only one job in it. The result is a set of chains \( C \) where each chain \( c \in C \) contains a set of jobs that have precedence relations.
To enable the derivation of makespan minimized processing schedules, two things have to be determined for each job chain \( c \in C \)

1. Which workers can process \( c \)
2. Given a worker \( w \) that can process \( c \), how long it will take

More formally, consider a set of workers \( W \) and a set of chains \( C \), then for each \( w \in W \) and each \( c \in C \) let

\[
y_{w,c} = \begin{cases} 
  \text{true}, & \text{if } w \text{ can process } c \\
  \text{false}, & \text{otherwise} 
\end{cases} 
\]  

(3.1)

\[
t_{w,c} = \text{the time it takes } w \text{ to process } c 
\]  

(3.2)

Note that the latter formula has no meaning when \( y_{w,c} \) equals \( \text{false} \). Since the workers are assumed to be identical and the processing system to be homogeneous, the processing time is equal at each worker. Moreover, at this stage, any \( y_{w,c} \) equals \( \text{true} \) since there is a valid processing plan derived for each chain. The processing time of a chain is defined as the sum of the processing times of the jobs in it, where the processing time of a job is defined as the sum of times it spent in each AP. More formally, consider any worker \( w \in W \), a chain \( c \in C \) and let \( t_{a,\varphi} \) denote the time AP \( a \in w \) requires to process job \( \varphi \in c \), then the processing time of \( c \) is given by

\[
t_c = \sum_{\varphi \in c} \sum_{a \in w} t_{a,\varphi} 
\]  

(3.3)

The time \( t_{a,\varphi} \) is estimated by using a prediction formula, which is derived by the Regressor. Such a formula is derived for all SAAs for which a Training Set is available. In case no formula exists, the time is set to 1\( \mu s \), enabling the Scheduler to balance the jobs. When an AP does not process \( \varphi \), the time is set to zero. The prediction formula is in function of algorithm parameters; common primitive parameters and the number of primitives in the job. The parameters are all given by \( \text{key-value} \) pairs, where the keys are used to fill out the formula. When a key is not included in the formula, it is ignored. Note that the AP overhead is neglected; experiments have shown that this overhead is indeed negligible compared to the actual processing (see Sections 5 and 6.1).

The processing times of the chains are used to pack chains with approximately similar times together into a single bag. This results in a set of bags \( B \), where each \( b \in B \) is a bag that has \( n_b \) job chains and any chain has a processing time of \( t_b \). The latter is not totally accurate as it is permitted to have chains in a bag that have a slightly different processing time. The deviation a chain may have in order to join a bag is given as a percentage, which can be adjusted at compile time. The time \( t_b \) is set when the first chain is added to \( b \). Let \( d \) denote the maximal deviation, then a chain \( c \) is added to \( b \) when

\[
\frac{100 - d}{100} t_b \leq t_c \leq \frac{100 + d}{100} t_b
\]  

(3.4)

The resulting set of bags \( B \) are sent to the Dispatcher and the definitions of the bags are sent to the Scheduler.
3.1.2 Scheduler

The Scheduler determines for each worker how many job chains it must process from each bag. Note that such a schedule does not determine an order; this is not needed since the chains are independent. If there is only one bag ($|B| = 1$), the simple scheduling algorithm, as given in Algorithm 1, is used. When $B$ contains two or more bags ($|B| \geq 2$), the more sophisticated Vector-Scheduler algorithm is used. This algorithm embodies one of the main results of this project and is therefore discussed in Chapter 4 separately. The Vector-Scheduler is developed for a more general context, namely to schedule independent jobs. Since the chains are independent and characterized with a processing time, it is a similar problem and can seamlessly be solved by this scheduler.

Algorithm 1: OneBag-Scheduler($W, B$)

1. Initialize $n \leftarrow n_1$
2. for $i \leftarrow 1$ to $|W|$ do
3.      $s_{i,1} \leftarrow \left\lceil \frac{n}{|W|-i+1} \right\rceil$
4.      $n \leftarrow n - s_{i,1}$
5. end
6. return $s$

As a means of a numerical example, consider five workers and the bag definitions as presented in Table 3.1. A possible schedule is given in Table 3.2, where each worker gets two jobs from both the first and second bag, as well as four from the third. The last column indicates the load of each worker, which equals the total work that it has to perform, i.e. the sum of the processing times of the jobs it must process.

<table>
<thead>
<tr>
<th>Bag</th>
<th>Number of Chains</th>
<th>Processing Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$b_1$</td>
<td>10</td>
<td>120ms</td>
</tr>
<tr>
<td>$b_2$</td>
<td>10</td>
<td>200ms</td>
</tr>
<tr>
<td>$b_3$</td>
<td>20</td>
<td>250ms</td>
</tr>
</tbody>
</table>

Table 3.1: Example of bag definitions.

<table>
<thead>
<tr>
<th>Worker</th>
<th>$b_1$</th>
<th>$b_2$</th>
<th>$b_3$</th>
<th>Load</th>
</tr>
</thead>
<tbody>
<tr>
<td>$w_1$</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>1640ms</td>
</tr>
<tr>
<td>$w_2$</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>1640ms</td>
</tr>
<tr>
<td>$w_3$</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>1640ms</td>
</tr>
<tr>
<td>$w_4$</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>1640ms</td>
</tr>
<tr>
<td>$w_5$</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>1640ms</td>
</tr>
</tbody>
</table>

Table 3.2: Possible schedule.

3.1.3 Dispatcher

The Dispatcher puts the jobs into the workers’ queues in accordance with the schedule while respecting the order enforced by the chains.
3.1.4 Regressor

The Regressor derives a formula that is used by the Job Chain Creator to predict the processing time $t_{a,\varphi}$ that an AP $a$ requires to process a parameterized job $\varphi$. The formula to predict $t_{a,\varphi}$ has the following form

$$t_{a,\varphi} = h_{\theta}(x_{\varphi}) = \theta_0 + \theta_1 x_1 + \ldots + \theta_n x_n$$  \hspace{1cm} (3.5)

Hence the predicted processing time equals the inner product (or dot product) of $\theta$ and $x_{\varphi}$. Where $\theta$ is a vector with coefficients that characterizes $a$, and $x_{\varphi}$ is vector with the parameters of $\varphi$; $x_{\varphi}$ has the following structure

$$x_{\varphi} = \begin{pmatrix} x_a \\ x_p \\ n \end{pmatrix} \in \mathbb{R}^{n+1}$$

Where $x_a$ is a vector with the algorithm parameters; $x_p$ is a vector with the common primitive parameters and $n$ equals the number of primitives in $\varphi$.

The theta vector for $a$ is obtained by performing a multivariate data regression over a sample set of previously processed jobs by $a$. (Also known as a Training Set.) The used regression method is called Ordinary Least Squares (OLS). The OLS formula is as follows

$$\theta = (X^T X)^{-1} X^T y$$  \hspace{1cm} (3.6)

In this formula, $X$ and $y$ form the training set. $X$ is a matrix where each row is a sample and each column corresponds to an algorithm parameter, primitive parameter or the number of primitives the job had. $y$ is a vector that has the registered processing time for each sample. The inversion used is Moore-Penrose pseudoinverse.

Each AP has a corresponding Training Set File in which each line is a sample. Each sample consists of algorithm parameters, common primitive parameters, the number of primitives and the registered processing time. This training set is read and saved in four matrices, namely $X_a$, $X_p$, $n$ and $y$. $X_a$ has all algorithm parameters, $X_p$ all common primitive parameters, $n$ the number of primitives and $y$ the registered processing times.

The algorithm and primitive parameters are given by key-value pairs, i.e. a name (key) and its value. The keys are used to load the values into the matrices and all values that have the same key are put into the same column. To quickly determine in which column the value belongs, an Index Resolver has been developed. This resolver is based on the trie (or radix-tree) data structure and returns the column index for a given key. Since the algorithm and primitive parameters may have equal key names, two separate resolvers are used. These resolvers are also used by the Job Chain Creator to map parameters onto the prediction formula. When the four matrices are filled, $X$ is created as follows

$$X = \begin{pmatrix} 1 & X_a & X_p & n \end{pmatrix}$$  \hspace{1cm} (3.7)

This $X$ is reduced to simplify the regression. All equal columns are removed, the resolvers are updated correspondingly. Moreover, equal rows are combined into one row; the new processing time in $y$ equals the average processing time. After these simplifications, the obtained $X$ is used with $y$ in Equation 3.6 to determine $\theta$. 

12 Department of Mathematics and Computer Science
3.2 A Worker

A Worker consists of a linear chain of APs. The composition of a worker is determined by the IPF configuration script. Each processor has a Processing Controller, which is executed by the worker’s thread when there is a job in the input queue of the AP. This is controlled by the IPF. Figure 3.4 depicts the general setup of a worker; AP1 and AP2 are Algorithm Processors one and two respectively.

![Figure 3.4: Structure of a Worker.](image)

An AP has a Processing Controller, Parameter Loader and Scan Analysis Algorithm. In addition, the Inscriber registers the AP (and the name of the SAA it implements) at the Broker during start-up time. Figure 3.5 depicts the setup of an AP, the Inscriber has a framed box as it is an off-line component.

![Figure 3.5: Structure of an Algorithm Processor.](image)

The Processing Controller gets the first job from the input queue and sends it to the Parameter Loader. In this loader, all pointers to parameters that are needed by the SAA are stored locally. When all required pointers are stored, the Processing Controller evaluates whether or not this AP must process the job. In case it must process the job, it registers the start time and then sends the job to the Scan Analysis Algorithm. When the analysis is completed, the Processing Controller registers the end time and puts the job into its output queue.
3.3 Job Accumulator

The Job Accumulator collects and combines stored processing results into one Processing Result and logs performance attributes to Training Set Files. These files are used by the Job Chain Creator of the Job Manager to predict processing times of jobs. The structure of the Job Accumulator is depicted in Figure 3.6.

![Figure 3.6: Structure of the Job Accumulator.](image)

The Accumulator gets processed jobs from the output queue of the workers and stores them locally. Note that only references to jobs are stored. When all jobs are stored, it sends the list of stored jobs to the Result Reporter and Performance Logger. The Result Reporter composes a Processing Result by storing all void pointers to processing results in a Processing Result. This Processing Result is put into the output queue of the Job Accumulator. The Performance Logger stores performance related attributes of processed jobs as follows. Consider a processed job $\varphi$. Then, for each AP $a$ that has processed $\varphi$, a sample is created. This sample consists of all algorithm parameters specified for $a$, all common primitive parameters, the number of primitives in $\varphi$ and the registered processing time. The samples are added to the corresponding Training Set Files.

3.4 Broker

The Broker registers the structure of the workers, i.e. what APs are included and in what order—recall that all workers are identical. This structure is used by the Job Chain Creator of the Job Manager to map the jobs onto the APs in the workers and by the Job Accumulator to store performance results of each AP in the worker.
Chapter 4

The Bag Scheduling Problem

In this chapter, a new scheduling problem is introduced along with an approximation algorithm that solves it. It is expected that the SAF gets bag sets were the number of bags is small, typically in the range $[2, 8]$. First, a formal definition of the problem is given.

**Definition 4.0.1.** A bag is a collection of jobs that have the same processing time.

**Definition 4.0.2.** In the Bag Scheduling Problem, the jobs in a given set of bags are mapped onto a set of identical workers such that the makespan—the total completion time—is minimized.

A large body of research efforts addressing different types of scheduling problems has been proposed in the literature. Examples include the Task Scheduling Problem and the Job Scheduling Problem. Task scheduling is the activity of mapping tasks onto a processing system, which is analogous to scheduling jobs onto workers. Garey and Johnson have proven that the problem of mapping non-preemptive tasks onto a multiprocessor system is $\mathcal{NP}$-complete [1]. (A non-preemptive task does not permit interruptions while being executed.) The tasks may be independent or have some precedence relations among them. In case of the latter, there are roughly two types, namely with and without cyclic dependencies. Models without cyclic dependencies are used to approximate programs with finite behavior (given that each task terminates within a finite amount of time). These programs are mostly modeled by a weighted Task Graph which is a directed acyclic graph (DAG). Examples of popular models that allow for cyclic dependencies include a Kahn Process Network (KPN) [2] and Synchronous Data Flow (SDF). Job scheduling is the process of controlling the job flow in a pipelined system. The flow of jobs is determined by a protocol, also called policy or load balancing protocol.

The defined Bag Scheduling Problem does not (directly) fall into one of these categories. Moreover, to the knowledge of the author, the Bag Scheduling Problem is not defined earlier as in Definition 4.0.2. Though, the term Bag-of-Tasks is used before by Cirne et al. [3], the meaning was quite different. Here, it refers to an application (bag) that is solely composed of independent tasks that have different execution times. Another problem is the Bin Scheduling Problem, proposed by Gunther et al. [4], which name suggests an equivalence with the Bag Scheduling Problem. However, this problem deals
with resources that are available in separate time slots, or shifts, in which tasks must be executed completely. Stimulated by the lack of schedulers that exploit the information bags provide, the Vector-Scheduler has been proposed. First a lower bound of the problem complexity is derived.

**Theorem 4.0.3.** The Bag Scheduling Problem is $\mathcal{NP}$-complete.

**Proof.** By unpacking the set of bags, a set of jobs is obtained. This reduction can be done in linear time, hence quicker than polynomial. The resulting problem is analogous to the classical multiprocessor problem, which is known to be $\mathcal{NP}$-complete [1].

### 4.1 The Vector-Scheduler

The Vector-Scheduler is a heuristic algorithm that solves the Bag Scheduling Problem as given in Definition 4.0.2. The name stems from the fact that it creates schedules by using vectors. This way, groups of jobs are scheduled rather than individual jobs. First, the remainder of this section presents some notations and then a foundation for this scheduler and pseudocode is laid out. It also gives a derivation of both its approximation ratio and running time.

#### 4.1.1 Some Notations

Consider a set of identical workers $W$ and a set of bags $B$. Furthermore, assume that the elements in $W$ and $B$ are numbered, i.e. $W = \{w_1, \ldots, w_{|W|}\}$ and $B = \{b_1, \ldots, b_{|B|}\}$. Let $n_j$ denote the number of jobs in bag $b_j \in B$ and $t_j$ be the processing time of any job in $b_j$. Then, let $n$ and $t$ denote the vectors whose entries give the number of jobs and processing times of each bag $b \in B$, where the processing times in $t$ are sorted in decreasing order

\[ n = \begin{pmatrix} n_1 \\ \vdots \\ n_{|B|} \end{pmatrix} \in \mathbb{Z}^{|B|} \] (4.1)

\[ t = \begin{pmatrix} t_1 \\ \vdots \\ t_{|B|} \end{pmatrix} \in \mathbb{Z}^{|B|} \] (4.2)

\[ \forall j \in \mathbb{Z}, (1 \leq j < |B|) \Rightarrow (t_j > t_{j+1}) \] (4.3)

A schedule $S$ has $|W|$ vectors, a vector $s_i \in S$ defines for worker $w_i \in W$ the number of jobs it must process from each $b \in B$

\[ s_i = \begin{pmatrix} s_{i,1} \\ \vdots \\ s_{i,|B|} \end{pmatrix} \in \mathbb{Z}^{|B|} \] (4.4)

Where $s_{i,j} \in s_i$ denotes the number of jobs worker $w_i$ must process from bag $b_j \in B$. The load $l_i$ of worker $w_i$ is given by

\[ l_i = s_i^T t \] (4.5)
and the makespan $T_{ms}$ equals

$$T_{ms} = \max_{1 \leq i \leq |W|} l_i$$ (4.6)

### 4.1.2 The Algorithm

The Vector-Scheduler attempts to assign to each worker the same collection of jobs, and schedules the workers one by one. The first $|W| - 1$ workers are scheduled using a three-step approach, the last worker gets all remaining jobs. Each step in this approach results in a vector and the schedule $s_i$ for worker $w_i$ equals the sum of these vectors: $s_i = s_i' + s_i'' + s_i'''$ with $s_i, s_i', s_i'', s_i''' \in \mathbb{Z}^{|B|}$. The upcoming text discusses how a worker $w_i$ is scheduled using the three steps. While scheduling $w_i$, the symbols $n_j(i)$, $n_j'(i)$ and $n_j''(i)$ are used to indicate the number of jobs in bag $b_j$ just before step one, two and three respectively. In addition, $L(i)$ denotes the total unscheduled load just before worker $w_i$ gets scheduled

$$L(i) = \sum_{1 \leq j \leq |B|} (n_j^T(i) t_j)$$ (4.7)

Consequently, the ideal or best load $l_i^*$ worker $w_i$ can have is given by

$$l_i^* = \frac{L(i)}{|W| - i + 1}$$ (4.8)

#### Step One

In this step, the non-empty bag that has the jobs with the greatest processing times is considered. From this bag worker $w_i$ gets the rounded up average number of jobs, where the average is the number of jobs in the bag divided by the number of unscheduled workers $(|W| - i + 1)$. It is assumed that the jobs from the considered bag are the hardest to schedule among the unscheduled jobs in $B$. Moreover, the number of unscheduled jobs that can compensate for the large jobs is less while scheduling the next workers. Therefore, the current worker gets the rounded up average. Let $b_{j+} \in B$ denote the non-empty bag that has the jobs with the greatest processing time, then $s_i'$ is assigned as follows

$$s_i' = \begin{bmatrix} 0 \\ n_{j+}(i) \\ \frac{n_{j+}(i)}{|W| - i + 1} \\ 0 \end{bmatrix}$$ (4.9)

Let $l_s'$ denote the slack load of $w_i$ right after the first step

$$l_s' = l_i^* - (s_i')^T t$$ (4.10)

When the slack load is zero or negative, adding jobs deteriorates the makespan and hence no jobs are added in this stage nor the next, thus $s_i''' = s_i'' = 0$. Note that in the first case, jobs could be replaced by smaller jobs, aiming at getting closer to $l_i^*$. However, then another worker gets at least the number of jobs from $b_{j+}$ that $w_i$ got. When the slack load is positive, adding jobs may decrease the error $l_i'' = |l_s'|$. This is attempted in the following two steps.
Step Two

In this step, the remaining bags are considered. Worker \( w_i \) gets a rounded down fraction of the average number of jobs from each of these bags, where the average is again the number of jobs in the bag divided by the number of unscheduled workers. The fraction is a real number between zero and one and accounts for the fact that in the first stage more jobs might be added than on average. Let \( l_r \) denote the average load in the remaining bags

\[
l_r = \frac{1}{|W| - i + 1} \left( \sum_{1 \leq j \leq |B|} (n'_j(i) - n'_j(i)) \right)
\]

Then, the fraction of load that results in the best load \( l^*_i \) yields

\[
f = \frac{l'_s}{l_r}
\]

and \( s''_i \) is assigned as follows

\[
s''_i = \begin{pmatrix} 0 \\ f \cdot \frac{n'_{i+1}(i)}{|W| - i + 1} \\ \vdots \\ f \cdot \frac{n'_{|B|}(i)}{|W| - i + 1} \end{pmatrix}
\]

Let \( l''_s \) denote the slack load right after step two

\[
l''_s = (s'_i + s''_i)^T t
\]

Lemma 4.1.1. If the slack load before step two \( (l'_s) \) is greater than zero, then the slack load after step two \( (l''_s) \) is in the range \( [0, 1^T t] \).

Proof. Assume that the floor operations are omitted while \( s''_i \) is assigned, then the added load in step one plus the added load in two is equal to the best load \( l^*_i \), leading to a slack load of zero. Now, suppose that the floor operations are applied. The resulting load is never higher than the load without the floor operation, and therefore the slack load cannot be lower than zero. Moreover, the floor operations discard less than one job from each bag, increasing the slack load \( l'_s \) with less than \( 1^T t \), which concludes the proof.

Step Three

In this step, the remaining error \( (l''_s = |l''_s|) \) is reduced by repeatedly adding a vector \( v \in \{0, 1\}^{|B|} \) to \( s''_i \), where \( v \) includes at most one job from each non-empty bag. (Initially \( s''_i = 0 \).) Moreover, each \( v \) is selected as such that it causes the maximal error reduction possible. Adding vectors stops when either there is positive slack load and adding any vector other than \( 0 \) would lead to a higher negative slack load or a negative slack load is obtained already. The expectation is that only one vector is added, which essentially rounds up the best candidates in the previous step. Moreover, letting a vector include one job from each bag at most, contributes to giving each worker the same collection of
jobs. However, adding only a single vector can lead to bad schedules, especially when there is one bag with jobs that have processing times much greater than the others. As an example, consider a bag set of two bags where the first bag has jobs with processing times much greater than those in the second. When the slack load is not big enough to include a job from the first bag and much greater than the processing time of jobs in the second bag. Then, only one job from the second bag is included, resulting in a large error while several jobs from the second bag could be added to significantly decrease this error. Therefore, vectors are added until it starts increasing the error.

Let \( V(k) \) denote the set of vectors that can be created after \( k \) vectors are added and let \( l_v \) be the load of a vector \( v \in V \). Consider the \( k \)th vector \( v_k \) that is added and let \( l_s(k) \) denote the slack load just after it is added, then by definition

\[
|l_s(k) - l_{v_k}| = |l_s(k)| \leq |l_s(k - 1)| \tag{4.15}
\]

\[
\forall v \in V(k), |l_s(k - 1) - l_{v_k}| \leq |l_s(k - 1) - l_v| \tag{4.16}
\]

To determine \( v \), a static list is built. This list includes all vectors that can be made by using zero or one job from each bag and, hence \( 2^{|B|} \) vectors. (Recall that the scheduler is designed for problem instances that have a few bags.) The vectors are sorted on the load it induces, i.e. \( v^T \cdot t \). Now, with a slack load given, finding the closest vector is done by first searching the vector that induces the greatest processing load that is equal or smaller than the slack load. From this vector, two searches are started. The first starts in the found vector and determines whether or not it can be composed of the unscheduled jobs. If this is possible it is the first candidate for \( v \). Otherwise the algorithm searches downwards until it finds a constructible vector. The second search does the same upwards, starting its search one vector upwards.

**Lemma 4.1.2.** Given a slack load, there always is at least one vector that induces a load smaller and one vector that induces a load higher than this slack load.

**Proof.** The vector that does not include any job is the null-vector \((0)\) and induces zero load, which is lower than the given slack load since this load is strictly positive. For the vector that induces a load higher than the slack load, consider the assignment of \( s_i''\) in step two. Since the slack load is positive, at least one entry is rounded down. Moreover, for each entry that is rounded down, there is at least one job in the corresponding bag. Therefore, the vector that includes one job at the entries that are rounded down is constructible. Since this vector includes “whole” jobs it induces a load that is strictly greater than the slack load.

Then, the resulting error is determined for both vectors and the one with the lowest error is chosen. In case of a tie, the vector that leads to a negative slack load is chosen.

**Lemma 4.1.3.** If the slack load \( l_s' \) is greater than zero before step two, then the slack load \( l_s'' \) is in the range \([-\frac{1}{2}t_1, \frac{1}{2}t_1]\) after step three.

**Proof.** Since the slack load before step two is greater than zero, the slack load after step two is greater or equal to zero, by using Lemma 4.1.1. If the slack load is equal to zero, no jobs are added in step three and \(-\frac{1}{2}t_1 \leq l_s'' \leq \frac{1}{2}t_1\) holds trivially (since \( t_1 \) is strictly
greater than zero). When the slack load is greater than zero, there is an error and the Vector-Scheduler attempts to reduce this error in step three. Now it remains to prove that when a negative slack load is obtained, it is higher or equal to \(-\frac{1}{2}t_1\) and when the error cannot be lowered, the slack load is smaller or equal to \(\frac{1}{2}t_1\). To this end, consider the last \(v\) added to \(s''''\) 

- **\(v = 0\)**: any combination of unscheduled jobs leads to a negative slack load that has an absolute value greater than the current slack load. Let \(b_{j-} \in B\) denote the non-empty bag that has the smallest processing time, then \(l''''_s(k) < \frac{1}{2}t_{j-}\). From Proposition 4.3 follows that \(\frac{1}{2}t_{j-} \leq \frac{1}{2}t_1\) and \(l''''_s(k) \leq \frac{1}{2}t_1\).

- **\(v \neq 0\)**: since this is the last added \(v\), the resulting slack load is negative. Moreover, removing any job from \(v\) leads to a positive slack load that is greater than the absolute value of the current slack load. This implies that for any job \(j \in v\) holds \(-\frac{1}{2}t_j \leq l''''_s(k)\). From Proposition 4.3 follows that \(-\frac{1}{2}t_1 \leq \frac{1}{2}t_j\) and \(-\frac{1}{2}t_1 \leq l''''_s(k)\) and therefore \(-\frac{1}{2}t_1 \leq l''''_s(k)\).
4.1.3 Pseudocode of the Vector-Scheduler

Algorithm 2 presents pseudocode of the Vector-Scheduler.

**Algorithm 2:** Vector-Scheduler\((W,B)\)

1. Build vector list \(V\) using \(B\)
2. Initialize \(L = \sum_{1 \leq j \leq |B|} n_j t_j\)
3. for \(i = 1\) to \(|W| - 1\) do
   1. Initialize \(s_{ij} = 0\) for \(1 \leq j \leq |B|\)
   2. // Step one
   3. Find a \(j^+\) such that \(t_{j^+} = \max_{1 \leq j \leq |B|} t_j \land n_{j^+} \neq 0\)
   4. \(s_{i,j^+} = \left\lceil \frac{n_{j^+}}{|W| - i + 1} \right\rceil\)
   5. \(n_{j^+} = n_{j^+} - s_{i,j^+}; \ l_s = l_b - s_{i,j^+}; \ L = L - s_{i,j^+}\)
   6. \(l_b = \frac{L}{|W| - i + 1}\)
   7. if \(l_s < 0\) then
      1. // Step two
      2. \(l_r = \frac{L - n_{j^+} t_{j^+}}{|W| - i + 1}\)
      3. \(f = \frac{l_s}{l_r}\)
      4. for \(j = j^+ + 1\) to \(|B|\) do
         1. \(s_{i,j} = \left\lfloor \frac{f n_j}{|W| - i + 1} \right\rfloor\)
         2. \(n_j = n_j - s_{i,j}; \ l_s = l_s - (s_{i,j} t_j); \ L = L - (s_{i,j} t_j)\)
      end
      6. // Step three
     7. repeat
       1. \(v = V.GetClosest(l_s, B)\)
       2. for \(j = 1\) to \(|B|\) do
          1. \(s_{i,j} = s_{i,j} + v_j\)
          2. \(n_j = n_j - v_j; \ l_s = l_s - (v_j t_j); \ L = L - (v_j t_j)\)
       end
     8. until \((l_s < 0 \lor v == 0)\)
  end
28. for \(j = 1\) to \(|B|\) do
   1. \(s_{|W|, j} = n_j\)
end
31. return \(S\)
4.1.4 Theoretical Analysis

This section deals with the approximation ratio and the time complexity of the proposed Vector-Scheduler.

**Approximation Ratio**

Given a set of bags \( B \) and a set of identical workers \( W \), there is mapping of the jobs in \( B \) onto \( W \) such that there is no other mapping that leads to a lower makespan. This mapping is called the optimal mapping and the resulting makespan equals \( \text{OPT} \). The following Theorem gives the deviation the makespan of a schedule, that is produced by the Vector-Scheduler, can have at most.

**Theorem 4.1.4.** Given set of at least two workers, the Vector-Scheduler has an approximation ratio of

\[
\frac{3}{2} + \ln \left( \sqrt{|W| - 1} \right).
\]

**Proof.** The lower bound used is based on two observations. First, ideally all jobs are scheduled so that all workers have the same load, namely \( \frac{n^Tt}{|W|} \). Second, there may exist jobs that have processing times that exceed this average. Therefore, the lower bound is defined as the maximum of these two

\[
LB := \max \left( \frac{n^Tt}{|W|}, t_1 \right)
\]  

(4.17)

For the derivation of the approximation ratio, consider a set of workers \( W \) and a set of bags \( B \). A crucial property of the Vector-Scheduler is that it allows a worker to have a slack load. When a worker has a slack load, the best load of the unscheduled workers increases with this slack load divided by the number of unscheduled workers. Consequently, the load of the last worker \( w_{|W|} \) equals the average load plus the weighted sum of the slack loads. In the worst case, each worker (but the last) has the greatest slack load it can have. According to Lemma 4.1.3, the greatest slack load a worker can have equals \( \frac{1}{2}t_1 \). From this follows that the sum of the worst case slack loads yields

\[
\frac{\frac{1}{2}t_1}{|W| - 1} + \frac{\frac{1}{2}t_1}{|W| - 2} + \cdots + \frac{\frac{1}{2}t_1}{|W| - |W| + 1}
\]

\[
= \frac{\frac{1}{2}t_1}{1} + \frac{\frac{1}{2}t_1}{2} + \cdots + \frac{\frac{1}{2}t_1}{|W| - 1}
\]

\[
= \left( \frac{1}{1} + \frac{1}{2} + \cdots + \frac{1}{|W| - 1} \right) \frac{1}{2}t_1
\]

\[
= H_{(|W| - 1)} \frac{1}{2}t_1
\]

\[
\leq \left( \ln (|W| - 1) + 1 \right) \frac{1}{2}t_1 \quad \text{with } |W| > 1
\]

The worst case load of worker \( w_{|W| - 1} \) is equal to worst case load of \( w_{|W|} \), which occurs when all workers scheduled before \( w_{|W| - 1} \) have the greatest slack load and \( w_{|W| - 1} \) has got the greatest negative slack load. The worst case of all other workers is lower, as there is less slack load accumulated. From this follows that the makespan \( T_{ms} \) is lower than the worst case load of \( w_{|W|} \)

\[
T_{ms} \leq \frac{n^Tt}{|W|} + \left( \ln (|W| - 1) + 1 \right) \frac{1}{2}t_1
\]

(4.18)
By using this inequality and the definition of the lower bound in Equation 4.17, the following approximation ratio can be derived

\[
T_{ms} \leq n^T t \frac{1}{|W|} + (\ln(|W| - 1) + 1) \frac{1}{2} t_1 \\
\leq LB + (\ln(|W| - 1) + 1) \frac{1}{2} LB \\
= \left(\frac{3}{2} + \frac{1}{2} \ln(|W| - 1)\right) LB \\
= \left(\frac{3}{2} + \ln\left(\sqrt{|W| - 1}\right)\right) LB \\
\leq \left(\frac{3}{2} + \ln\left(\sqrt{|W| - 1}\right)\right) OPT
\]

\[\Box\]

The following Corollary extends this analysis and is concerned in cases where the jobs in a bag have deviations in their processing times.

**Corollary 4.1.5.** Given a bag set in which the deviation of jobs in a bag is bounded by ±d per cent, the Vector-Scheduler has an approximation ratio of \(\frac{3}{2} + \ln\left(\sqrt{|W| - 1}\right)\) \(\frac{100 + d}{100}\).

**Proof.** In the worst case, the worker that has the greatest load, and therefore determines the makespan, only got jobs with processing times that are d per cent higher than the bag specified. For this reason, its load is d per cent higher and Equation 4.18 becomes

\[
T_{ms} \leq \left(\frac{n^T t}{|W|} + (\ln(|W| - 1) + 1) \frac{1}{2} t_1\right) \frac{100 + d}{100}
\] (4.19)

Substituting this into the derivation of the approximation ratio in Theorem 4.1.4 leads to

\[
T_{ms} \leq \left(\frac{n^T t}{|W|} + (\ln(|W| - 1) + 1) \frac{1}{2} t_1\right) \frac{100 + d}{100} \\
\leq \left(\frac{3}{2} + \ln\left(\sqrt{|W| - 1}\right)\right) \frac{100 + d}{100} OPT
\]

\[\Box\]

**Running Time**

In order to derive the running time of the Vector-Scheduler, two Lemmas are proven first.

**Lemma 4.1.6.** The repeat-loop is executed at most \(\frac{1^T t}{|B|} + 1\) times.

**Proof.** In the worst case, the error to reduce is as large as possible and in each iteration of the repeat-loop, only one job—namely the smallest—is added. By using Lemma 4.1.1 and Proposition 4.3, it can be concluded that in the worst case \(\frac{1^T t}{|B|}\) jobs are added without the slack load becomes negative. The Vector-Scheduler adds one extra job when this does not enlarge the error, which concludes the prove. \[\Box\]

**Lemma 4.1.7.** Finding the closest vector has a time complexity of \(O(|B| + 2^{|B|})\).
Proof. Finding the vector that induces the greatest load smaller or equal to the given slack load takes $O(\log_2 2^{|B|}) = O(|B|)$ time. However, finding the two vectors that both can be composed of the unscheduled jobs can take up to $2^{|B|}$ time, which concludes the proof. \hfill \Box

**Theorem 4.1.8.** The Vector-Scheduler has a time complexity of $O\left(2^{|B|}\left(|B| + \frac{T_t}{|B|}|W|\right)\right)$.

**Proof.** Constructing the list can be done in $O\left(2^{|B|} \log_2 \left(2^{|B|}\right)\right) = O\left(2^{|B|}|B|\right)$ time. $\Theta\left(2^{|B|}\right)$ to produce all the vectors—recall that there are $2^{|B|}$ vectors—and $O\left(2^{|B|}|B|\right)$ to sort them. Determining the total load in $B$ costs $\Theta\left(|B|\right)$ time. Executing the operations on the lines 4 to 17 takes $O\left(|B|\right)$ time. The for-loop from line 21 to 24 costs $O\left(|B|\right)$ time and from Lemma 4.1.7 follows that GetClosest can be done in $O\left(|B| + 2^{|B|}\right)$ time. Now, by using Lemma 4.1.6 it can be concluded that the repeat loop costs $O\left(\left(\frac{T_t}{|B|} + 1\right) 2^{|B|}\right) = O\left(\frac{T_t}{|B|} 2^{|B|}\right)$ time. Therefore, the total running time of the loop on the lines 3 to 27 equals

$$
O\left(\left(|W| - 1\right)\left(|B| + (1) 2^{|B|}\right)\right)
= O\left(\frac{T_t}{|B|}|W|2^{|B|}\right)
$$

(4.20)

Note that the fraction is strictly greater than 1. Finally, assigning the remaining load to the last worker takes $O(|B|)$ time. This boils down to a total running time of

$$
O\left(2^{|B|}|B| + |B| + \frac{T_t}{|B|}|W|2^{|B|}\right)
\Rightarrow O\left(2^{|B|}\left(|B| + \frac{T_t}{|B|}|W|\right)\right)
$$

\hfill \Box
Chapter 5

The Scan Analysis Algorithms

As part of this graduation assignment the Scan Analysis Algorithms (SAAs) that are used to evaluate the conditions of printing elements are implemented in the developed Scan Analysis Framework (SAF). This analysis involves three SAAs, namely Find Markers, Vertical Averages and Find Stripes. Each of them are implemented in an Algorithm Processor (AP). Typically, the APs are pipelined as illustrated in Figure 5.1.

![Figure 5.1: Processing pipeline to evaluate printing element functionality.](image-url)

The related test charts consist of patterns and each pattern has two types of subpatterns. The first type is a collection of stripe shaped primitives, these must be processed by the Vertical Averages AP and then by the Find Stripes AP. The second type consist of four position markers (four primitives), these must be processed by the Find Markers AP. Each stripe is oriented vertically, one pixel thick and is totally printed by one printing element. The absence of a stripe indicates that the corresponding printing element does not work, and if it has a different position, it indicates that the printing element had shot at an angle.

The Vertical Averages AP requests a snippet from the image descriptor that includes all the stripes and determines the average pixel value for each column in the snippet. Recall that a snippet only includes one color channel. The result is a vector, of which the size is equal to the width of the snippet, that is used by the Find Stripes AP to determine the positions of the printed stripes. It then compares these measured positions with the defined positions to deduce the condition of each printing element. The positions of the primitives, and thus of the stripes, are defined relative to the upper left corner of the sheet. However, several inaccuracies in the print process cause positions in the image to be slightly shifted. Examples of causes include: the sheet was rotated while being printed; the sheet was rotated while being scanned; the sheet was shrunk by heat while being transported to the scanner. These disturbances are identified by examining the differences in the measured and defined positions of the position markers. The positions of the stripe primitives are corrected accordingly. The markers are shaped such that they are easy to find and can be positioned accurately, which is done by the
Find Markers AP.

Matlab implementations of these algorithms are available, which are used as a starting point as well as functional reference, i.e. to verify the obtained SAAs. The SAAs are implemented in APs according to the design flow as presented in Figure 5.2.

First, the Matlab implementations are ported to naive C++ AP implementations that run correctly in the SAF. The Matlab implementations are designed to realize the required functionality rather than obtaining high performances. Therefore, in the second stage, it is investigated if the same functionality can be achieved with less computational intensive (sub) algorithms. In the last three stages it is attempted to let the SAAs run faster by performing several optimizations. Dataflow optimizations are intended to eliminate data transfers and reduce data storage. Albeit space is not a major concern, the transfers and storage operations require time and, therefore, these optimizations implicitly decrease running time. Control flow optimizations are focussed on arranging the order in which data is processed, aiming at improving the regularity of accesses and data locality, while preserving dependencies. Finally, the instruction optimizations are concerned with the used instructions. This involves simplifications of formulas when possible and varying the instruction types, intending to take advantage of the CPU’s instruction pipeline.

First, the remainder of this section describes the Matlab implementations and the complicated optimizations for each SAA. In addition, it describes an algorithm that is used by the SAF to detect the left, right and top edge of a sheet in an image. Finally, the improvements of the optimizations are highlighted.

5.1 Find Markers

The provided Matlab implementation starts with finding a rough position of the first marker—that is listed in the job—in a relatively big snippet, resulting in an initial offset. Then, all markers (including the first) are located accurately by using relatively small snippets. Locating a marker in a snippet is done in four steps. First, a kernel that mimics the shape of the marker is created. Second, this kernel is convolved over the snippet.
Third, the lowest value in this convolved snippet is determined, which is the location where the kernel fits best and should be the position of the marker. Finally, for the sake of robustness, the inverse of the kernel is used to verify that the marker is present at the found location.

**Algorithmic Optimizations**  The algorithmic optimizations are focussed on the convolutions, since these virtually determine the processing time. The following table presents averaged times of a profiling report, from which becomes vivid that the five convolutions account for an exceeding 99 per cent in the total processing time.

<table>
<thead>
<tr>
<th>Rough position</th>
<th>Fine positions</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>3881000</td>
<td>821000</td>
<td>822000</td>
</tr>
</tbody>
</table>

Table 5.1: Averaged processing times of Find Marker in $\mu$s.

The use of separable kernels and convolutions underlie the algorithmic optimizations, as these are considerably less computationally intensive than the original convolutions. The convolution in the first stage is completely replaced by a separable convolution. The convolution in the second stage is preceded by a separable convolution to estimate the position, which is then refined using a much smaller snippet. This 2D convolution cannot totally be replaced since the marker is not separable and for this reason the separable convolution is performed with an approximation kernel—which leads to less accurate positions.

**Dataflow Optimizations**  The Find Minimum function only evaluates the image that is convolved with a full kernel, thus the borders are not used. To save data transfers as well as operations, these borders are not computed (and stored). This also reduces the complexity of the control flow, as convolution with partial kernels at the borders are eliminated. The second dataflow optimization is the merge of the second (vertical) 1D convolution and the Find Minimum function.

**Instruction Optimizations**  Intel’s SSE intrinsic vector instructions are used to parallelize several operations in the 1D convolutions.

### 5.2 Vertical Averages

The provided Matlab implementation duplicates the whole image and splits it into four color planes (cyan, magenta, yellow and black). Each plane is then filtered with a zero-phase, vertical averaging filter in order to obtain a better image to measure positions horizontally. Then, this filtered image is used to gather the average vectors by simply copying pixels.

The initial implementation of the SAA does not preprocess the whole image, i.e. split into four color planes and filter each of them. Moreover, the zero-phase filtering—which are two 1D convolutions—are eliminated. Instead, the SAA requests a snippet from the...
image descriptor that includes the target stripes and uses a formula to calculate the average pixel value of each column. This greatly decreases both the number of operations as well as the amount of data that is stored and transferred. (The sum of the used areas per color channel is about five per cent of the total image.)

5.3 Find Stripes

The provided Matlab implementation establishes the condition of each printing element by comparing the positions on the printed stripes with the position of the defined stripes. The positions of the printed stripes are deduced from the average vector that is computed by a Vertical Averages AP. This is done by an operation that has a complexity equal to a 1D convolution. The defined positions are corrected. These corrections are based on the measured and defined locations of the position markers as well as the position of the average vector.

No further optimization opportunities were identified, which was not essential since it already has a short running time compared to the Find Markers AP.

5.4 Find Edges

The Matlab implementation locates the edges in roughly two steps. First, the average blue pixel value of each row and column is computed, resulting in one row and one column vector. Since the environmental view is rather dark and the sheet is light, the vectors include a rapid variation in value at the position of an edge. These changes are used in the second step to locate the position of an edge. In case of two images, the left and top edge are located in the left image and the right edge is located in the right image.

Algorithmic Optimizations Calculating the average pixel values of each row and column is a relatively heavy computational task—the typical size of one scan is 4600 x 5200 pixels. The algorithmic optimized SAA first estimates the position of an edge by locating the changes in one row or column of the image. Then, this position is refined by using a small average vector.

5.5 Performance Evaluation

To establish the impact of the performance improvements a number of tests were conducted. These tests were carried out on an Intel i5-650 based PC with 4GB main memory and Microsoft Windows 7. The Intel Speed-Step and Turbo functionalities of this CPU both were disabled, aiming at realizing constant performance rather than best performance on energy and time. Since the CPU has two physical cores, one worker was included.

The resulting running time per SAA is given in Table 5.2. It can be observed that processing time of the Find Markers SAA is improved by an exceeding 250 times. Also the Find Edge algorithm is more than 188 times quicker.
### Table 5.2: Processing times in µs.

<table>
<thead>
<tr>
<th></th>
<th>Find Markers</th>
<th>Vertical Averages</th>
<th>Find Stripes</th>
<th>Find Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td>7185000</td>
<td>-</td>
<td>1000</td>
<td>528000</td>
</tr>
<tr>
<td>Algorithmic</td>
<td>1191000</td>
<td>85</td>
<td>-</td>
<td>14000</td>
</tr>
<tr>
<td>Dataflow</td>
<td>93000</td>
<td>-</td>
<td>490</td>
<td>-</td>
</tr>
<tr>
<td>Control flow</td>
<td>56000</td>
<td>-</td>
<td>-</td>
<td>2800</td>
</tr>
<tr>
<td>Instruction</td>
<td>28000</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
Chapter 6

Analysis

This section describes four experiments that were carried out and discusses the results of them. Three experiments were intended to assess individual components of the Scan Analysis Framework (SAF). The first experiment established the overhead induced by the SAF; the second determined how well the processing times are predicted and the third investigated the quality of the processing schedules. The fourth experiment was conducted to test the entire SAF.

Two test benches were developed to perform the experiments. The first test bench was designed to test properties of the SAF and was executed on a machine that included two Intel Xeon eight-core processors and 32GB main memory. The second test bench was designed to test the Vector-Scheduler and was executed on a machine that included an Intel i5 processor and 4GB main memory.

6.1 Overhead of the Scan Analysis Framework

This experiment consisted of two parts. The first part investigated how the Job Managers’ components scale with the number of workers, bags and jobs. The second part established the overhead induced by an Algorithm Processor (AP).

6.1.1 Methods

In the first part, the SAF-test bench created test chart descriptors and let them be processed by the Job Manager. There were two types of descriptors. The first type led to 32 jobs that were spread out evenly over a variable number of bags, namely 2 to 24. The second type included a variable number of jobs that was evenly distributed over five bags, the number of jobs was selected from the set \{10, 20, 50, 100, 200, 500, 1000, 2000, 5000, 10000\}. Each created descriptor was processed by the Job Manager for 2, 4, 8, 16, 32 and 64 workers. Moreover, each test was performed 20 times in order to get an average running time.

In the second part, the SAF-test bench let the “printing element” test chart be processed by SAFs that included 2, 4, 6, 8 and 10 workers. The registered start and stop times were used to determine the overhead. Note that this time includes the overhead of both the
AP and the IPF. This experiment is bounded to 10 workers, since the machine used had no more cores available.

Note that the regressor is not evaluated. This is not needed as this component is ran during start-up time and has more than enough time (minutes) to perform its computation (which is in the range of milliseconds, depending on the number and sizes of the training set files).

6.1.2 Results and Discussion

The results of the first part are visualized in Figure 6.1. The figures on the left show the running times as a function of the number of bags, the figures on the right as a function of the number of jobs. Each figure has a graph for 2, 4, 8, 16, 32 and 64 workers. It is apparent that the number of workers has little influence on the running time of the components. The registered deviations are probably caused by external influences and are not related to the number of workers. The running time of the Job Chain Creator has a linear relationship to the number of jobs. This can be explained by the fact that it must create each job. The rapid growth the graphs exposes is most likely caused by reading the Test Chart Descriptor line by line. This growth has also a linear relation with the number of bags, though it does not strongly influence the running times. The running time of the Scheduler has an exponential relation with the number of bags. This corresponds with the size of the tree. (The tree-size has an exponential relation with the number of bags.) The time is slightly affected by the number of jobs, which is most likely caused by an increase in repetitions of step three. The Dispatcher shows similar behavior as the Job Chain Creator. The running time has a linear relation with the number of jobs. This can be explained by the fact that it loads each individual job into a queue. Moreover, like the Job Manger, it is slightly effected by the number of bags. Finally, it can be observed that the accumulated running times are virtually equal to the running time of the Job Chain Creator when there is a small number of bags. When there are 13 bags or more, the Scheduler starts playing a major role in the running time.

The results of the second part are visualized in Figure 6.2. It shows that 75 per cent of the registered overhead times were lower than 8µs and that the average overhead time was about 9µs. This rather higher average can be explained by the fact that 25 per cent of the registered times was between 8 and 1001µs. Further research pointed out that only a few high overheads were registered, which are most likely caused by external factors. In general, it can be concluded that these times are negligible compared to the processing times of jobs, see Chapter 5. Moreover, the overhead induced by the AP is even lower since the considered times include the IPF overhead.

![Figure 6.2: Registered overheads of AP in µs.](image-url)
Figure 6.1: Running times of Scan Analysis Framework’s components for 2, 4, 8, 16, 32 and 64 workers as a function of the number of bags (left column) and as a function of the number of jobs (right column).
6.2 Scheduler

To assess the quality of the processing schedules, a collection of bag sets were generated and each bag set was scheduled by the Vector-Scheduler as well as by the widely known LPT-Scheduler. The quality is expressed as a competitive factor between the two resulting makespans. More precisely, consider a bag set \( B \), then the competitive factor \( q \) is given as

\[
q = \frac{T^{VS}}{T^{LPT}}
\]  

(6.1)

where \( T^{VS} \) and \( T^{LPT} \) denote the makespans of the Vector and LPT scheduler respectively. This experiment is extended with the evaluation of the running times of both schedulers.

6.2.1 Methods

Two types of bag sets were used. The first type attempts at resembling bag sets that are extracted from test chart descriptors. Although this already reveals how well the scheduler will work in the SAF, this experiment is extended to assess how well it deals with more general bags, which is the second type.

The processing times of the first bag sets were generated as follows. The first bag got a processing time in the range \([1000, 8000]\). The processing time of each other bag equals the processing time of the previous bag plus an extra load in the range \([100, 600]\). Both the first and additional loads were generated using a Normal Distribution. The number of jobs in a bag is also generated using a Normal Distribution and was chosen from the range \([10, 200]\).

The processing times of the general bag sets were generated with a bounded Pareto distribution. This is a hyperexponential distribution and hence the sets typically have a few bags with large jobs that account for a major part of the total load. The distribution is characterized by the parameters \( k \), \( p \) and \( \alpha \). The \( k \) and \( p \) determine the range of the distribution, where \( k \) denotes the lower bound and \( p \) the higher bound. In the experiment, \( k \) and \( p \) were set to 10 and \( 10^6 \) respectively, i.e. all jobs have a load in the range \([10, 10^6]\). The \( \alpha \) determines the shape of the distribution and has a domain from 0 to 2. A lower \( \alpha \) results in a higher degree of variation in the generated samples. Harchol-Balter [5] empirically determined that an \( \alpha \) equal to 1 results in a job set that coincides with the workload in a network of Unix workstations. Therefore, in this experiment, the \( \alpha \) was set to 1. For each bag, the number of jobs in it was generated using a Normal Distribution. The lowest number of jobs was 1 and the highest number was 100.

To compare the time efficiency of the Vector-Scheduler to the LPT-Scheduler, several job sets were generated. Each set was scheduled by both schedulers for a set of 8 workers. The job sets were generated similar to the general job sets described above. However, the number of bags was in the range \([2, 15]\) and the number of jobs was selected from the set \(\{10, 20, 50, 100, 200, 500, 1000, 2000, 5000, 10000\}\). Each possible bag set was generated 500 times and each set was scheduled by both workers, leading to an average competitive
factor, which was defined as

\[ q = \frac{t_{VS}}{t_{LPT}} \]  

(6.2)

where \( t_{VS} \) and \( t_{LPT} \) denote the running times of the Vector and LPT scheduler respectively.

### 6.2.2 Results and Discussion

The results of the experiments where the first bag sets were used are visualized in Figure 6.4. The box-and-whisker plots reveal that the majority of the average competitive factors are equal to 1 and in some cases below 1. The latter indicates that the Vector-Scheduler had produced a better scheduler than the LPT-Scheduler, i.e. a lower makespan. Moreover, there seems to be a relation between the deviation and the number of workers. The deviations increase when the number of workers grows. A reasonable explanation would be that a higher number of workers results in more scheduling possibilities and hence allows for greater deviations. In general, these variations are in favor of the Vector-Scheduler. Another observation would be that the deviation decreases with the number of bags. This is probably caused by the fact that the schedulers have more opportunities to perform the balancing. In case of the LPT-Scheduler this means that it can correct bad scheduling decisions. Overall, it seems that the Vector-Scheduler performs fairly similar to LPT-Scheduler.

The results of the experiments using the generalized job sets are visualized in Figure 6.5. The box-and-whisker plots suggest that the competitive factor is on average equal to 1. Furthermore, when the number of workers increases, the deviations grow. This is probably caused by the same reasons as explained in the previous paragraph. In general these variations are in favor of the LPT-scheduler.

The registered competitive factors between the running times of the Vector and LPT Scheduler are visualized in Figure 6.3 using a contour graph. The contour lines have intervals of 1 (note that competitive factors have no units), and the contour line at the bottom represents 1. Therefore, for each bag set below this line, the LPT-Scheduler is quicker than the Vector-Scheduler and for all job sets above this line the Vector-Scheduler is quicker. As expected, the Vector-Scheduler performs especially well when the number of jobs is great and the number of bags is small. For instance, in the upper left corner a speed-up of more than 550 times is registered.
Figure 6.3: Competitive factors of the running times in a contour graph.
Figure 6.4: Competitive factors of bag sets based on expected test charts. Above 1 means that the Vector-Scheduler is better than the LPT-scheduler.
Figure 6.5: Competitive factors of general bag sets. Above 1 means that the Vector-Scheduler is better than the LPT-scheduler.
6.3 Processing Time Prediction

These experiments were focussed on investigating the quality of the processing time predictions, as well as on determining the effect of the training set size.

6.3.1 Methods

The quality is obtained by letting the SAF analyze the printing element test chart 500 times and compare the predicted processing times to the (actual) registered times. Note, that the training set is extended at the end of each test.

6.3.2 Results and Discussion

Figure 6.6 shows the result of the experiment. The left figures present histograms of the percentage deviation, the figures on the right show the absolute percentage deviation as a function of the training set size. The values in the latter are filtered with a running average filter that has 100 entries.

It can be observed that the predictions for the Find Marker are more accurate than the other two. This is probably due to its significantly longer processing time, masking disturbances caused by external factors. The graph on the right shows that the predictions become better when the training set grows. The predictions of the Vertical Averages may be good on average but exhibit large errors in general. This is most likely caused by external disturbances, which have a great influence on the processing times, as these are quite low. This issue may be exacerbated further by the fact that multiple instances of the Vertical Average SAA request snippets from the Image Descriptor, causing memory collisions. This explanation tallies with the graphs of both the Find Marker and Find Stripes SAAs. They both do not request snippets and have larger processing times. The Find Stripes SAA has many predictions that deviate less than 5 per cent. The graph on the right reveals that the predictions are below 5 per cent from the start.
(a) Find Markers.
(b) Find Markers.
(c) Vertical Averages.
(d) Vertical Averages.
(e) Find Stripes.
(f) Find Stripes.

Figure 6.6: Running times of Scan Analysis Framework components as a function of the number of bags (left column) and as a function of the number of jobs (right column).
6.4 A Total Test

In this last experiment, it was determined what the processing times were of the Matlab implementation and SAF implementation to evaluate a printing element test chart. The resulting times are compared to establish the gained performance improvement.

6.4.1 Methods

As there is only one test chart (and corresponding descriptor) was available, only this chart was used in the experiment. It is evaluated 100 times by both implementations.

6.4.2 Results and Discussion

Table 6.1 shows that the SAF is an exceeding 14 times faster. It is also worth mentioning that the test chart had led to two bags, confirming the expectation that only a few bags are created.

<table>
<thead>
<tr>
<th>Matlab</th>
<th>SAF</th>
</tr>
</thead>
<tbody>
<tr>
<td>3708600</td>
<td>263432</td>
</tr>
</tbody>
</table>

Table 6.1: Averaged processing times in $\mu$s for evaluating a printing elements test chart.
Chapter 7

Conclusion

This project was concerned with the processing of arbitrary test charts. Océ printers use these charts to monitor their conditions. These charts consist of patterns and each pattern must be analyzed by a set of Scan Analysis Algorithms (SAAs). A Scan Analysis Framework (SAF) has been developed that enables the existing Image Processing Framework (IPF) to process these test charts on a multicore processing system.

The SAF includes a set of identical workers which perform the analysis. A worker contains a set of SAAs which can easily be added and removed, also the number of included workers can be adjusted. The SAF extracts patterns from a test chart and creates for each pattern a job. The processing time of a job is predicted based on statistics of previously processed jobs. This self-learning mechanism ensures that the SAF adapts itself according to the characteristics of the system it runs on. Therefore, any SAA can be added without any benchmarking in advance. It has been observed that many jobs have similar processing times. This observation is exploited by packing jobs with equal processing times into a single bag. To the best knowledge of the author, the resulting problem has not been considered earlier. Therefore, this project also led to the introduction of the Bag Scheduling Problem, in which a set of jobs that are packed into bags must be mapped onto a set of identical workers. To solve this new type of problem, the Vector-Scheduler has been proposed, which is used in the SAF to balance the jobs over the workers. The results of the individual analysis are merged into a single result.

The scope of the project is expanded with the implementation of three SAAs in the SAF. These algorithms are used to evaluate the condition of printing elements. The SAAs successfully run in the framework and meet the functional requirements. Moreover, the SAAs are optimized, leading to speed-ups of an exceeding 250 times. Below, the main results of this project are summarized

- A generic Scan Analysis Framework
- A self-learning mechanism to predict processing times of jobs
- The introduction of the Bag Scheduling Problem
- The Vector-Scheduler that solves this problem
- Three SAAs that run successfully in the SAF
Several experiments were conducted to investigate multiple performance aspects of the SAF. They reveal that the overhead induced by the workers is negligible compared to the processing it performs. The Vector-Scheduler is designed to schedule a large number of jobs that packed into a few bags. Experiments show that the number of bags indeed highly affect the running time. The prediction of the processing times is fairly accurate and becomes better when the number of processed jobs increases. Measured inaccuracies are most likely caused by external influences.

The current SAF can only deal with identical workers that are executed on a homogeneous set of cores. Future work may concentrate on the use of non-identical workers that are executed on a heterogeneous processing system. Deriving jobs for a test charts start when both the scan data is available, prolonging the total execution time. Deriving the jobs while the test chart is printed and scanned would be an improvement of the current design. A last direction of pursuit suggested here is the development of an alternative method to perform step three of the Vector-Scheduler, as this step virtually determines the running time of the Vector-Scheduler.

In order to conclude this thesis, it must be pointed out that all requirements stated in the problem description are realized. The developed SAF facilitates the processing of test charts on the existing IPF, which dramatically decreases the design time and costs. Equally important is the added flexibility, that not only facilitates the processing of arbitrary test charts, but also offers the opportunity to extend the usage of the SAF.
Appendix A

Profiler Tool

The Profiler Tool has been developed to gain insight in the processing. Figure A.1 presents an overview of the tool. It visualizes for each worker what jobs it has processed (by using job IDs) as well as the start and stop times of them. The start time is the time right before the Algorithm Processor starts the SAA component, the stop time is the time right after this SAA has finished its computation. These times are showed when the mouse pointer is on the job (roll-over function), as is illustrated in Figure A.2.2.

Figure A.1: Overview of Profiler Tool.
Figure A.2: Some features of the Profiler Tool.
Bibliography


Index

Algorithm Descriptor, 9
Algorithm Processors, 7
AP, 7, 13
Bag-of-Tasks, 15
Bin Scheduling Problem, 15
Broker, 7, 14
DAG, 4, 15
Data regression, 12
Directed Acyclic Graph, 4, 15
Dispatcher, 11
Dot product, 12
Fork-join pipeline, 7
Heuristic algorithm, 16
Image Descriptor, 9
Image Processing Framework, 7
Inner product, 12
IPF, 7
Job Accumulator, 14
Job Chain Creator, 9
Job Manager, 8
Job Scheduling, 15
Job Scheduling Problem, 15
Kahn Process Network, 15
Key-value, 10, 12
KPN, 15
Moore-Penrose pseudoinverse, 12
Multivariate data regression, 12
OLS, 12
Ordinary Least Squares, 12
Primitive Descriptor, 9
Processing Parameter, 7
Processing Request, 7
Processing Result, 8, 14
Processors, 3
Push-Pop, 5
Radix-tree, 12
Regressor, 12
Replicated Parallel Pipeline, 3
SAA, 4, 7
SAF, 7
Scan Analysis Algorithm, 4, 7
Scan Analysis Framework, 7
Scheduler, 11
SDF, 15
Snippet, 9
Synchronous Data Flow, 15
Task Graph, 15
Task Scheduling Problem, 15
Test Chart Descriptor, 5
Training Set, 12
Training Set File, 8, 12, 14
Trie, 12
Type A, 4
Type B, 4
Type C, 4
Vector-Scheduler, 16
Worker, 13