MASTER

Workload distribution in heterogeneous SMT assembly lines

Snoeren, J.P.H.

Award date:
2016

Eindhoven University of Technology
Department of Mathematics and Computer Science

Workload Distribution in Heterogeneous SMT Assembly Lines

Master's Thesis

J.P.H. Snoeren

Supervisors:
prof. dr. M.T. (Mark) de Berg (TU/e)
dr. ir. D.H.P. (Dirk) Gerrits (Kulicke & Soffa)

Eindhoven, August 2016

Abstract

Kulicke & Soffa applies Surface Mount Technology to manufacture printed circuit boards (PCBs). They use two different types of machines to outfit these PCBs with components. We study the problem of distributing the workload among these different types of machines in order to minimize overall cycle time. Our approach consists of a simulated annealing algorithm, accompanied by a model of the machines used to estimate the cycle time of a particular solution. Various variants of simulated annealing and different ways to incorporate all operations of the machines in the model are discussed. The best variant of our simulated annealing algorithm yields an average improvement of 7.99% compared to the current system used to balance the workload. Taking the best result encountered in our experiments, we obtain an average improvement of 9.99%. We suggest improvements for future work that we believe to further improve the performance of our algorithms.

Contents

Chapter 1

Introduction

Kulicke and Soffa (K&S) is a global leader in the design and manufacture of semiconductor, LED, and electronic assembly equipment. Recently K&S has expanded its expertise to advanced Surface Mount Technology (SMT) by acquiring Assembléon Netherlands B.V. SMT is a method used for producing electronic circuits with components mounted on the surface of printed circuit boards (PCBs). K&S produces and sells machines applying SMT, which allow customers to assemble PCBs by outfitting them with components.

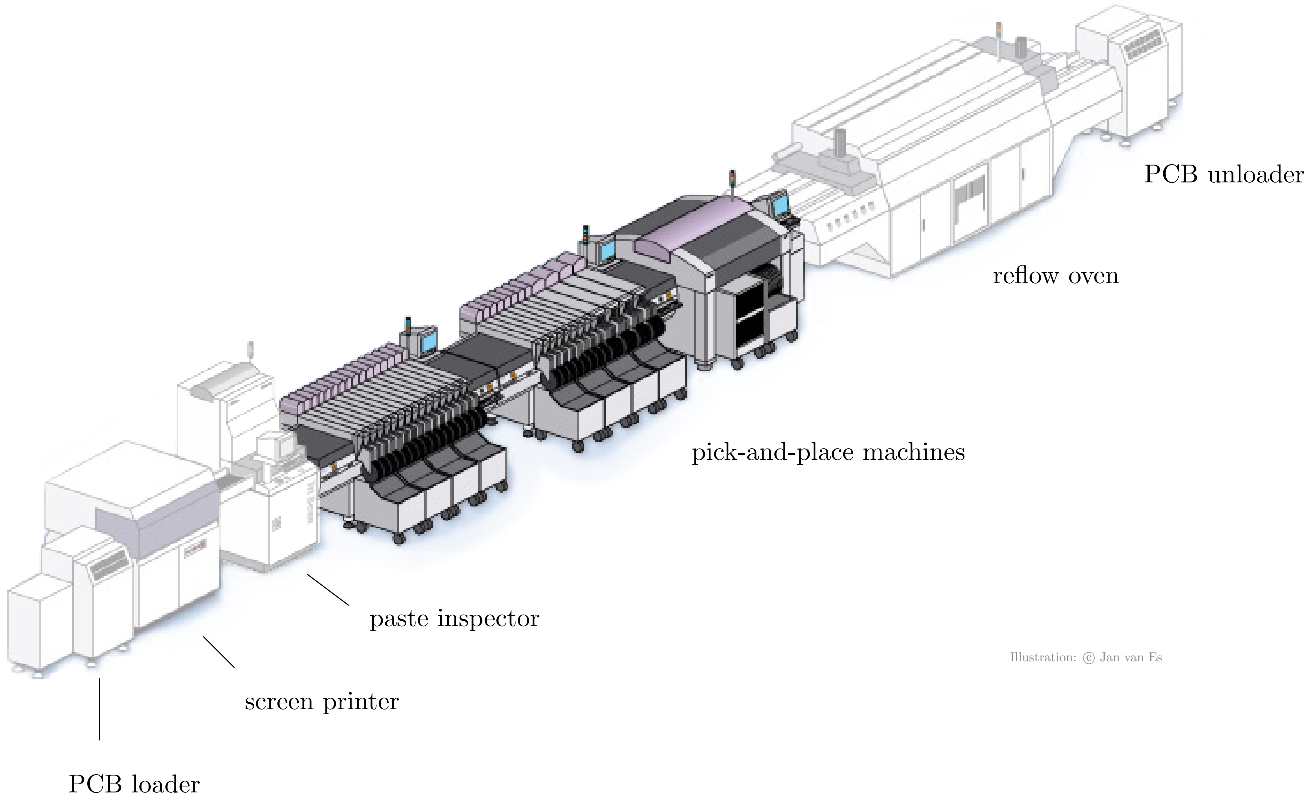


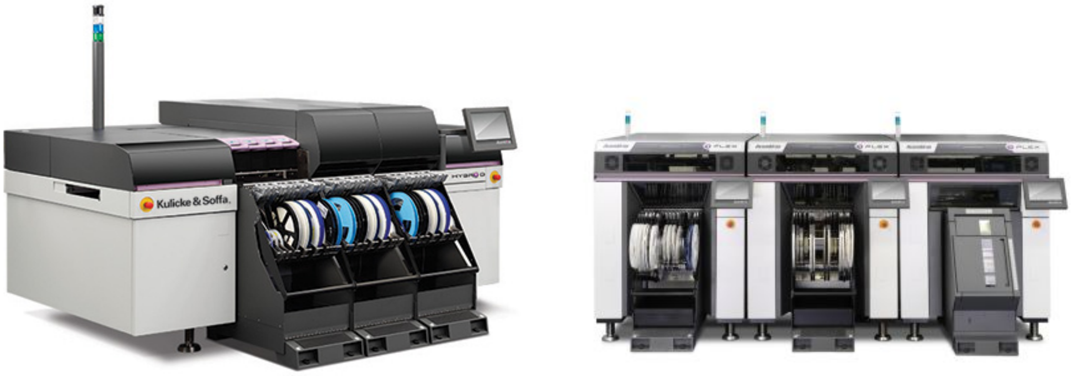
Illustration: © Jan van Es

Figure 1.1: An example assembly line. In this thesis we focus on the pick-and-place machines mounting components on the PCBs.

1.1 High-level problem

The PCBs are assembled using an assembly line, consisting of a number of machines. An example of such an assembly line is depicted in Figure 1.1. Once the PCB has passed through all machines, all components have been placed and the PCB is completely assembled. Part of the assembly line consists of pick-and-place machines, which are assigned the task of placing a number of components on the PCB. The other machines are used to solder the components on the PCB. The pick-and-place machines sold by K&S come in two different flavours: the iX machine and the iFlex machine, as displayed in Figure 1.2. Each of these machines has its own characteristics, which are covered in great detail in Sections 2.1.1 and 2.1.2. Within an assembly line for PCBs, the pick-and-place machines used may be iX machines, iFlex machines or machines from other manufacturers. Although it is not required to deploy both iX machines and iFlex machines, doing so allows the customer to make the best use of both machines by emphasizing each of the machines' strengths. A customer determines which combination of machines is required based on the specific PCBs they want to produce. In this thesis, we focus on the specific case where the pick-and-place machines within an assembly line consist of a combination of an iX machine and an iFlex machine.

The aim is to determine a schedule for the pick-and-place machines specifying when and how each component should be placed such that the total cycle time is minimized. The measure of cycle time is explained in detail in Section 2.2.2. K&S has software in place to optimize the cycle time of both the iX and iFlex machine, determining which placement robot (within such a machine) places which component. Since each of these machines has its own characteristics, the natural question arising is how the workload should be distributed among the two different types of machines in order to minimize the cycle time. In other words: which components should be placed using the iX machine and which ones using the iFlex machine?



(a) An iX machine with three segments. (b) An iFlex machine with three segments.

Figure 1.2: The two types of machines manufactured by K&S to allow picking and placing components in the assembly line.

To balance the workload between the two different machines, K&S's customers use an external software product: Valor MSS Process Preparation. This external software is quite generic. Although capable of dealing with pick-and-place machines of all vendors, it is not tailored toward the characteristics of the specific machines. By exploiting the specific characteristics of K&S's machines, we aim to improve upon this external system and increase the efficiency of the assembly line by determining a partitioning of the components over the two types of machines yielding a lower cycle time than Valor MSS Process Preparation.

1.2 Related work

The problem under study is a variant of the well-known and well-studied load balancing or machine scheduling problem. In this section we discuss literature both from the academic community, generally more focused on simpler versions of the problem but providing guarantees on quality and running time, and literature studying problems arising from industry, focusing more on our specific domain of SMT and assembly lines.

Scheduling problems have been studied extensively in the past. For a survey on scheduling algorithms for different variants we refer to Graham, Lawler, Lenstra, and Rinnooy Kan (1979), and more recently the survey by Potts and Strusevich (2009). In particular, in the notation of Graham et al. (1979) we are mainly interested in the scheduling problem $R \mid\mid C_{\max}$. The job scheduling problem asks for assigning n tasks to one of m processors. There are a number of different variants of the scheduling problem. The variant with identical parallel machines considers tasks with equal processing times for each machine. Another variant considers uniform machines, where the machines all operate with a certain speed factor. For the $R \mid\mid C_{\max}$ problem, the R in Graham's notation indicates that the m processors are unrelated. That is, the processing time incurred by assigning task i to processor j_1 is unrelated to the time incurred by assigning task i to processor j_2. The $C_{\max}$ in Graham's notation indicates that the objective function is the maximum over the completion times of all tasks. Since the iX and iFlex machine have their own characteristics, the variant concerning unrelated parallel machines is of importance to us. Note that although the machines are placed in series, each of the components is only placed by one of them. Hence, from a mathematical point of view we consider the machines in parallel.

The $R \mid\mid C_{\max}$ problem has a rich history of research. Horowitz and Sahni (1976) provide an approximation scheme of time complexity $O(n^{2m}/\varepsilon)$ for approximation factor $1 + \varepsilon$. Davis and Jaffe (1981) provide a polynomial-time approximation algorithm with approximation ratio $2\sqrt{m}$ and an algorithm with running time $O(m^m + mn \log n)$ and approximation ratio $1.5\sqrt{m}$. Potts (1985) devised a heuristic method using linear programming yielding an approximation algorithm with approximation ratio 2 and running time polynomial in n, but exponential in m. For the special case of $m = 2$ they provide a linear-time 3/2-approximation. Potts extends his work with Hariri (Hariri & Potts, 1991) to focus on other heuristics as well, leading to algorithms with rather unsatisfactory quality. Lenstra, Shmoys, and Tardos (1990) provide a polynomial-time 2-approximation based on linear programming relaxation and prove that no polynomial-time approximation algorithm with approximation factor less than 3/2 exists unless P = NP. Van de Velde (1993) use Lagrangian duality to obtain a branch-and-bound algorithm solving $R \mid\mid C_{\max}$ optimally, as well as a heuristic algorithm applying local search. Martello, Soumis, and Toth (1997) improve upon the ideas of Van de Velde (1993) to obtain a faster algorithm. Jansen and Porkolab (2001) design a PTAS with running time at most $n(m/\varepsilon)^{O(m)}$. More recently, algorithms have been developed and evaluated for extensions of the $R \mid\mid C_{\max}$ problem such as a flow shop environment (Low (2005)) and setup times (Kim, Kim, Jang, and Chen (2002); Ying, Lee, and Lin (2012)) and combinations of them.

Taking a look at extensions that are useful to us, we examine vector scheduling, which is the multidimensional variant of $P \mid\mid C_{\max}$, where the P indicates that the processors are identical. That is, each task has a multidimensional processing time, where d is the number of dimensions. In our case, a dimension corresponds to a board type as we discuss in Chapter 2. Chekuri and Khanna (1999) study this problem and provide a PTAS and an $O(\ln^2 d)-$ approximation for d dimensions. Bansal, Caprara, and Sviridenko (2009) provide a polynomial-time algorithm with asymptotic approximation ratio arbitrarily close to $1 + \ln d$ for the dual problem vector bin packing. Bansal, Vredeveld, and van der Zwaan (2014) provide lower bounds on the running time of algorithms for the vector scheduling problem, and provide an algorithm with essentially optimal running time. To conclude our discussion of the vector scheduling problem, we mention Meyerson, Roytman, and Tagiku (2013) studying the online version of the problem.

Other problems are of interest to us as well: The multi-way number partitioning problem asks to partition n numbers over m bins such that the maximum sum of the numbers in each bin is minimized. Korf (2009) describes a number of methods to solve the multi-way number partitioning problem and evaluates the algorithms for a number of instances. These instances are relatively small compared to our input sizes: Whereas Korf (2009) performs experiments on inputs of size $n \leq 40$ and $m \leq 5$, our instances often have much larger values for m and n. Pop and Matei (2013) study the multidimensional version of two-way number partitioning and design a genetic algorithm to find good solutions.

Our problem is similar to $R \mid\mid C_{\max}$, since we are studying unrelated parallel machines and aim to minimize the cycle time (which is similar to the completion time). However, as described in great detail in Section 2.2 our problem has a number of additional constraints and complexities that need to be dealt with. These complexities contain among others constraints on the number of tasks of a certain type that can be placed onto one processor, and a processing time depending on the type of tasks already assigned to that processor. Although at this point it might not be clear to the reader how these additional complexities come about, it should be clear that the problem under study is more complex than the standard $R \mid\mid C_{\max}$ problem and new methods for solving the problem have to be developed. Note that removing these complexities yields a reduction from our problem to $R \mid\mid C_{\max}$, proving that it is NP-hard. In Section 2.2 we provide a more formal argument proving that the problem under study is NP-hard. Due to our problem's complexity, we should not expect to find an algorithm computing an optimal solution in polynomial time.

Apart from the mathematical problem of job scheduling, there is a vast body of literature devoted to more practical problems, often originating from industrial settings. Such a problem related to our setting is the well-studied assembly line design problem. The assembly line design problem asks to assign tasks to stations within an assembly line such that cycle time is minimized. For a survey of assembly line design problems and existing algorithms for different variants we refer to Rekiek, Dolgui, Delchambre, and Bratcu (2002). Although the assembly line design problem is similar, most research is concerned with dealing with precedence constraints and variants such as different types of processing times, or equipment selection. For these problems heuristics are developed which can not be used for our problem of workload distribution.

The remainder of this thesis is structured as follows. In Chapter 2 we provide a detailed description of the operations of both the iX and iFlex machine and provide models for them.

After the operations of the machines have been described in Section 2.1, we provide a formal problem description in Section 2.2. Chapter 3 describes the various algorithms designed and implemented to solve the problem. In Chapter 4 we describe both the accuracy of our model compared to K&S's software in Section 4.1 as well as the performance of our algorithms in Section 4.2. Finally, we conclude with a discussion on to what extent the problem has been solved and proposals for future work in Chapter 5.

Chapter 2

Models

In this chapter we describe the operations of K&S's pick-and-place machines. Due to the large amount of complexity of the machines, we have to construct a model of these operations, abstracting from a number of aspects of the machine. We describe where we abstract from, and discuss the impact of our abstractions. In Section 2.1 we describe the pick-and-place process, including the difference between the two different types of machines in Subsections 2.1.1 and 2.1.2. Subsequently, we provide a formal description of the problem in Section 2.2 and show its NP-hardness in Section 2.3.

2.1 Machine models

In this section we describe how a pick-and-place machine mounts components on the PCB. First we introduce the basic operations and terminology of such a machine. A schematic representation of part of a pick-and-place machine is depicted in Figure 2.1.

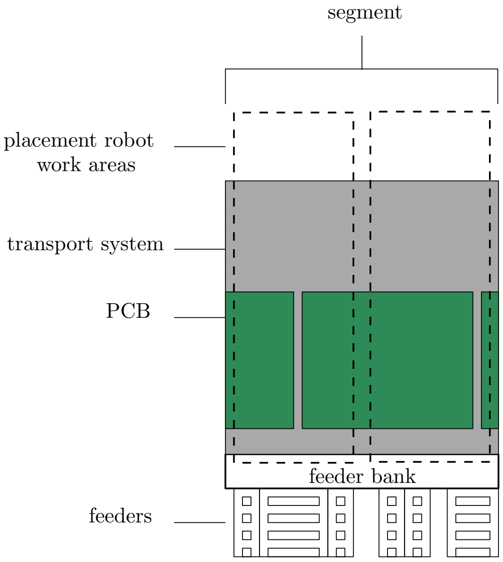


Figure 2.1: A segment of a pick-and-place machine.

A pick-and-place machine is centered around a *transport system*, transporting the PCBs from one end of the machine to the other. The actual placement of components is executed using *placement robots*, each consisting of a robot arm equipped with one or more *placement heads*. Before the components are mounted on the PCB, they are located in one of several *feeders*, as depicted in Figure 2.2. These are containers for components appearing in different forms such as tapes, trays and wafers. Each of the feeder types has a different width. Components have a certain type, which is referred to as a *part*. A feeder can only contain components of a single part, but contains multiple components of the same part. There are special types of feeders allowing for multiple parts per feeder, but we do not consider such feeders in our problem. A machine consists of a number of *segments*, indicating the boundaries of placement robots and feeders. Depending on the type of machine, such a segment contains either one or two *feeder banks*, containing the feeders for that segment.

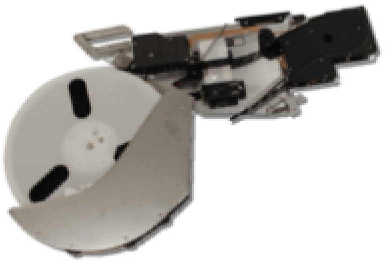


Figure 2.2: A tape feeder to be mounted on the feeder bank, containing components.

The basic pick-and-place process consists of a robot arm moving to the feeder bank, picking up a component from the feeders, moving the arm to the corresponding location on the PCB and mounting the component, after which the robot arm moves back to the feeder bank to pick up another component and repeat the process. The PCBs are not transported during placement of components. Only after all robots on the machine have finished picking and placing components, the PCBs are transported a certain distance along the transport system. Now that the PCBs have moved to another location, they can be reached by other placement robots assigned other feeders. Hence, components of different parts can be placed on the new PCB's location. This process continues until the PCB reaches the end of the machine and all components have been mounted.

The process described above is the basic pick-and-place process. However, there are a number of aspects deviating from simply picking and placing of components that have a large impact on the performance of the machines that we have to take into account as well. In what follows we discuss each of these separately, along with the impact they have on performance.

To prevent defects and misplaced components while placing a component, the machine needs to check the position of a component before placing and align the component. There are two different methods for detecting misalignment of a component. If the component is small enough, it can be checked for misalignment within the placement head itself. The placement head, depicted in Figure 2.3 contains a *laser alignment module*, which is able to detect how much the component needs to be realigned while the robot arm is moving. Using the laser alignment module effectively takes no time, since the action can usually be performed while the robot arm is moving. In our model, we assume that realignment using the laser alignment module can always be performed while the robot arm is traveling to the PCB. However, if the component is too large to fit within the opening of the placement head, alignment needs to be checked using a *camera*. Some placement robots can optionally be equipped with such a camera. Since the camera is attached to the static frame of the machine and not to the placement robot itself, camera usage requires the robot arm to move to the camera, increasing the cycle time. Since not all placement robots are equipped with a camera, not all components can be placed by all robots. There are other reasons for components being incompatible with certain placement robots. For the purpose of our model, it is sufficient to know which components are compatible with which placement robots without distinguishing the various reasons of incompatibility.

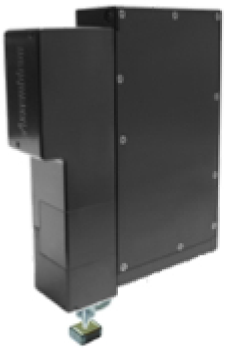


Figure 2.3: A placement head attached to the end of a robot arm, used to pick and place components. The placement head contains a laser alignment module.

While the PCB is moving through the machine, it may become misaligned by slight movements on the transport system, or by expansion and shrinkage due to temperature differences. To allow for alignment, a PCB is equipped with a number of *fiducials*. These are markers whose location can be measured to determine the exact position of the PCB. If the discrepancy between the expected and actual position of the PCB is too large, components may be placed incorrectly, leading to a malfunctioning PCB. The time required for measuring fiducials depends among others on the board type and type of placement robot, but the variations are relatively small compared to the total cycle time. Hence, in our model we assume that the time required for measuring fiducials is constant. Furthermore, we assume that each placement robot measures the same number of fiducials. Under this assumption omitting fiducials from our model does not influence our optimization algorithms.

Recall that each segment is equipped with one or two feeder banks, on which the feeders are mounted. Those feeders, containing components, are mounted on the feeder bank before the pick-and-place process begins. Feeders can only be mounted on the feeder bank in discrete positions. An interesting question, out of the scope of this thesis, is how we should determine a *feeder setup* specifying for each feeder a location on the machine. Since a feeder can only contain components of a single part, choosing a bad feeder setup may cause one robot to be overloaded yielding imbalance between the robots and increasing production time. In our model we do take the feeder setup into account to some extent, but refrain from computing an optimal feeder setup.

Picking up a component requires the placement head on a robot arm to be equipped with a certain *toolbit* (see Figure 2.4). Since not all components are compatible with all toolbits, it might be necessary to change the toolbit on the placement head during assembly of the

Figure 2.4: A number of different toolbits. Each component needs a compatible toolbit to be picked.

PCB. Whether a toolbit is compatible with a component is determined by the *shape* of the component. All components of the same part have the same shape. Such a toolbit exchange is performed as follows. First, the robot arm (without component) moves to the *Toolbit Exchange Unit* (TEU), as displayed in Figure 2.5. The old toolbit to be exchanged is placed in the TEU, and the new toolbit is extracted. Finally the arm moves to a feeder to pick up the component for which the toolbit exchange was required and continues picking and placing components. Since exchanging toolbits takes a relatively large amount of time, we include them in our model. Since this exchanging process is so costly, the number of toolbit exchanges is preferably kept to a minimum. On the other hand, avoiding as many toolbit exchanges as possible does not necessarily yield the best partitioning since this may cause imbalance of the workloads of the robots.

Figure 2.5: A toolbit exchange unit, used for exchanging toolbits within the pick-and-place process.

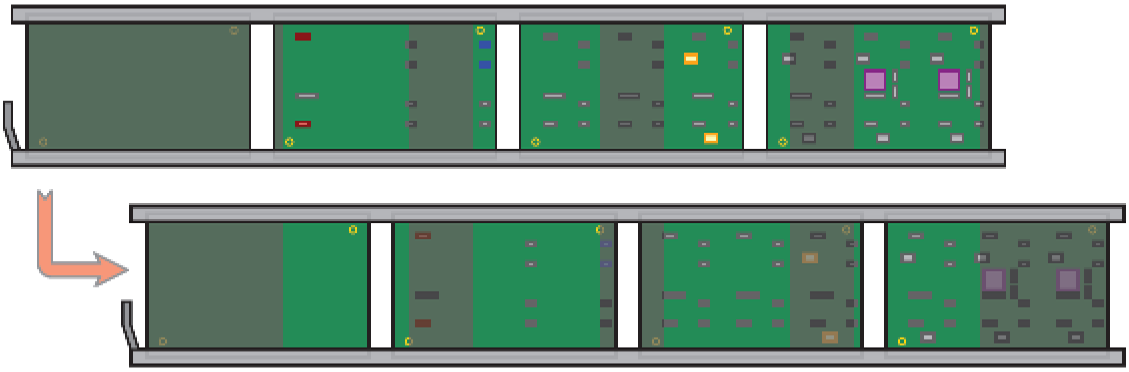


Figure 2.6: The process of an index step. A non-shaded area represents the work area of a placement robot. Shaded areas cannot be reached by any placement robot during that index.

Earlier it was explained that only after all placement robots have finished their tasks, the PCBs are transported along the transport system. The transportation of PCBs through the machine follows a stepwise process, as shown in Figure 2.6. One such step starts by the PCBs being located on a certain position. Then the placement robots do their assigned job by placing a number of components on the PCBs within their work area. When all placement robots are finished with their tasks, the transport system moves the PCBs such that other components can be placed on them. All actions taking place while the PCB's are in a certain position are executed over the course of an *index*. The number of index steps, and the locations of the PCBs in each index step is determined by the *transport scheme*. Observe that in case the workload between the placement robots is unbalanced, one robot is still placing components while the other robots are idle. That is, the duration of an index is defined as the maximum over the processing times of the placement robots. Since long idle times are indicators of possible improvement, we include index steps in our model. The overhead caused by transport is assumed to be constant.

In this section we have explained the operations of the pick-and-place machines under study. Although we covered the most important and most impactful aspects of these machines, there are still a lot of features that we omitted. Including all of the machines' complexity in a model is not reasonable, and yields a model which is impractical to work with. Although the model captures the most important aspects of the machines, one should keep in mind that our constructed model is not equal to reality, and that reality keeps changing due to new requests from customers.

2.1.1 iX Specifics

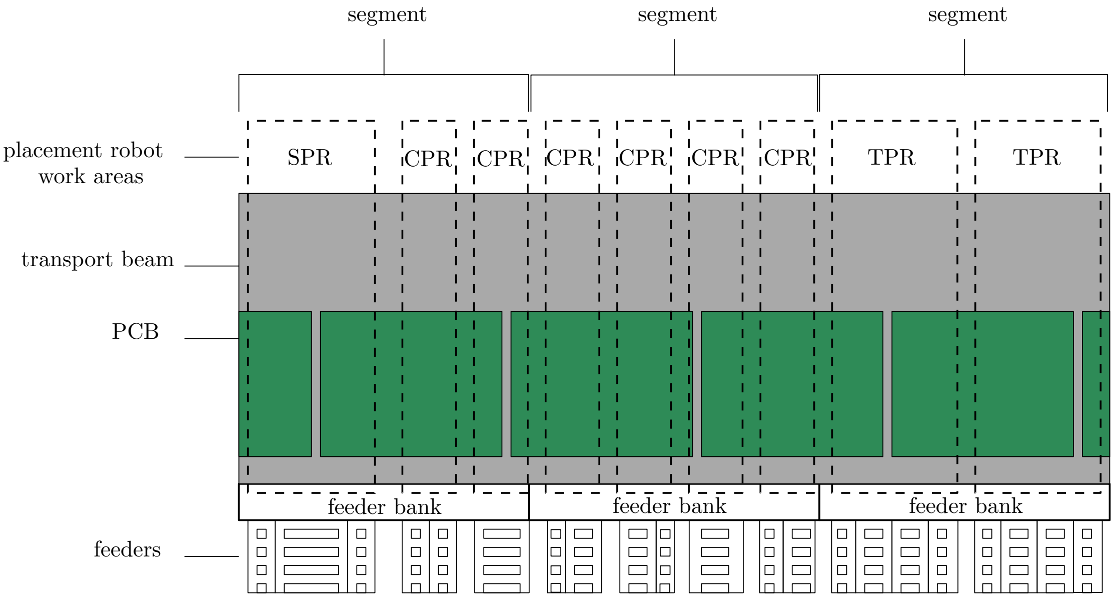
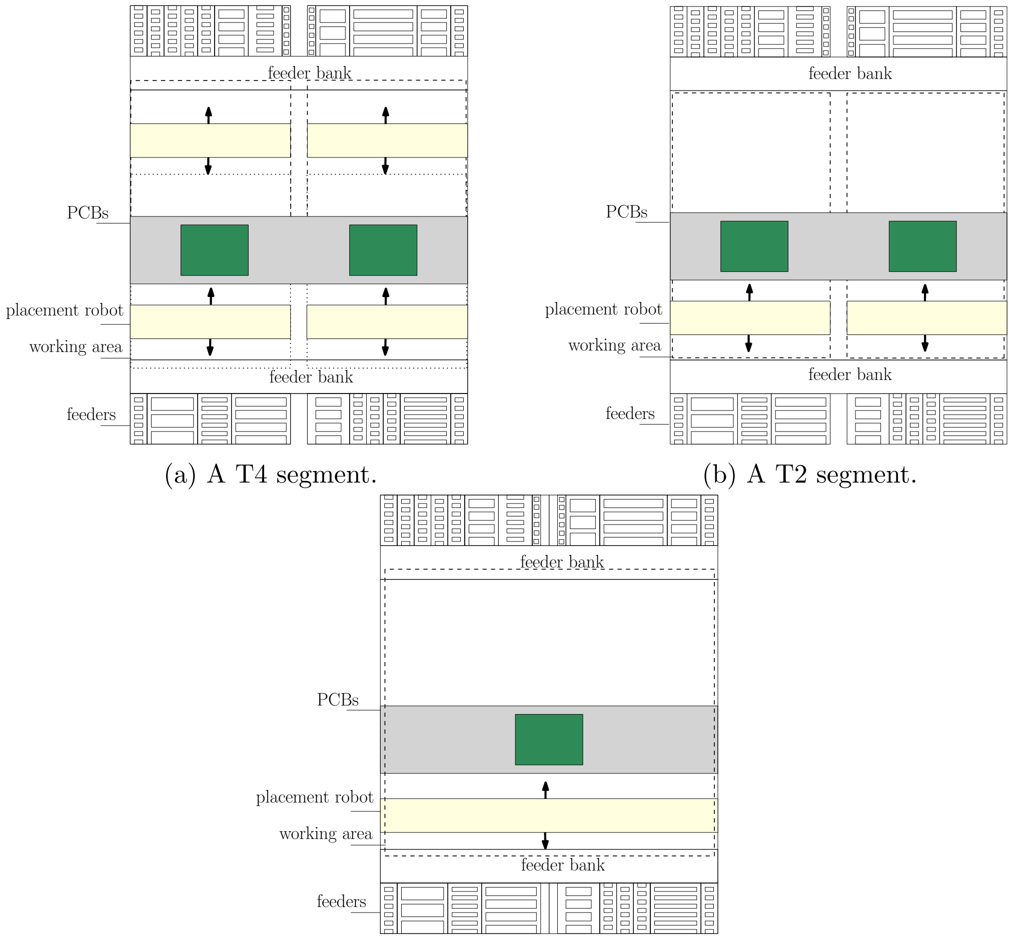


Figure 2.7: An example iX configuration.

Figure 2.7 provides a schematic illustration of an iX machine. The iX machine is divided into three or five different segments, each with four slots available for placement robots. There are three different types of placement robots: compact placement robots (CPRs), standard placement robots (SPRs) and twin placement robots (TPRs). A CPR takes up one slot, while an SPR and TPR take up two slots. TPRs always come in pairs, so a segment containing a TPR is always occupied by two TPRs. The work area of these different types of placement robots differs: A CPR has less space to move in the direction along the transport beam than an SPR or TPR. Furthermore a TPR has a slightly smaller work area than an SPR. Another difference between the type of robots is the equipment they have: An SPR or TPR can optionally be equipped with a camera whereas a CPR does not allow cameras due to limited space, for instance.

2.1.2 iFlex Specifics



(a) A T4 segment.

(b) A T2 segment.

(c) An H1 segment.

Figure 2.8: Three different segments of the iFlex machine.

Figure 2.8 provides a schematic illustration of the three different segment types of an iFlex machine. Whereas an iX machine allows for up to 20 placement robots, an iFlex machine typically has much less. The placement robots on an iFlex machine are much faster though. They have a larger work area and can reach more feeders. Whereas an iX segment merely indicates a boundary for the feeder banks and placement robots, an iFlex segment is more stand-alone. An iFlex segment has not one, but two feeder banks: one on the front of the machine and one at the back. There are three different types of iFlex segments: An H1 segment containing only one placement robot, a T2 segment containing two placement robots which can each reach (approximately) half of the feeders, and a T4 segment containing four placement robots, each reaching (approximately) a quarter of the feeders. These segments differ in the equipment they have: Some of them are equipped with a camera, some with a laser alignment module within the placement robots and some are equipped with both. Whereas in general an iFlex machine consists of up to eight segments, for our specific case of combination with an iX machine it usually only has one or two, mostly H1 and T2 segments.

Whereas the segments of an iX machine all share a transport scheme, iFlex segments each have their own transport scheme. Compared to iX segments, iFlex segments work much more independent of each other: they do not share index steps. One could argue that each of the segments of an iFlex machine constitutes its own independent machine within the assembly line.

Whereas an iX placement robot (of any type) has only one placement head, an iFlex placement robot has two of them. This entails that a placement robot of the iFlex machine can move to the feeder bank, pick up two components, move to the PCB to be assembled and place both components sequentially before returning to the feeder bank. Since this latter process requires less travel time for the robot arm, depending on the exact placement of the feeders it is often more efficient to use both placement heads instead of only one.

2.2 Problem Description

In this section, we formally describe the problem of distributing the workload over the two different types of machines. First we provide a high-level description in Subsection 2.2.1. Then we give a detailed explanation of our objective function and the corresponding model in Subsection 2.2.2. Finally, we provide a formal problem description in the form of an Integer Linear Program (ILP).

2.2.1 High-level problem description

Figure 2.9 provides a high-level overview of the current situation. A task, consisting of one or more PCBs to be processed, is registered by Valor MSS Process Preparation. This system determines which components should be placed using the iX machine and which components using the iFlex machine. Given this partition, K&S's optimizer software (also referred to as the optimizer) estimates the corresponding cycle time by computing a feeder setup, a toolbit setup and by determining which action needs to be performed by which placement robot in which index step. Based on the cycle time of each of the machines as returned by K&S's optimizer software, Valor MSS Process Preparation determines a new partition of components over the machines. After this process has been repeated a number of times, the best of the examined partitions is returned. Our aim is to improve upon the partition as determined by Valor MSS Process Preparation, and reduce the overall cycle time.

For the purpose of limiting the scope of research, we assume that the pick-and-place machines in the assembly line consist of exactly one iX machine followed by one iFlex machine. Investigations in the past have shown that in case of multiple machines of the same type, we can simulate them by one machine with more placement robots without deviating too much from reality.

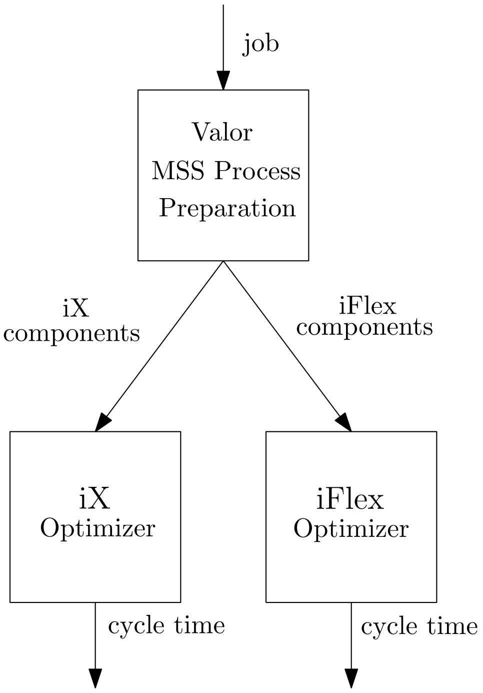


Figure 2.9: High-level problem description.

2.2.2 Objective function

The objective function that we try to minimize is the *cycle time*. The concept cycle time, denoted by CT, can be defined in multiple contexts. Although generally speaking the cycle time is defined as the duration of one cycle, depending on the context a different cycle may be meant. We refer to the type of a PCB as a *board type*. The cycle time $CT_M(C)$ of a set of components C, all on the same board type, on a certain machine M is defined as the time required to place all those components using the specified machine. Recall that a task consists of multiple PCBs, some of which are of the same type. The cycle time of a board type is defined as the time between one PCB leaving the machine and the next PCB of the same board type leaving the machine.

We refer to the number of PCBs of a board type b within a task as its *quantity* q_b. The cycle time of a partition $\Pi = (C_{\text{iX}}, C_{\text{iFlex}})$ of a set of components C is modeled by the summation over the cycle times of the different board types, weighted by their quantity. More formally, the cycle time of a partition $\Pi = (C_{\text{iX}}, C_{\text{iFlex}})$ of components C is given by

$$CT(\Pi) = \sum_{b \in \text{board types}} q_b \cdot \max\{CT_{\text{iX}}(C_{\text{iX}} \cap C_b), CT_{\text{iFlex}}(C_{\text{iFlex}} \cap C_b)\},$$

where C_b is the set of components corresponding to board type b. Note that $(C_{\text{iX}} \cap C_b) \cup (C_{\text{iFlex}} \cap C_b) = C_b$

Next we describe how we model the cycle time $CT_M(C)$ of a set of components C on a certain machine M. Recall that components are placed over multiple index steps. In each index step, the robots have to perform a set of actions taking a certain amount of time. In Figure 2.10

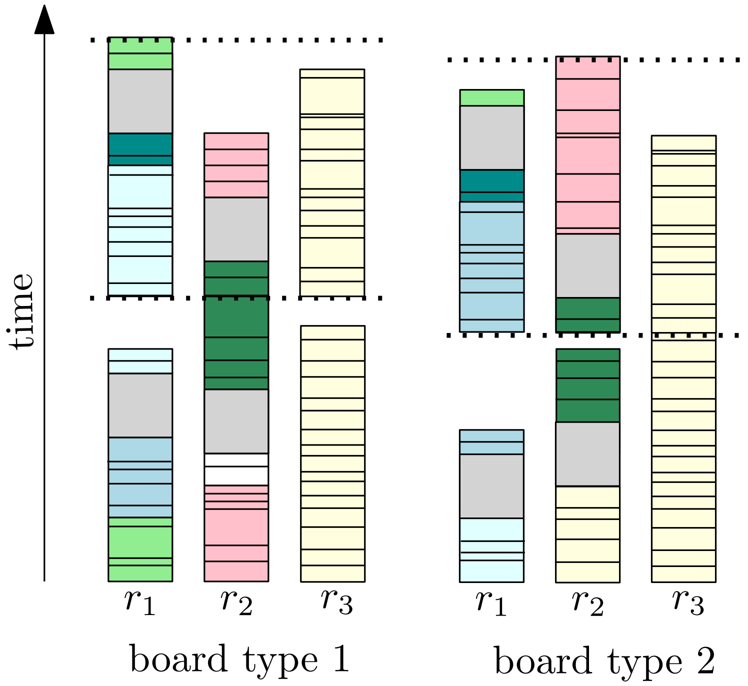


Figure 2.10: A schematic illustration of the concept of cycle time. In this example we have three placement robots on one segment and two board types, each with two index steps. Gray rectangles indicate toolbit exchanges, whereas rectangles of other colors represent the placement of a component, where components with the same color have the same part. As mentioned in Section 2.1 we omit fiducial measurements and transport actions from our model.

each dashed line indicates the end of an index step, and each vertical bar between two dashed lines (or the x-axis) represents the time each placement robot takes in that index step. Recall that we can only start transportation of the PCBs once all placement robots have finished performing their assigned actions, so placement robots have to wait for each other. In our model, the cycle time of a set of components C, all corresponding to board type b, on a certain machine M is given by

$$CT_M(C) = \begin{cases} \sum_{k \in \text{index steps}(b)} \max_{r \in \text{robots}} \{t_{bkr}\} & \text{if } M = \text{iX} \\ \max_{s \in \text{segments}} \sum_{k \in \text{index steps}(b,s)} \max_{r \in \text{robots}(s)} \{t_{bkr}\} & \text{if } M = \text{iFlex}, \end{cases} \tag{2.1}$$

where t_{bkr} is the time spent by placement robot r in index step k on board type b, index steps(b, s) is the set of index steps of board type b on iFlex segment s, index steps(b) is the set of index steps of board type b (for the iX machine) and robots(s) is the set of placement robots on segment s. Equation 2.1 shows that each iFlex segment has its own transport scheme, whereas all iX segments on an iX machine have the same transport scheme.

2.2.3 Formal problem description

With all terminology and definitions of the previous subsections in place, we are ready to present a formal problem definition of the problem under study.

Workload distribution among machines
Given a task consisting of a number of PCBs to be processed containing components C, determine a partition $\Pi = (C_{\text{iX}}, C_{\text{iFlex}})$ of C such that the cycle time of the partition $CT(\Pi)$ is minimized.

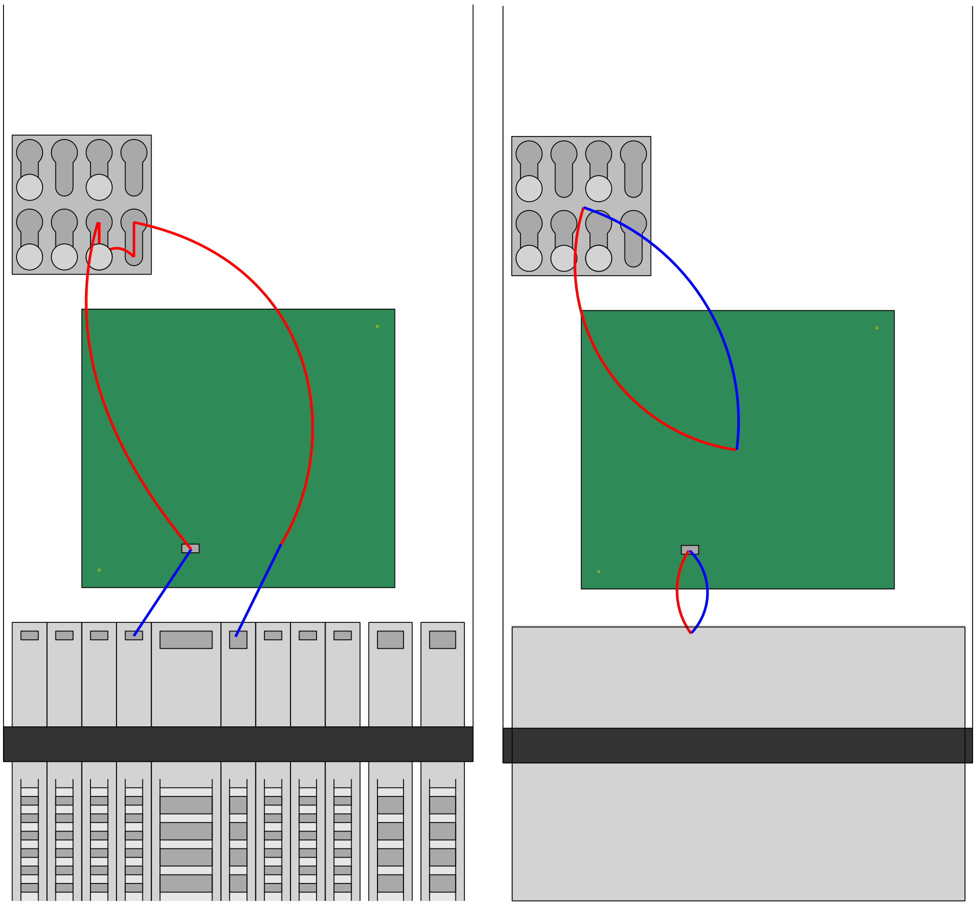
In order to evaluate our algorithms, we need to be able to compute $CT(\Pi)$ for a given partition. Although we could potentially determine $CT(\Pi)$ by executing our algorithms on the physical machines, this is not feasible for analysis of our algorithms. Instead, we could use the optimizer software to evaluate $CT(\Pi)$ for us. Depending on the size of the task, this computation takes somewhere in the range of a few minutes. This implies that we cannot evaluate a large number of partitions within our algorithms if we want them to return a solution within a reasonable amount of time. Since we want to be able to evaluate partitions fast, we have to refrain from viewing K&S's optimizer software as a black box, and have to include more detail in our model. In what follows we present a model for K&S's optimizer software. The model should still resemble the actual operation of the machines to a decent extent, while being sufficiently fast to provide us with a means to analyze our algorithms. The optimizer software is solving the following optimization problem.

Workload distribution among placement robots
Given a set C of components to be placed on a machine M, determine an assignment of C to the set of placement robots minimizing $CT_M(C)$.

In addition to computing an assignment of components to placement robots, the optimizer computes many more such as the feeder setup, the toolbit setup, the transport scheme and the order in which the components are placed. To keep our model simple, we only focus on determining an assignment of components to robots. In order to model the optimizer software, we can no longer reason on the level of machines: we need to include the assignment of components to robots in our model. In the remainder of this section we extend our model to accommodate for reasoning on the level of placement robots and provide an ILP formally describing the problem.

The time required to process a component can be subdivided into a number of steps. Recall that all components need to be picked from the feeder bank, transported to the camera and measured for alignment if needed, moved to their corresponding position on the PCB and placed on the PCB, whereafter the robot arm has to move back to the feeder bank. Since all these steps have to be performed for every component, we aggregate them into one single estimate to obtain the processing time, denoted by p_{cr}, for component c on placement robot r.

The time required for a toolbit exchange on placement robot r consists of the travel time from the PCB to the TEU, the time it takes to actually exchange the toolbit and the travel time back to the PCB. Figure 2.11 displays how we include the toolbit exchange time in our model. While actually a toolbit exchange requires the robot arm to move from the PCB to the TEU and to the feeder, we only take the movement to the PCB into account. Since an iFlex placement robot has two placement heads, it can pick up two toolbits sequentially saving travel time required. However, to keep our model simple we assume that there is always only one toolbit exchange performed. We assume that the exchange time is independent of the toolbit picked up. Although in reality the position of the toolbits in the TEU matters, the error we induce is disproportionate relative to the travel time. We aggregate the time required for each of these steps in one single estimate $exch_r$ representing the toolbit exchange time on placement robot r.



(a) The red lines indicate the movement of the robot arm for a toolbit exchange. The blue lines indicate the travel time required to pick and place a component without toolbit exchange. To perform a toolbit exchange, we need to travel along the red lines before returning to normal operation.

(b) The way the timing model measures the toolbit exchange in Figure 2.11. Note that the travel time deviates from reality, since our model assumes we stop the robot arm while moving from the TEU to the feeder bank, while in reality we can move the robot arm without stopping. Furthermore note that the feeder setup is unknown in our model.

Figure 2.11: A comparison of the movements needed for a toolbit exchange shown in Figure 2.11a and the way we modeled these movements in Figure 2.11b.

As depicted in Figures 2.7 and 2.8, some parts of the transport system are unreachable by any of the placement robots. Mounting a feeder in front of such a 'dead zone' is not useful, since the feeder cannot be used at all. Although some parts of this dead zone can still be productive by mounting wider feeders, we do not take such details into account. Although the feeders can only be placed in discrete positions on the feeder bank, we assume that they can be mounted anywhere along the feeder bank in a continuous fashion. Since customers only have a limited number of feeders available, there may be an limit on the number of feeders used for a particular part. We refer to this limit as the *inventory limit* of that part or feeder. This allows us to model the feeders and feeder banks by introducing a variable f_p, which is the feed space required for a feeder containing part p in mm and ignore the existence of dead zones and special, wider, feeders.

We introduce another variable Y_{pkr}, denoting the set of components of part p that can be placed on robot r during index step k. That is, $c \in Y_{pkr}$ for a component c of part p if and only if the component is compatible with robot r and the place where the component needs to be placed on the PCB is within the working area of robot r during index step k.

For the toolbit exchange times, we introduce a variable T_{bkr} indicating the set of toolbits needed on robot r during index step k on board type b. Since the locations of the components on the PCBs of one part are in general not grouped together, often we need the same set of toolbits in each index step for the same board type. We assume that if we need t different toolbits on robot r during index step k on board type b, we need $t-1$ toolbit switches, since we can use one of the required toolbits from the previous index step. In other words, in this model we ignore that $T_{bkr} \bigcap T_{b(k+1)r}$ may be empty.

We now present an ILP for the Workload distribution among placement robots problem. The other variables used are relatively straightforward and not explained in further detail. Note that the model below technically is not an ILP, since we use sets for output variables and we use the maximum function. However, we can transform the model below to an ILP straightforwardly.

Input parameters:

$$
\begin{aligned}
B &= \text{set of board types to be produced} \\
q_b &= \text{quantity of board type } b \\
P_b &= \text{parts for which components need to be placed on board type } b \\
N_p &= \text{set of components of part } p \\
p_{cr} &= \text{processing time of component } c \text{ on robot } r \\
exch_r &= \text{time required to exchange toolbits on robot } r \\
f_p &= \text{feed space needed for a feeder containing part } p \text{ in mm} \\
\ell_p &= \text{inventory limit of part } p \\
w_r &= \text{the width of the feeder bank of robot } r \text{ in mm}^1 \\
c_r &= \text{toolbit capacity on robot } r \\
comp(p,t) &= \begin{cases} 1 & \text{if part } p \text{ is compatible with toolbit } t \\ 0 & \text{otherwise.} \end{cases} \\
Y_{pkr} &= \text{set of components of part } p \text{ that can be placed on robot } r \\
&\quad \text{during index step } k
\end{aligned}
$$

Output variables:

$$
\begin{aligned}
T_{bkr} &= \text{set of toolbits needed on robot } r \text{ during index step } k \text{ on board type } b \\
C_{pkrb} &= \text{set of components of part } p \text{ placed on robot } r \\
&\quad \text{during index step } k \text{ on board type } b
\end{aligned}
$$

The objective function is the weighted sum over the cycle times of the board types. Now that our model contains more detail, we can specify t_{bkr}, the processing time of robot r in index step k on board type b, more precisely. It consists of the pick and place time of the components placed during index step k on robot r, and the toolbit exchange time, yielding Equation 2.2.

$$t_{bkr} = \sum_{p \in P_b} \sum_{c \in C_{pkrb}} p_{cr} + (|T_{bkr}| - 1) \cdot exch_r. \tag{2.2}$$

The objective function is then simply the weighted sum over the cycle times of the board types, where we substitute the right hand side of Equation 2.2 for t_{bkr} in Equation 2.1.

$$\text{Minimize} \sum_{b \in B} \left(q_b \cdot CT_M(C_b)\right),$$

where C_b is the set of components to be placed on board type b.

An assignment from components to placement robots is feasible if and only if it satisfies the following constraints:

[1] For H1 and T2 segments this is the sum of the widths of the feeder banks that robot r can reach

1. Every component is placed exactly once.

$$\bigcup_{k \in \text{index steps}} \bigcup_{r \in \text{robots}} C_{pkrb} = N_p \text{ for all } p \in P_b, b \in B$$

$$\sum_{k \in \text{index steps}} \sum_{r \in \text{robots}} |C_{pkrb}| = |N_p| \text{ for all } p \in P_b, b \in B$$

2. If at least one component of part p is placed on robot r during index step k, there should be a compatible toolbit on robot r during index step k.

$$C_{pkrb} \neq \emptyset \Rightarrow \sum_{t \in T_{bkr}} comp(p,t) > 0 \text{ for all } p \in P_b, b \in B, r \in \text{robots}, k \in \text{index steps}$$

3. The feeders needed for the parts assigned to a robot do not exceed the robot's feed space.

$$\sum_{k \in \text{index steps}} \sum_{\substack{p \in P_b \\ C_{pkrb} \neq \emptyset}} (f_p) \leq w_r \text{ for all } r \in \text{robots}, b \in B$$

4. The inventory limit of parts is not exceeded.

$$\sum_{r \in \text{robots}} \max\{1, \sum_{\substack{k \in \text{index steps} \\ C_{pkrb} \neq \emptyset}} 1\} \leq \ell_p \text{ for all } p \in P_b, b \in B$$

5. All components of part p assigned to robot r during index step k can be placed on robot r during index step k.

$$C_{pkrb} \subseteq Y_{pkr} \text{ for all } p \in P_b, b \in B, r \in \text{robots}, k \in \text{index steps}$$

6. The number of toolbits required on a placement robot does not exceed its TEU's capacity.

$$\left| \bigcup_{k \in \text{index steps}} T_{bkr} \right| \leq c_r \text{ for all } r \in \text{robots}, b \in B$$

2.3 Complexity

In this section, we formally prove that the problems Workload distribution among machines and Workload distribution among placement robots are NP-hard.

Theorem 1. The decision variant of Workload distribution among placement robots is NP-hard.

Proof. Let (J, M) be an instance of $P \mid\mid C_{\max}$, where J is a set of tasks that needs to be scheduled on one of the identical processors in M. Let t_j be the time it takes to execute task j on any processor in M. To construct an instance of Workload distribution among

PLACEMENT ROBOTS, let there be only one PCB containing a component for each task $j \in J$ with processing time t_j on any placement robot, and suppose all components do not need the camera for alignment. Suppose that all actions are performed in one index step. Furthermore suppose that all components have the same part, such that only one feeder and one toolbit is required on each placement robot. Trivially, the instance of WORKLOAD DISTRIBUTION AMONG PLACEMENT ROBOTS constructed by this reduction has a schedule with cycle time less than or equal to t if and only if the original instance of LOAD BALANCING has a schedule with completion time less than or equal to t, and the reduction runs in polynomial time. □

Theorem 2. The decision variant of WORKLOAD DISTRIBUTION AMONG MACHINES is NP-hard.

Proof. Take $M = \{M_1, M_2\}$ and assume both the iX machine and iFlex machine have only one placement robot. Then using the same reduction as above yields that WORKLOAD DISTRIBUTION AMONG MACHINES is NP-hard. □

Chapter 3

Algorithms

In this chapter we describe the algorithms we developed to solve the problem as formalized in the previous chapter. Our main approach consists of a simulated annealing algorithm, applied to the specifics of our domain, which is explained in Section 3.1. Although we could evaluate a partition of components over machines using the optimizer software, it takes quite some time. For this reason, we develop a greedy algorithm to distribute components over placement robots, as described in Section 3.2.

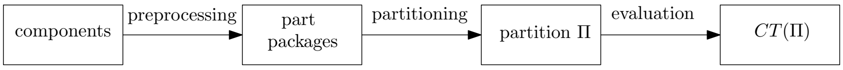


Figure 3.1: An overview of the steps performed to obtain a cycle time given a set of components.

Before describing our algorithms, we discuss an important design decision. Since feeder banks have limited width and each feeder only contains components of one part, assigning a large number of different parts to one placement robot yields an infeasible solution. On the other hand, some parts are relatively large in terms of number of components. Assigning such a large part to a single placement robot certainly does not yield an optimal solution. For this reason in our algorithms we consider workload distribution on the level of parts, where we allow for splitting parts. For example, a large part with 100 components may be split into two and assigned to two placement robots, yielding two *part packages* with 50 components each. These part packages do not necessarily need to have the same number of components. In Section 3.2.1 we explain this splitting process in much more detail. In our algorithms, we consider a part package to be an indivisible unit, thus constraining the solution in such a way that all components in a part package are placed by the same placement robot. Our balancing algorithms distribute the workload on the level of these part packages.

Figure 3.1 outlines the main steps performed in the algorithm. Given a set of components as input, we apply preprocessing in the form of creating part packages. Then, we apply our balancing algorithm to obtain a partition Π over the machines. Finally, the partition is evaluated to obtain a cycle time $CT(\Pi)$ for partition Π.

3.1 Simulated Annealing

Simulated annealing is a commonly used, generic method for solving combinatorial optimization problems. The main strength of simulated annealing lies in the adjective 'generic': it can be applied to a large number of optimization problems. However, one should not expect the algorithm to report a global optimum. Instead, the aim is to obtain an acceptable local optimum within a reasonable amount of time. The algorithm is inspired by the field of thermodynamics. Annealing is a physical process in which a heated metal is cooled by slowly decreasing the temperature. In the fully heated state, all particles of the metal are randomly arranged. By cooling, the atoms of the metal get aligned, eventually resulting in a low energy level state. Simulated annealing applies the same concept in the context of optimization. In the following paragraphs we first describe how simulated annealing works in general, after which we apply it to our specific problem.

In the spectrum of optimization techniques ranging from random brute-force search to local search, simulated annealing lies somewhere in between. During execution of the algorithm, simulated annealing shifts from a random brute-force approach to an approach more like local search. A major downside of using a random brute-force search is that a lot of evaluations have to be examined, which takes too much time. A major downside of local search is that the quality of the solution returned by the search depends heavily on the choice of the initial solution, due to local optima in the search space. Simulated annealing attempts to overcome the drawbacks of both approaches by means of a probabilistic approach, combining both paradigms.

```
SIMULATED ANNEALING($\Pi_{init}, T_{init}, T_{stop}$)
$\Pi_{current} = \Pi_{init}$ // current solution
$T = T_{init}$ // temperature
while $T > T_{stop}$
    $\Pi_{neighbor}$ = GENERATENEIGHBOR($\Pi_{current}$) // neighboring solution
    if $\mathbb{P}(\Pi_{current}, \Pi_{neighbor}, T) > Random[0, 1)$
        $\Pi_{current} = \Pi_{neighbor}$
    $T$ = COOLINGSCHEDULE($T$)
return $\Pi_{current}$
```

The procedure SIMULATED ANNEALING($\Pi_{init}, T_{init}, T_{stop}$) contains the pseudocode for the algorithm. The algorithm starts from an initial solution, after which a neighboring solution is generated using a certain neighborhood function. If the neighboring solution has a lower cost, denoted by C, it is accepted. Otherwise, the neighboring solution is accepted with a certain *acceptance probability* $\mathbb{P}(\Pi_{current}, \Pi_{neighbor}, T)$. Note that we sometimes want to accept higher-cost solutions, to avoid degenerating our search to local search. This acceptance probability depends on the cost of both solutions and how many solutions have been evaluated over the course of the execution of the algorithm. The more solutions have been evaluated, the smaller the acceptance probability for neighboring solutions with a higher cost than the cost of the current solution. This means that at the start of the execution of the algorithm one expects worse solutions to be accepted with high probability, whereas towards the end of the execution the algorithm converges to a local optimum, whose value is hopefully close to the value of a global optimum.

The dynamic aspect of the acceptance probability is captured by the concepts of temperature and cooling schedule. The algorithm starts with a pre-computed initial temperature and decreases the temperature after each iteration. The amount of decrease in temperature is determined by the cooling schedule. The lower the temperature, the lower the acceptance probability for higher cost solutions. A survey as provided by (Suman & Kumar, 2006) discusses numerous studies that have been devoted to finding an initial temperature and cooling schedule yielding acceptable results. Finding good parameters for simulated annealing in general is no easy task: the parameters yielding the best solutions are often dependent on the application. Using a high temperature and slowly decreasing cooling schedule implies evaluating a relatively large number of solutions, whereas using a too low temperature may result in the simulated annealing algorithm to degenerate to a local search algorithm.

To limit the scope of the thesis, we refrain from finding an optimal initial temperature and cooling schedule. Instead, we make use of the vast body of literature available for choosing the initial temperature and cooling schedule. Kirkpatrick, Vecchi, and Gelatt (1983) propose to choose the initial temperature T_{init} as the maximal cost difference between any two neighboring solutions. More recently, Aarts, Korst, and Michiels (2005) have shown that choosing $T_{\text{init}} = K \cdot \sigma_\infty^2$ yields acceptable results, where K is a constant typically ranging from 5 to 10 and σ_∞^2 is the variance of the underlying distribution of cycle times. Since approximating the variance of a distribution requires a large number of samples (Thompson & Endriss, 1961), we instead use an initial temperature based on average increase as proposed by Johnson, Aragon, McGeoch, and Schevon (1989). More precisely, the initial temperature is determined by $T_{init} = -(\Delta E / \ln \chi_0)$, where ΔE is an estimate of the cost increase of the objective function, where a decrease is counted as an increase of zero. The value χ_0 is the desired acceptance ratio, which is set to 0.4 in our implementation. As for the cooling schedule, we use the commonly used geometric cooling schedule, given by $T_{i+1} = \alpha \cdot T_i$, where T_i is the temperature during iteration i and α is a constant, typically ranging between 0.8 and 0.99. In our implementation we determine α based on the initial temperature, final temperature and number of evaluations. That is, $\alpha = \sqrt[x]{(T_{stop}/T_{init})}$, where x is the number of evaluations. Results of experiments performed to find a good initial temperature and cooling schedule for our specific problem can be found in Chapter 4.

Now that the general approach used by simulated annealing is clear, we describe how we apply the method to our specific problem. Our search space consists of 2^n partitions for n part packages. The cost of a partition Π is defined by the cycle time of that partition $CT(\Pi)$. As for the neighborhood function, there are various methods we can use. We discuss each of the neighborhood function used in Subsection 3.1.1.

We use the acceptance probability function in Equation 3.1 as proposed by Kirkpatrick et al. (1983) and commonly used in practice, also known as the Boltzmann probability distribution.

$$P(\Pi_{current}, \Pi_{neighbor}, T) = \exp\left(\frac{C(\Pi_{current}) - C(\Pi_{neighbor})}{T}\right) \tag{3.1}$$

In Equation 3.1 $\Pi_{current}$ is the current partition, $\Pi_{neighbor}$ is the neighboring partition, $C(\Pi)$ is the cost of partition Π and T is the current temperature.

3.1.1 Neighboring functions

We have specified the search space, quality function, and acceptance probability function, and have discussed the initial temperature and cooling schedule. It remains to specify the neighboring function. The choice of neighboring function determines the optimization landscape. By applying the neighborhood function to a certain partition, preferably the cycle time of the partition changes. Ideally, a neighboring function avoids barriers in the landscape. That is, the number of local optima is not too large. Having a large number of such local optima may cause the algorithm to become stuck in such a local optimum, with a very low probability of getting out if the landscape is shaped in such a way that the local optimum lies within a deep valley (or steep hill, in case of maximization). In this subsection, we describe a number of neighboring methods that can be used within the simulated annealing framework for our specific problem.

3.1.1.1 Move-k neighborhood

The "natural" and intuitive neighboring method is to move one part package from its current machine to the other machine. In our algorithm, we choose a slightly more general version, where the neighborhood of a partition Π is given by all partitions obtained by moving k part packages to the other machine. Note that this neighboring function also accounts for swaps: For example, if $k = 2$ and we choose two part packages on different machines, the neighboring partition will have these two part packages swapped.

3.1.1.2 Clustering

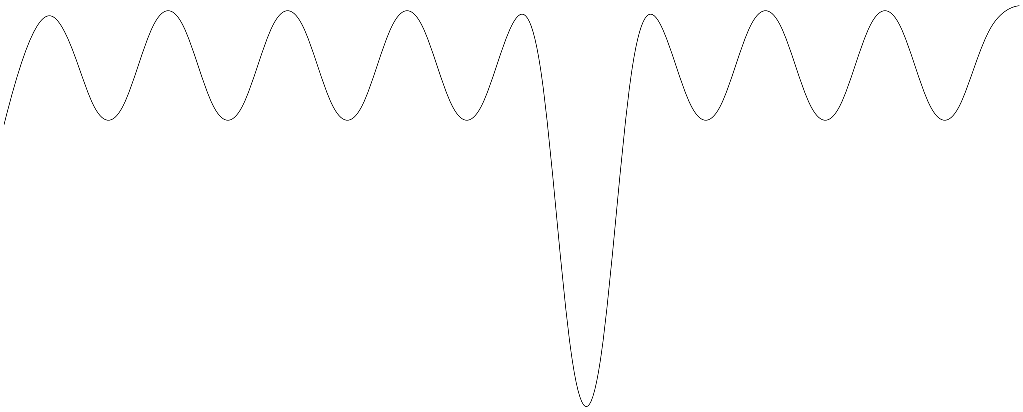


Figure 3.2: A schematic representation of the optimization landscape for number partitioning using the move-k neighborhood.

The neighboring function in the previous subsection directly changes the partition. In this subsection we describe an indirect neighboring function: it changes the partition only indirectly via a constraint imposed upon the partition. Our motivation for this less-intuitive neighboring function is justified by the barrier avoidance property that we would like a neighborhood function to satisfy. By applying this other neighboring function we obtain a different optimization landscape, which yields different results for the simulated annealing framework.

Johnson, Aragon, McGeoch, and Schevon (1991) discuss the application of simulated annealing on the related problem of number partitioning: given a set of numbers, partition them over two sets such that the sum of the numbers in each set is minimized. They report that the "natural" neighborhood function similar to our move-k neighborhood does not yield desirable results due to the large number of local optima in the search space. These local optima are surrounded by steep hills in the landscape (as displayed in Figure 3.2, and the value of these local optima is much larger than the value of the global optimum. More specifically, the expected value of the optimal solution is exponentially small, while the expected cost difference between neighboring solutions is only polynomially small. The neighborhood function discussed in this section is inspired by Ruml, Ngo, Marks, and Shieber (1996). They impose a constraint on the partition to obtain an optimization landscape with less local optima for number partitioning, resulting in a bumpy funnel as displayed in Figure 3.3.

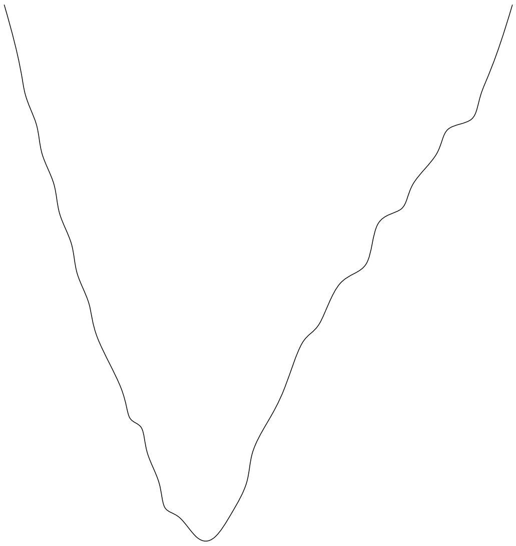


Figure 3.3: A schematic representation of the optimization landscape for number partitioning using the clustering neighborhood.

The constraint we impose upon the partition consists of a *clustering* of the part packages. Each part package is assigned to a certain cluster, grouping part packages together, as depicted in Figure 3.4. All part packages in the same cluster, from here on referred to as a *part cluster*, are obliged to be grouped together within the partition, whether we distribute on machine level or the level of placement robots. By mutating the clustering of the part packages, we impose different constraints upon the partition to be evaluated, yielding our indirect neighboring function. The question remains how a clustering of part packages translates to an actual partition. Essentially, we still have the same problem as before: Determine which units should be assigned to which machine, or to which placement robot, where the units are no longer part packages but part clusters.

To obtain a partition from a clustering of part packages, we need a different algorithm to assign part clusters to machines. There are a number of different ways how to obtain such an algorithm. One of them is to use the move-k neighboring function as described above in Subsection 3.1.1.1 in another simulated annealing algorithm, yielding an approach using two layers of simulated annealing. In the outer simulated annealing algorithm we perform

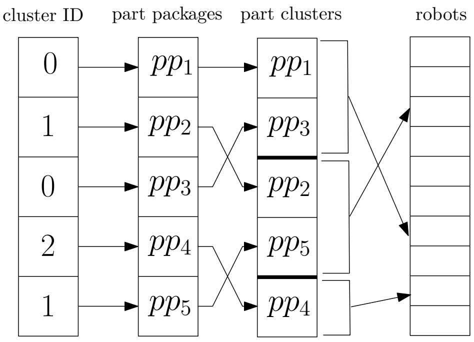


Figure 3.4: The process of constructing part clusters. Part clusters are identified by their cluster ID.

mutations upon the part clustering, whereas the inner simulated annealing algorithm assigns the formed part clusters to either machines or placement robots. One should question the number of evaluations required to obtain reasonable solutions using such a two-layered approach: How many evaluations should be performed within the inner layer before returning a partition? In any case, the number of evaluations required to obtain a reasonable solution increases considerably compared to the direct move-k neighborhood function.

Another approach assigns part clusters greedily. In order for such a greedy algorithm to work, we need to know how much time a part cluster needs on a particular machine. However, quantifying the processing time of a component or set of components on a machine is not trivial: it depends on the distribution among the placement robots and the required toolbits. Instead of assigning part clusters to machines, we utilize our more detailed model and assign them to placement robots instead. We use a greedy algorithm to compute an assignment of part clusters to placement robots, which is described in detail in Section 3.2. The output of this greedy algorithm is an assignment of part clusters to placement robots, which we can lift to the layer of machines: If the assigned placement robot is an iX robot the part package should be placed on the iX, otherwise it should be placed on the iFlex.

Now that we have a way to transform a clustering of part packages to a partition over machines, we can evaluate a clustering. However, we should still specify how the clustering comes about and how we change it between simulated annealing iterations. As initial part clustering, we choose an integer k indicating the number of clusters and iterate over the part packages one by one, such that the i^{th} part package that we process is assigned to part cluster $i \mod k$. The number of clusters can be chosen proportionally to the number of placement robots. By setting $k = c \cdot \#(\text{placement robots})$ for a positive real c, we introduce a tradeoff. Putting c low yields more constraints upon the distribution, whereas putting c high increases the size of the neighborhood and hence may reduce the rate of convergence of the algorithm.

As for mutating the clustering, we have multiple options. The intuitive approach is to select an arbitrary part package and move it to an arbitrary part cluster. Combined with the approach of assigning part clusters to the same placement robot, however, this has a drawback: If the part package being moved is compatible with a different toolbit than part packages in the part cluster that it is being moved to, we immediately introduce a toolbit exchange on the placement robot that the part cluster is assigned to. Recall that in general toolbit exchange compose a considerable portion of the cycle time. Toolbits cause another problem for our clustering approach: Suppose we have a part cluster with a number of part packages all placed with the same toolbit, and assume this part cluster is assigned to a certain placement robot r. By moving only one part package to another part cluster, we do not decrease the number of toolbits required on robot r. Only by moving all part packages placed with a certain toolbit, we might decrease the number of toolbit exchanges, potentially leading to a decrease in cycle time. On the other hand, moving all part packages to the same placement robot may yield a large workload on that particular placement robot, so we do want to introduce a maximum size on the number of components in a part cluster. For this reason, we restrict the size of a part cluster in terms of components to at most the expected number of components placed per placement robot: the total number of components divided by the number of placement robots. In other words, we do not take into account the domain knowledge we have on toolbit exchanges by mutating the part clustering using the intuitive mutation.

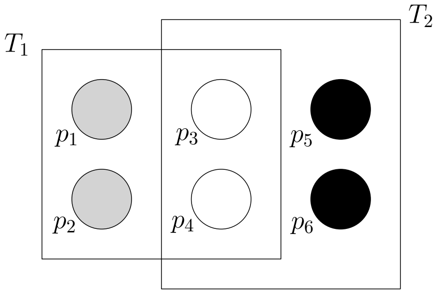


Figure 3.5: An example instance illustrating toolbit clusters. In the figure, circles represent parts and rectangles indicate toolbits. A part is compatible with a toolbit if it is located within the rectangle corresponding to that toolbit. There are two toolbit clusters: one containing the gray parts and one containing the black parts.

We can improve upon the mutations as described above by taking into account the toolbit exchanges implied by our choice of part clusters. As reasoned above, mutating the clustering of part packages such that part packages that are placed with the same toolbit are kept together intuitively makes sense. To find such a set of part packages requires us to know which toolbit is being used to place the components in the part packages. Within the simulated annealing framework using the greedy algorithm, we have an assignment from part packages to placement robots obtained from the previous iteration of simulated annealing. Using this assignment, we can compute which mutations yield a difference in number of toolbits required.

Given a placement robot r, its set of assigned part packages P_{rb} on board type b and the corresponding set of toolbits used T_{rb}, define a *toolbit cluster* P_{rtb} as a minimum cardinality set of part packages such that $P_{rtb} \subseteq P_{rb}$, all part packages in P_{rtb} are compatible with toolbit t and all part packages in $P_{rb} \setminus P_{rtb}$ can be placed using only toolbits in $T_{rb} \setminus \{t\}$. An example instance is shown in Figure 3.5. Our mutation can now be described as follows. Instead of choosing an arbitrary part package and moving it to an arbitrary part cluster, we select an arbitrary placement robot r and an arbitrary toolbit t from the set of toolbits used to place components on r. We compute the toolbit cluster P_{rtb} and move all part packages in P_{rtb} to another arbitrary part cluster. The mutation described in this paragraph provides us with a means to incorporate domain knowledge within the algorithm by keeping parts placed using the same toolbit together.

3.2 Greedy Algorithm

Within our simulated annealing framework, we need to compute the cycle time $CT(\Pi)$ of a partition Π for every iteration. Although we can ask K&S's optimizer software to provide an estimate of $CT(\Pi)$ for us, it takes a relatively large amount of time to compute such an estimate. Instead, we use our own *greedy algorithm* to assign part packages (or part clusters) to placement robots. Based on this assignment, we compute an estimate of the cycle time using our *timing model.* Of course, we can also use the greedy algorithm directly to obtain a partition over placement robots, without using simulated annealing at all.

The greedy algorithm combined with the timing model is essentially a realization of a model of the machines. However, the model is simpler as the one described in Chapter 2. While it does take into account the distribution of components among placement robots and different board types, it abstracts from index steps and computing a feeder setup. In terms of the ILP in Subsection 2.2.3, we do not take into account constraints 3 and 4 and abstract from index steps. We only take index steps into account when computing the cycle time: We assume that we need toolbit exchanges in each index step. The number of index steps is extracted from the optimizer.

The remainder of this section is structured as follows. First we discuss how we group components together to obtain part packages in Subsection 3.2.1, after which we explain the greedy algorithm itself in Subsection 3.2.2. In an attempt to improve the greedy algorithm, we introduce a partial toolbit setup in Subsection 3.2.3. Afterwards in Section 3.2.4, we explain our timing model used to evaluate a partition, where we devote some extra attention to computing the number of toolbit exchanges in Subsection 3.2.5.

3.2.1 Determining indivisible units

By visualizing the number of components per part, we find a typical distribution as depicted in Figure 3.6. By naively assigning parts (that is, all components of the same part) to placement robots, robots that are assigned parts with a large number of components become a bottleneck. It is apparent that we cannot consider the set of all components of a certain part as one single indivisible unit. On the other hand, regarding one single component as an indivisible unit yields problems as well. Recall that a placement robot can only reach a limited number of feeders, and each feeder only contains components of a single part. By assigning components to placement robots, we should take into consideration while developing our algorithms that we cannot assign too many components of different parts to one placement robot, lest we obtain an infeasible solution. Additionally, by considering components as indivisible units the search space is much larger compared to larger units.

Since both considering an entire part or a single component as an indivisible unit yields problems, we choose a unit which is somewhere in between, in the form of a *part package.* A part package is defined as a set of components of the same part. The part packages of a part form a partition of that part's components. Our choice for using part packages as indivisible units raises the question of how these part packages should be chosen, which we will address in the remainder of this subsection.

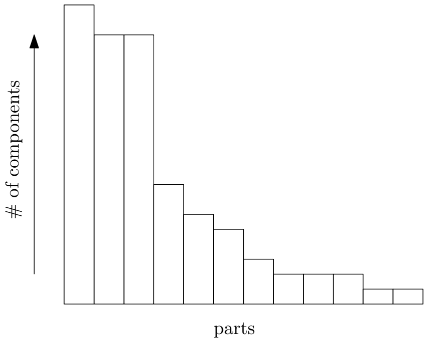


Figure 3.6: A typical distribution of parts in terms of number of components.

Determine part packages
Given a set of components C to be processed on a set of placement robots R, determine a set of part packages PP such that the cycle time $CT(C)$ is minimized.

As a first observation, we note that we should take into account on which board type components reside. Consider the situation where we have two board types b_1 and b_2 each containing components of the same part. Suppose board type b_1 contains 500 components, while board type b_2 contains 50 components. For determining our part packages, we have to make sure that the 50 components of board type b_2 are distributed among the placement robots, and not put together in one part package if we want the workload for board type b_2 to be balanced. Depending on the total number of components, we want to split the 500 components of board type b_1 in multiple indivisible units.

In what follows we describe a number of heuristics designed to tackle the problem of determining part packages. When we decide to split the components of a part into a number of part packages, we could assign each of the part packages the same number of components. Instead, all our heuristics work by determining a package size up front. The intuition of the package size is an indicator as to how many components should be placed on a single placement robot. As long as there are part packages that are larger than this desired package size, we split the part package into two packages: one with size equal to the desired package size and one containing the remainder. We keep iterating this process until all part packages have a size smaller than or equal to the desired package size, or until the *inventory limit* has been reached. Some parts have an inventory limit, indicating the maximum number of feeders containing that part that may be mounted to the machine. The heuristics we use for the package size are the following:

- average number of components per robot
- $\varepsilon \cdot$ average number of components per robot
- average pick-and-place time required per robot
- $\varepsilon \cdot$ average pick-and-place time required per robot.

Here ε is a parameter between 0 and 1 controlling how much 'space' we leave for other components to be placed on the placement robot. The intuition is as follows: A part package with size of the average number of components or average time per robot likely is placed on one placement robot, leaving no slack for other parts to be placed in a solution where each robot has the exact same amount of work. By leaving some room for other parts, we prevent the solution that a lot of robots have only one assigned part, whereas all smaller part packages are placed on one placement robot. This solution is not desirable since assigning a large number of parts to a single placement robot yields a large number of toolbit exchanges, and the solution may require so many feeders that the feed space for that placement robot is exceeded.

3.2.2 Greedy algorithm using workload vectors

In order to evaluate a partition Π of components over the iX and iFlex machine without using K&S's optimizer software, we need to allocate part packages to individual placement robots since quantifying the processing time of a single component on a machine is not an easy task: We would have to incorporate the distribution of components to placement robots and toolbit exchanges. In this subsection we describe a greedy approach for computing an assignment from part packages to placement robots, which is not designed to assign part packages optimally, but to give a reasonable assignment within a short amount of time.

In order to make our explanation of the algorithm clearer, we first consider a much simpler scheduling problem, commonly known as $P||C_{\max}$ using the notation of Graham et al. (1979). This problem asks to assign tasks to processors such that the maximum completion time of the tasks is minimized, where the processing time of a task is fixed and independent of the processor on which it is scheduled. The greedy algorithm for $P \mid\mid C_{\max}$ is quite straightforward: From the set of unscheduled tasks, pick a task with maximal processing time and schedule it on the processor that has the smallest workload at that point in time. Our greedy algorithm for the assignment problem of part packages to placement robots is based upon the greedy algorithm for $P \mid\mid C_{\max}$. We cannot directly apply this simple greedy algorithm to the problem of workload distribution among placement robots for two reasons. The first reason is that the processing time for each part package differs per placement robot and depends on the parts that are already placed on that robot (since toolbit exchanges may be introduced). Secondly, the algorithm does not incorporate different board types. The distribution of parts may differ greatly per board type. Additionally, since some boards are more important than others (due to having a higher board quantity), the problem becomes multidimensional.

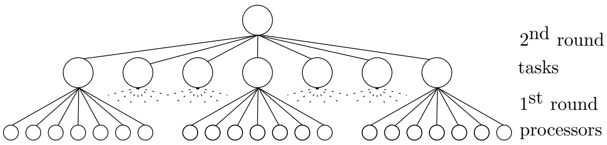


Figure 3.7: The tournament used in the greedy algorithm for computing a robot partition.

We can solve the first of these problems by viewing the standard greedy algorithm for $P \mid\mid C_{\max}$ from a different perspective: During each iteration we perform a two-round tournament among the tasks and processors, as illustrated in Figure 3.7. In the first round of the tournament, we are given a task and need to determine on which processor the task should be placed to obtain the lowest cycle time. In the second round of the tournament, we wish to select the task with the largest processing time if assigned to the processor determined in the first round. We use this concept of a tournament within our greedy algorithm for workload distribution among placement robots. However, to actually perform the tournament we need to have a means to compare different distributions over placement robots, and take multiple board types into account.

Including multiple board types boils down to adding an extra dimension for each board type. In order to compare two different assignments from part packages to placement robots, we introduce the concept of a *workload vector*. A workload vector contains for each board a workload for each robot as depicted in the left column of Figure 3.8. For b different board types, we obtain b different workloads over the robots. From these b different board workloads, we construct a workload vector as illustrated in Figure 3.8. First, we sort each of the board workloads in a non-ascending order, such that the first entry represents the robot with the largest workload. Now we take the weighted sum over all b sorted board workloads, where the weight of board type b is given by its board quantity. The result is a workload vector, where the first element of the vector is equal to the cycle time. Two workload vectors are compared using a lexicographical ordering. That is, first the workload vectors are compared using their cycle times. If the cycle times are the same, we compare the workload vectors using their second element, which indicates how constrained the robots are. The intuition is that a workload vector with a higher second element has less 'space' left: by putting another part on the machine the cycle time will increase earlier than for a workload vector with a smaller second element. If the second element of the workload vector is the same as well we compare the next element, in line with the usual lexicographical ordering.

Our greedy algorithm with workload vectors works analogous to the greedy algorithm for the $P \mid\mid C_{\max}$ problem using the two-round tournament. For each part package, we consider placing it on each placement robot and store the robot yielding the smallest workload vector according to the lexicographical ordering defined above. After repeating this process for all part packages, we take the combination of part package and placement robot yielding the largest workload vector, in line with our intuition of assigning part packages with a large processing time first.

3.2.3 Partial toolbit setup

The greedy algorithm using workload vectors, being a heuristic algorithm, has its downsides. Consider the number of toolbit exchanges of a partition over placement robots as determined by the greedy algorithm. At the start of the execution of the algorithm, all placement robots are empty. Suppose that the first x part packages scheduled by the greedy algorithm are compatible with the same toolbit, and that this toolbit is uncommon among the other part packages. While intuitively these x large part packages should be placed on the same placement robot to avoid toolbit exchanges, the greedy algorithm distributes the toolbits over x different ones, introducing toolbit exchanges on each of them. This example shows that the greedy algorithm can perform quite badly in terms of number of toolbit exchanges, and hence possibly in terms of cycle time as well (many toolbit exchanges usually increase the cycle time, but not necessarily). To attempt to solve this problem, we introduce the concept of *partial toolbit setup*.

A partial toolbit setup consists of an assignment of toolbits to placement robots per board type. Assigning a part package to a placement robot introduces a toolbit exchange whenever

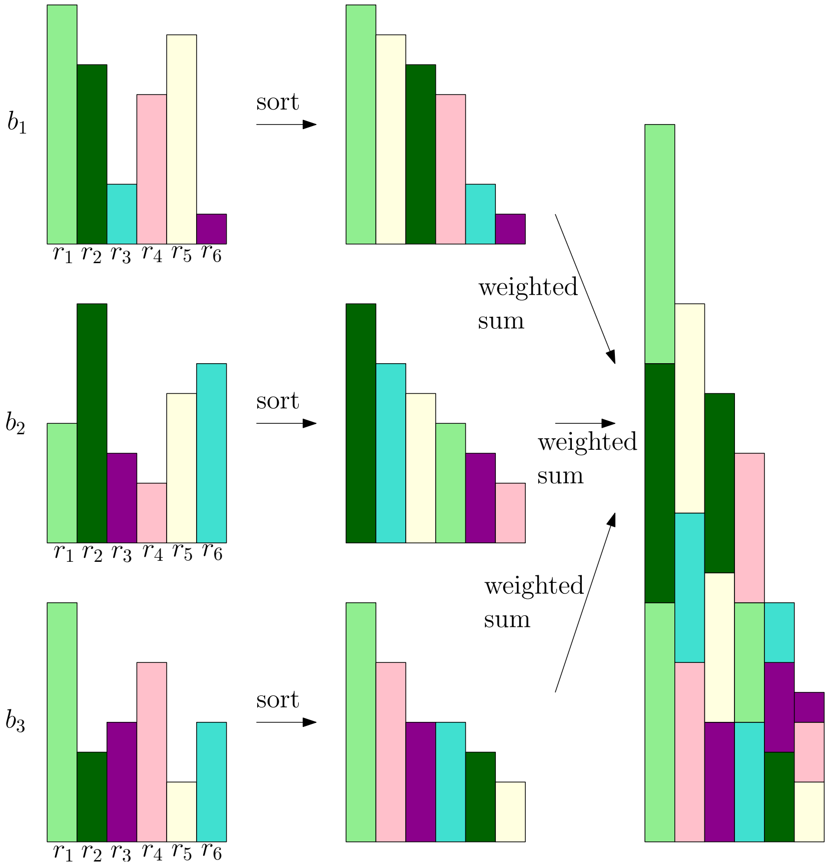


Figure 3.8: The process of constructing a workload vector. The left column contains a workload over the robots for each of the boards (three in this case). The middle column contains these same workloads, now with the robots sorted on non-ascending workload. Finally, the workloads in the middle column are summed together yielding the workload vector on the right.

the part package is not compatible with the toolbit already assigned as usual. In other words, the greedy algorithm prefers to place part packages on placement robots with the same toolbit, until the workload of such a robot becomes so large that introducing a toolbit exchange on another robot yields a lower cycle time.

There are various strategies for computing such a partial toolbit setup. Let T_c and C_t be the set of toolbits compatible with component c and the set of components compatible with toolbit t respectively. We approximate the fraction of placement robots that require toolbit t for board type b by computing

$$\frac{\sum_{c \in C_t} (1/|T_c|)}{|C|}$$

for each toolbit t on board type b, where C is the set of components on board type b. Intuitively, this fraction is the number of compatible toolbit-component combinations, normalized by the number of toolbits compatible with that component divided by the total number of toolbit-component combinations. Now, for each board type we sort the fractions computed for each toolbit. We start by assigning the toolbit with the highest fraction to the corresponding number of placement robots (rounded up), and continue this process until all placement robots have exactly one toolbit assigned to them to obtain our partial toolbit setup.

Although the partial toolbit setup captures the distribution of toolbits over placement robots, the greedy algorithm still has a problem. Suppose our partial toolbit setup contains a certain toolbit t on exactly one placement robot r and all part packages compatible with t are assigned first by the greedy algorithm. After some of these part packages have been assigned, the time required for a toolbit exchange on a different placement robot is smaller than the total time required to place the part packages on r, although introducing a toolbit exchange does not necessarily give the best assignment over the placement robots. As an attempt to solve this problem, we take the idea of a toolbit setup one step further and introduce a *fixed* partial toolbit setup. That is, whenever we need to assign a part package to a placement robot we only have the option to choose placement robots that do not introduce a toolbit exchange, unless this is not possible. This puts more responsibility on the partial toolbit setup: If the partial toolbit setup is not accurate with respect to the actual distribution of toolbits over robots, the workload over placement robots will not be balanced.

3.2.4 Timing model

In this subsection, we describe our timing model. The timing model is assigned with the task of computing an estimate for the cycle time of a partition. While we could use K&S's optimizer software, it takes into account a lot of details yielding a relatively high running time. Our timing model applies a lot of simplification to obtain an estimate much faster. In this subsection we describe the simplifications made in the timing model, while we discuss accuracy and running time in Section 4.1.

Our timing model incorporates the basic operation of picking and placing components, and toolbit exchanges. It does not take into account index steps, since we only assign part packages to placement robots, not to index steps. That is, we assume that there is only one index step for each board type. This simplification causes our estimation of the cycle time to be lower than the actual cycle time for two reasons: We do not take into account transport overhead (that is, the time required to move the transport beam or transport belt) and do not consider idle time of robots properly. By abstracting from index steps we decrease the running time of our timing model, at the expense of a deviation from reality.

The first part of our timing model concerns the travel time of the robot arm. We estimate the time needed to pick up a component from the feeder bank, move the robot arm to the board, place the component and move the robot arm back to the feeder bank. If the component requires the camera for alignment, we add an estimate for the travel time to the camera, the time for alignment and the travel time to the board. A summation of the estimates for each of these processing steps is obtained from K&S' optimizer software, yielding an estimate for the processing time of a component on a placement robot. Note that the location of some components is quite far from the feeder bank, whereas others are relatively close. This difference causes the travel time to be larger for some components than for others.

Similar to the time needed to pick and place a component, we estimate the time required for a toolbit exchange on a placement robot. The time for a toolbit exchange consists of an estimate of the time required to move the robot arm to the TEU, place the toolbit currently equipped in the TEU, pick up the new toolbit and move the arm back to the feeder bank. Since the various toolbits in the TEU are quite close to each other, we ignore the difference in toolbit exchanges among different toolbits. We do take into account the difference between the various placement robots, since the time required to move to the TEU can differ substantially. For instance, the TEU of a CPR is much closer to the placement area than the TEU of an H1 segment. Another reason why exchange times may differ is the difference in movement speeds of the robots.

A placement robot of the iFlex machine has two placement heads instead of just one. This entails that two components can be picked up and placed sequentially. Since this saves travel time of the robot arm compared to the operation on the iX machine, we want to include this in our model. We do so by adapting our estimate for the processing time of a component on a placement robot as described in the previous paragraph. For an iFlex placement robot, we sum the time needed to pick up two components (where we assume that the two components are of the same part), determine alignment of both components through the camera if needed, place the two components and an estimation of the travel time in between. To obtain an estimate for the processing time of a single component, we simply divide the result by two. Certainly this is a simplification from reality. We do not take into account the distance between the two feeders containing the components and the distance between the desired locations on the board. For toolbit exchanges, we ignore the issues regarding two placement heads and assume that we always exchange two toolbits sequentially. Although these simplifications may be crude, they allow us to capture a lot of complex movements in a single estimate for the processing time of a job. This simplifies our model and hence our algorithms by quite a lot, while still yielding estimations of processing times close to reality, as demonstrated in Section 4.1.

3.2.5 Estimating the number of toolbit exchanges

Other than picking and placing components, we may have to perform toolbit exchanges. Since these take a relatively large amount of time and are important factors in evaluating the cycle time of partitions, we cannot ignore toolbit exchanges and need to include them within our timing model. In this subsection we include them by considering the Toolbit cover problem.

Toolbit cover
Given a set P of parts assigned to placement robot r and a set of toolbits $\mathcal{T}$, each compatible with a subset of P, compute a set of toolbits $\mathcal{T}' \subseteq \mathcal{T}$ with minimal cardinality such that all parts in P can be placed using $\mathcal{T}'$.

The problem Toolbit cover is a simplification of the actual toolbit selection problem. Within our timing model we do not take into account in which index step which parts are placed. Although we could, given an assignment from part packages to placement robots and index steps, compute the set of toolbits required on each index step on each placement robot, deciding in which index step a part package should be scheduled is not included within our model. Furthermore, the actual toolbit selection problem requires taking into account a number of domain-specific issues which we abstract from for the purpose of simplicity.

The simplified version of the problem (Toolbit Cover) is an instance of the Set Cover problem: Given a family of sets $\mathcal{A}$ over a universe U of elements, determine a minimum-cardinality subset $X \subseteq \mathcal{A}$ such that $\bigcup_{x \in X} x = U$ (that is, all elements are covered by X). For our specific problem, $U = P$ and $\mathcal{A}$ contains a set for each toolbit containing all parts that can be placed using that toolbit. An example instance is displayed in Figure 3.9.

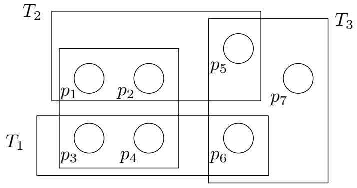


Figure 3.9: A toy instance for Set Cover. A circle denotes an element (part) and a rectangle denotes a set (toolbit).

The Set Cover problem has been studied extensively in the past. Van Rooij and Bodlaender (2008) describe a fixed-parameter tractable (FPT) algorithm solving the Set Cover problem in FPT time. The algorithm works by means of kernelization: First a set of reduction rules is exhaustively applied to reduce the problem instance without changing the solution. If the instance can no longer be reduced, a choice is made by applying a branching rule. This process is iterated until all parts have been covered, yielding a set of toolbits for that particular robot.

We use three reduction rules of the six proposed by Van Rooij and Bodlaender (2008). The other three rules are relatively complicated and can rarely be applied: we expect the computational effort required for determining whether they can be applied does not outweigh the speedup provided by them.

1. *Subset rule*: If the instance contains two toolbits (sets of parts) T and T' such that $T \subseteq T'$, then we can remove T from the instance without changing the solution. This reduction rule is safe since all parts covered by T are covered by T' as well.

2. *Subsumption rule*: Let $T(p)$ be the set of toolbits compatible with part p. If $T(p) \subseteq T(p')$, we can remove part p' from the instance without changing the solution. This reduction rule is safe since any toolbit covering part p also covers part p'.

3. *Singleton rule*: If after exhaustive application of the previous two rules, any toolbit T covering a single part p remains, add T to the toolbit cover and remove the corresponding part from the instance. This reduction rule is safe since T has to be the only toolbit covering part p, otherwise it would have been removed by the subset rule.

The branching rule consists of choosing whether a toolbit T should be included in the toolbit cover or not. In case we put T in the toolbit cover, all parts covered by T are removed from the instance. In case we do not include T in the toolbit cover, we remove the corresponding toolbit from the instance. Instead of choosing a toolbit arbitrarily for our branching rule, we choose the toolbit covering a maximal number of parts motivated by the intuition that covering a relatively large number of parts with one toolbit is often efficient and yields smaller subproblems.

The procedure SolveToolbits contains pseudocode for the Set Cover algorithm, where P is a set of parts and $\mathcal{T}$ is the set of sets of parts representing the toolbits. The procedure returns a set of toolbits with minimal cardinality such that all parts in P can be placed.

```
SolveToolbits($P, \mathcal{T}$)
X = $\emptyset$
while changes occur
    $X$ = ApplySubsumptionRule($P, \mathcal{T}, X$)
    $X$ = ApplySubsetRule($P, \mathcal{T}, X$)
$X$ = ApplySingletonRule($P, \mathcal{T}, X$)
if $X$ covers $P$
    return $X$
else
    $T^* = \arg\max_{T \in \mathcal{T}} |T|$
    $T_{\text{include}}$ = SolveToolbits($P \setminus T^*, \mathcal{T}$) $\cup \{T^*\}$
    $T_{\text{exclude}}$ = SolveToolbits($P, \mathcal{T} \setminus \{T^*\}$)
    return $\arg\min_{T \in \{T_{\text{include}}, T_{\text{exclude}}\}} |T|$
```

Chapter 4

Results

In this chapter we discuss the evaluation of our algorithms. First we discuss the accuracy of the timing model with respect to K&S's optimizer software in Section 4.1, whereafter we discuss the performance of our algorithms in Section 4.2.

All experiments have been performed on a test set consisting of six instances. The characteristics of these instances are displayed in Table 4.1. As a disclaimer, recall that the cycle times in this chapter have been determined by K&S's optimizer software. Since the optimizer is an approximation of the actual cycle time achieved by the machines, some results may be inaccurate when compared to the actual cycle time.

Instance ID	# of components	# of parts	# of shapes	# of board types
a	1500	52	24	3
b	3280	82	27	9
c	3780	32	11	6
d	9280	129	34	10
e	1845	82	20	1
f	3920	75	31	3

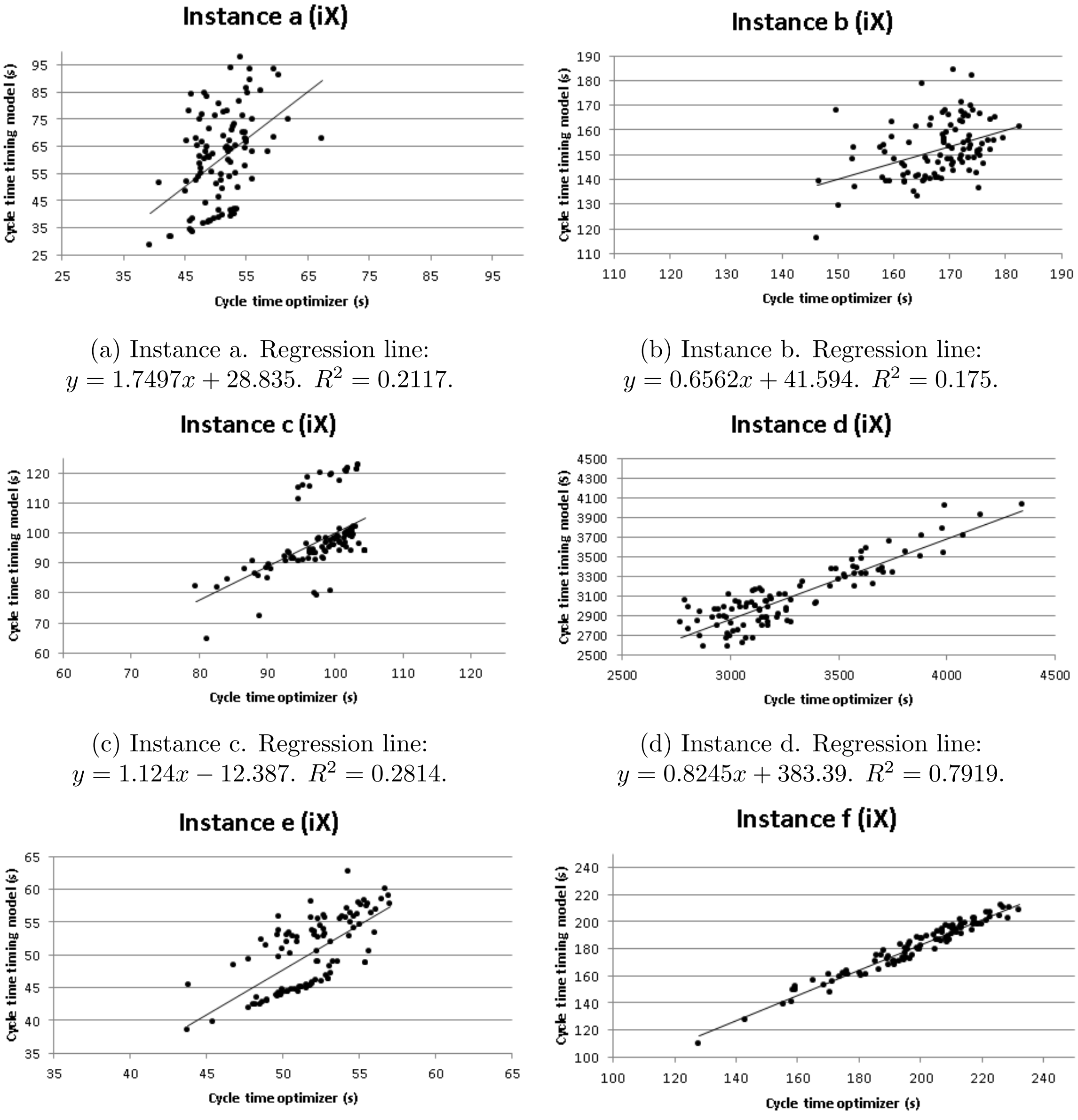
Table 4.1: Characteristics of each of the six instances in the test set.

4.1 Accuracy model of optimizer

In this section we discuss the accuracy of our model of K&S's optimizer software. We distinguish two different components of the model: The timing model (see Section 3.2.4), which assigns a cycle time to a distribution over placement robots, and the greedy algorithm (see Section 3.2.2) determining such an assignment from components to placement robots. We discuss the accuracy of the timing model and the greedy algorithm in Section 4.1.1 and 4.1.2 respectively.

4.1.1 Accuracy timing model

We determine the accuracy of the timing model by generating 100 partitions (using simulated annealing), and evaluating them both with the optimizer and with the timing model, where we use the assignment from components to placement robots as determined by the optimizer. Comparing those yields the graphs in Figure 4.1 for the iX machine and Figure 4.2 for the iFlex machine. We performed a linear regression using ordinary least-squares, and calculated the coefficient of determination R^2 for each of the instances.



(a) Instance a. Regression line: $y = 1.7497x + 28.835$. $R^2 = 0.2117$.

(b) Instance b. Regression line: $y = 0.6562x + 41.594$. $R^2 = 0.175$.

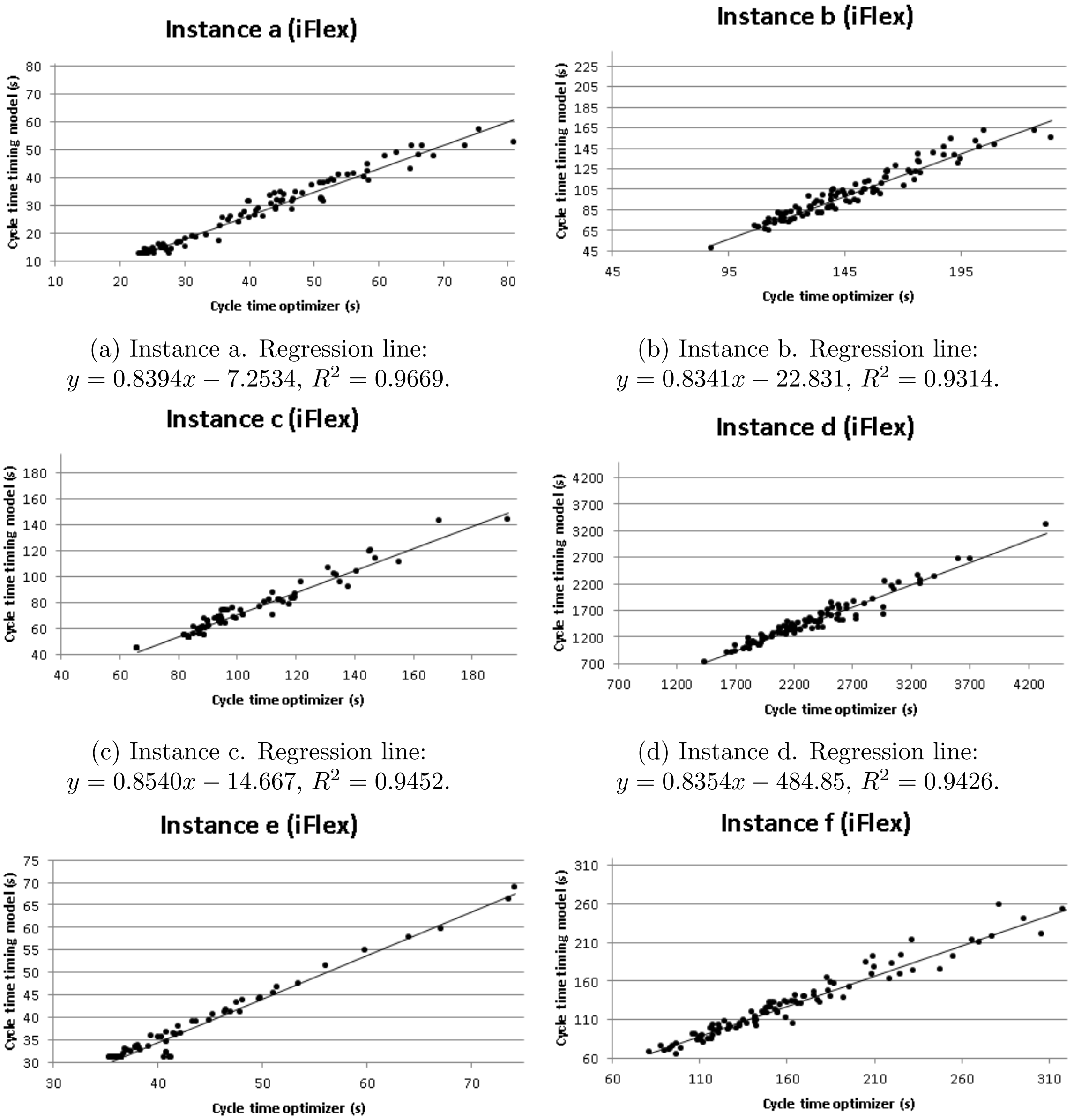
(c) Instance c. Regression line: $y = 1.124x - 12.387$. $R^2 = 0.2814$.

(d) Instance d. Regression line: $y = 0.8245x + 383.39$. $R^2 = 0.7919$.

(e) Instance e. Regression line: $y = 1.3696x - 20.812$. $R^2 = 0.4762$.

(f) Instance f. Regression line: $y = 0.9281x - 3.3992$. $R^2 = 0.9549$.

Figure 4.1: Accuracy timing model with respect to the optimizer for the iX machine.



(a) Instance a. Regression line: $y = 0.8394x - 7.2534$, $R^2 = 0.9669$.

(b) Instance b. Regression line: $y = 0.8341x - 22.831$, $R^2 = 0.9314$.

(c) Instance c. Regression line: $y = 0.8540x - 14.667$, $R^2 = 0.9452$.

(d) Instance d. Regression line: $y = 0.8354x - 484.85$, $R^2 = 0.9426$.

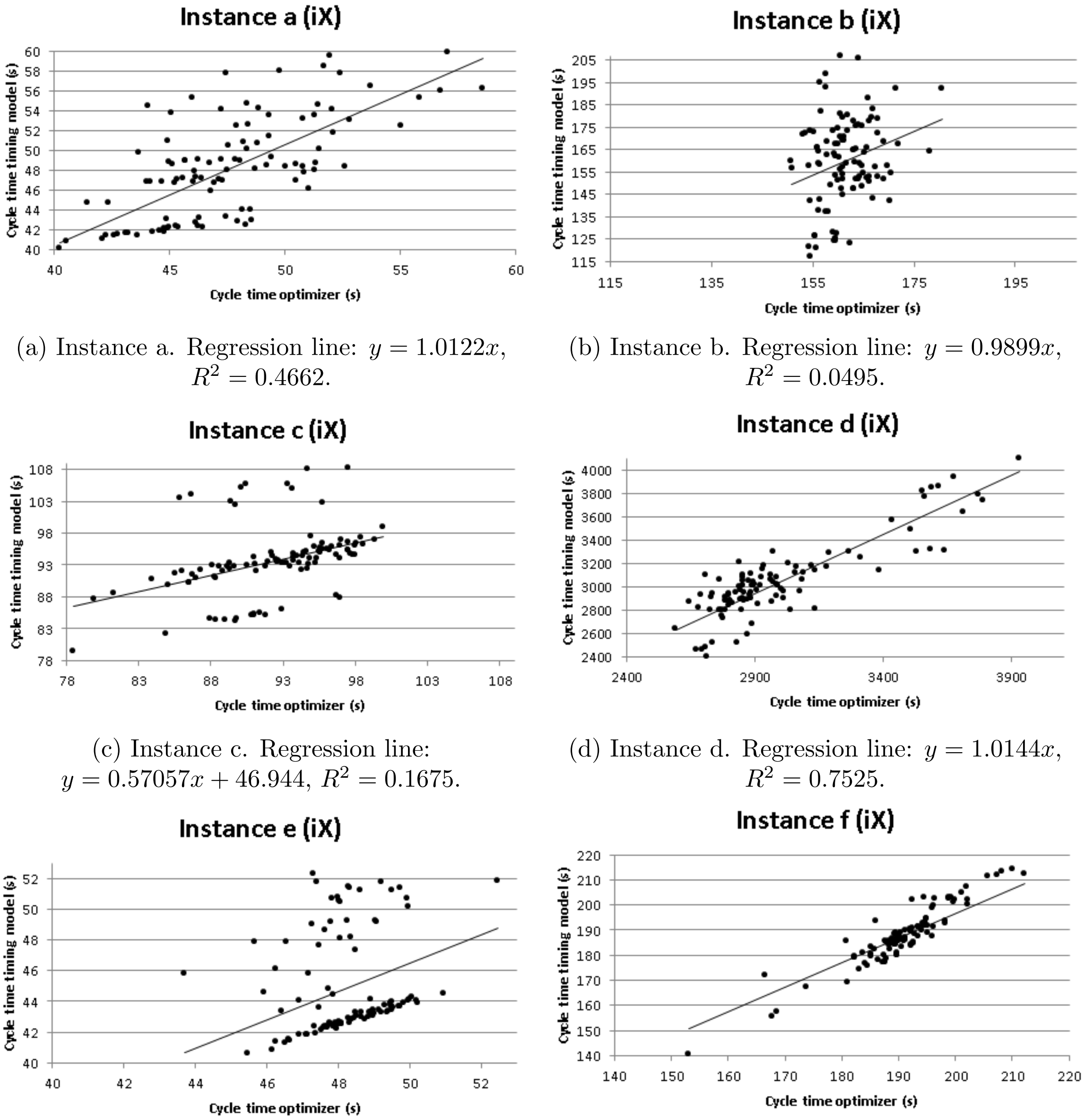
(e) Instance e. Regression line: $y = 0.9667x - 4.207$, $R^2 = 0.9718$.

(f) Instance f. Regression line: $y = 0.7873x + 0.9707$, $R^2 = 0.954$.

Figure 4.2: Accuracy timing model with respect to the optimizer for the iFlex.

Preferably, the data points lie as close to the line $y = x$ as possible. It is to be expected that we do not achieve a perfect fit due to the simplifications we made in the timing model. Some of the simplifications such as measuring fiducials and transport overhead cause an additive deviation. On the other hand, the simplifications we made for the two placement heads on an iFlex robot may cause a multiplicative deviation since we incur an error relative to the number of components. We can overcome these discrepancies by adjusting our estimates in the timing model, yielding an adjusted timing model. We adjust the estimate for the processing time of each component for the multiplicative deviation, and add a factor to each estimate to compensate the additive error. For new instances, this would require either generating adjustments or interpolating the adjustments from other instances, possibly based on the characteristics of the instance. The interpolation of these adjustments is left to future work.

Evaluating the accuracy of the adjusted timing model yields the graphs in Figure 4.3 for the iX machine and Figure 4.4 for the iFlex machine.



(a) Instance a. Regression line: $y = 1.0122x$, $R^2 = 0.4662$.

(b) Instance b. Regression line: $y = 0.9899x$, $R^2 = 0.0495$.

(c) Instance c. Regression line: $y = 0.57057x + 46.944$, $R^2 = 0.1675$.

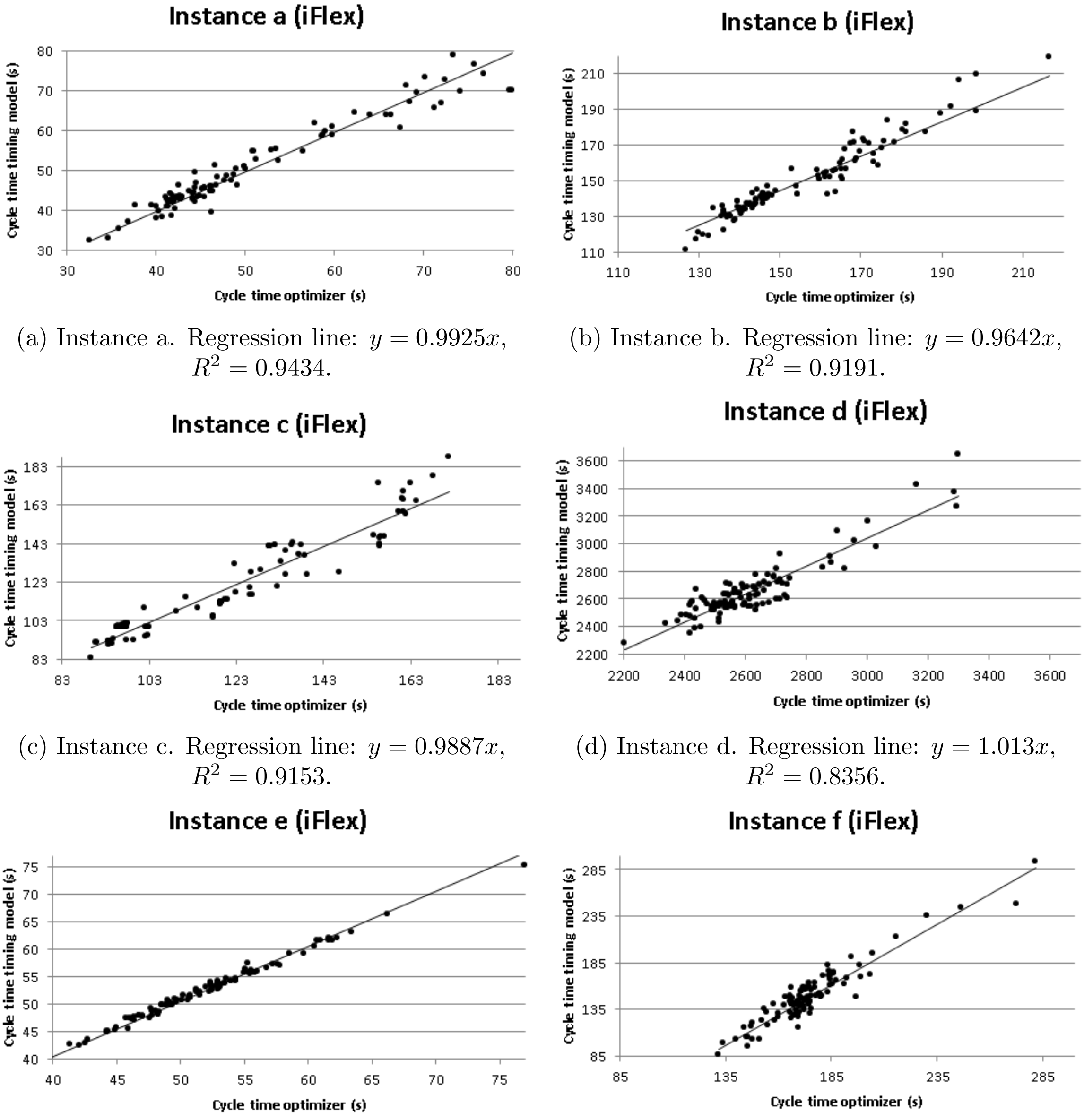
(d) Instance d. Regression line: $y = 1.0144x$, $R^2 = 0.7525$.

(e) Instance e. Regression line: $y = 0.9302x$, $R^2 = -0.007$.

(f) Instance f. Regression line: $y = 0.9841x$, $R^2 = 0.8235$.

Figure 4.3: Accuracy adjusted timing model with respect to the optimizer for the iX machine.

We observe that the slopes of the regression lines for the adjusted timing model for the iX machine are quite close to 1. The variance in the errors (between the regression line and the data points) differs for each instance: For instance d and f the variance is quite low, as also apparent from the values of R^2 that are relatively close to 1. For instance c and e there are “gaps” in the data points, possibly caused by simplifications in the timing model. Since a toolbit exchange takes about two seconds on an iX machine, the discrepancies may be explained by a different number of toolbit exchanges calculated by the timing model and K&S’s optimizer software. For instance a and b, the discrepancy between data and regression line is quite high (as shown by the low value of R^2) and the timing model seems to insufficiently estimate K&S’s optimizer software.



(a) Instance a. Regression line: $y = 0.9925x$, $R^2 = 0.9434$.

(b) Instance b. Regression line: $y = 0.9642x$, $R^2 = 0.9191$.

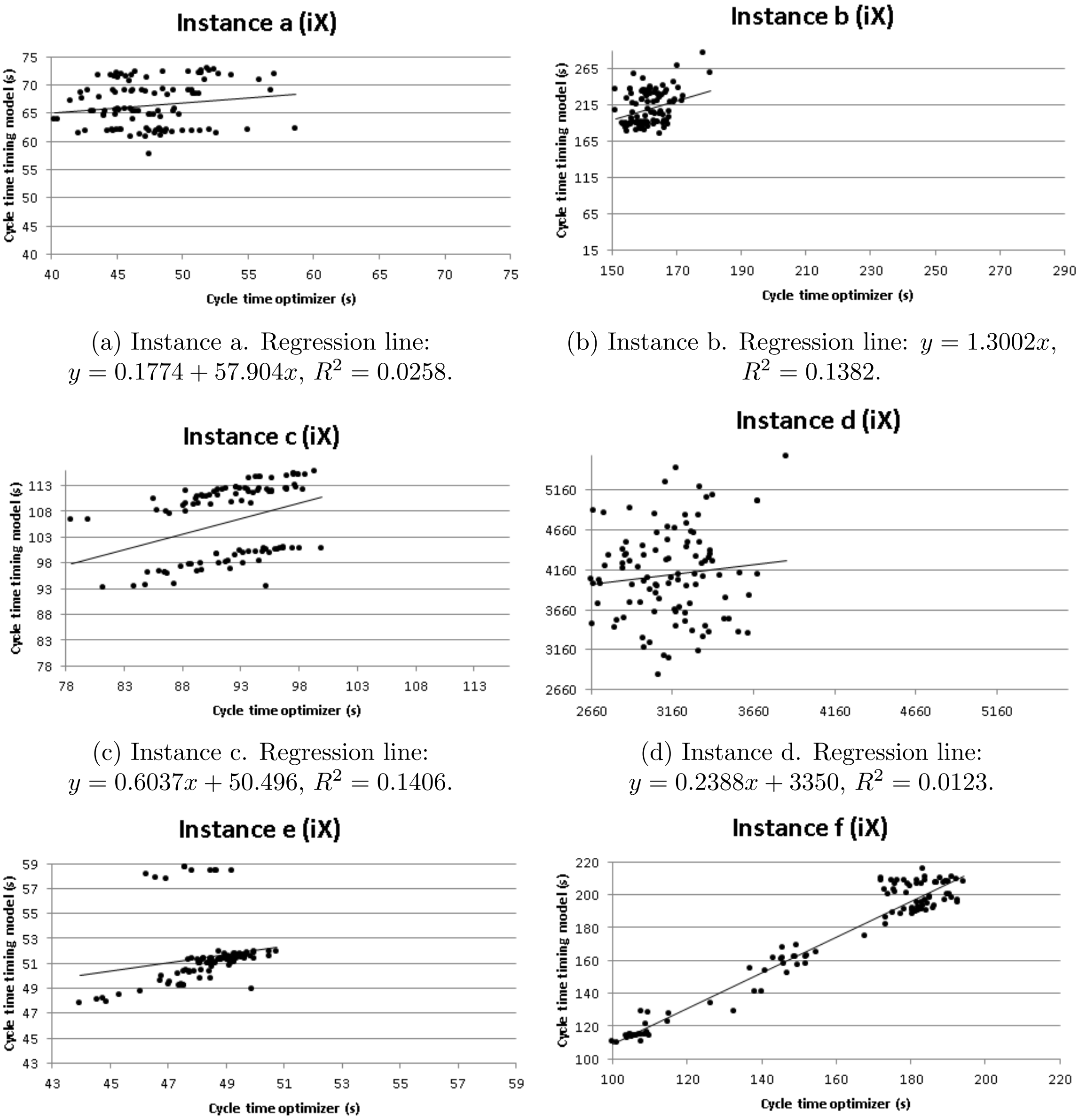
(c) Instance c. Regression line: $y = 0.9887x$, $R^2 = 0.9153$.

(d) Instance d. Regression line: $y = 1.013x$, $R^2 = 0.8356$.

(e) Instance e. Regression line: $y = 1.0071x$, $R^2 = 0.9870$.

(f) Instance f. Regression line: $y = 1.302x - 79.242$, $R^2 = 0.8921$.

Figure 4.4: Accuracy adjusted timing model with respect to the optimizer for the iFlex.



(a) Instance a. Regression line: $y = 0.1774 + 57.904x$, $R^2 = 0.0258$.

(b) Instance b. Regression line: $y = 1.3002x$, $R^2 = 0.1382$.

(c) Instance c. Regression line: $y = 0.6037x + 50.496$, $R^2 = 0.1406$.

(d) Instance d. Regression line: $y = 0.2388x + 3350$, $R^2 = 0.0123$.

(e) Instance e. Regression line: $y = 0.3421x + 34.961$, $R^2 = 0.0307$.

(f) Instance f. Regression line: $y = 1.0855x$, $R^2 = 0.9481$.

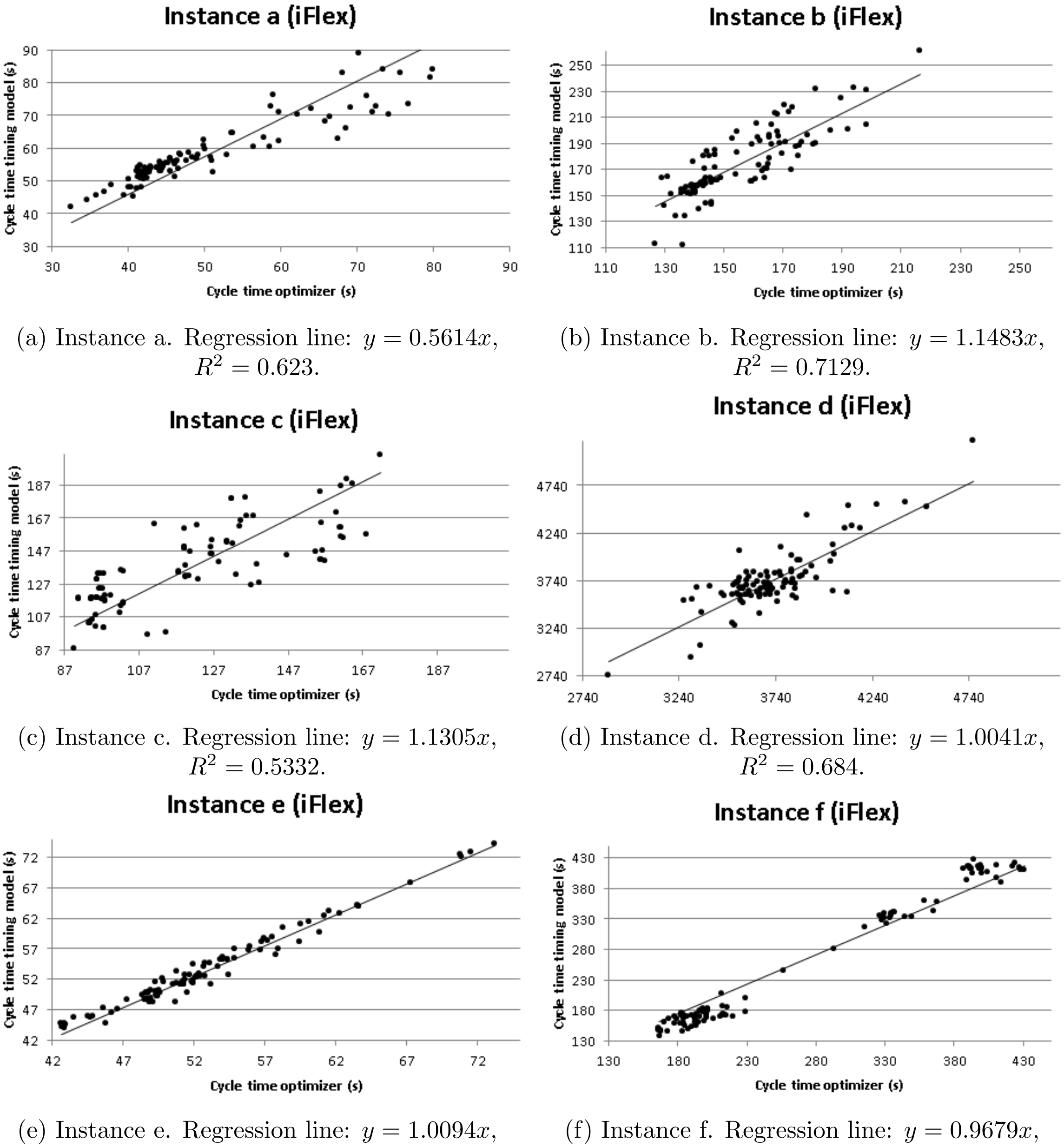
Figure 4.5: Accuracy adjusted timing model with greedy distribution with respect to the optimizer for the iX machine.

As for the iFlex machine, the timing model seems to estimate K&S's optimizer software quite well, as shown both by the slopes of the regression lines and the coefficients of determination being relatively close to 1. This indicates that our simplifications for the iFlex machine are justified. Especially taking into account our simplifications regarding two placement heads, we believe this is a remarkable result.

The running time of the timing model is small compared to the running time of K&S's optimizer software, as can be seen in Table 4.2.

	Instance a	Instance b	Instance c	Instance d	Instance e	Instance f
Optimizer	23526.47	59403.31	67329.96	146913.75	47265.12	132768.54
Timing model	8.58	53.05	20.96	123.20	4.05	22.03

Table 4.2: Average running time (in ms) over 100 instances for the optimizer and the (adjusted) timing model.



(a) Instance a. Regression line: $y = 0.5614x$, $R^2 = 0.623$.

(b) Instance b. Regression line: $y = 1.1483x$, $R^2 = 0.7129$.

(c) Instance c. Regression line: $y = 1.1305x$, $R^2 = 0.5332$.

(d) Instance d. Regression line: $y = 1.0041x$, $R^2 = 0.684$.

(e) Instance e. Regression line: $y = 1.0094x$, $R^2 = 0.9735$.

(f) Instance f. Regression line: $y = 0.9679x$, $R^2 = 0.9590$.

Figure 4.6: Accuracy adjusted timing model with greedy distribution with respect to the optimizer for the iFlex.

4.1.2 Accuracy greedy algorithm

Now that we have compared the timing model with K&S's optimizer software, in this subsection we incorporate our own assignment of components to placement robots as determined by the greedy algorithm. This results in the graphs in Figure 4.5 for the iX machine and Figure 4.6 for the iFlex machine.

For the iX machine, we mainly observe what we expected: We find similar results as those in Figure 4.3, although slightly worse due to incorporation of the greedy algorithm. For instance f the timing model and greedy algorithm quite accuractely model the optimizer, whereas for instance c and e the gaps have become larger. For instance a and b the accuracy seems to be at an inadequate level. Only for instance d we observe a difference compared to the accuracy of the adjusted timing model: Whereas the adjusted timing model quite accurately describes the optimizer, incorporating the greedy algorithm yields undesirable results. We conclude that our greedy algorithm assigns components differently to the placement robots than the optimizer, resulting in the graph in Figure 4.5d.

For the iFlex machine, even with the incorporation of the greedy algorithm the accuracy is quite high. Although the coefficients of determination are lower and the slopes of the regression lines are not as close to 1, for our purposes our model approximates the optimizer accurately well.

The greedy algorithm does cause our model to slow down compared to the timing model, but still a significant speedup is achieved in most instances, as can be seen in Table 4.3.

	Instance a	Instance b	Instance c	Instance d	Instance e	Instance f
Optimizer	24377.65	64680.84	79059.90	146913.75	131833.80	100161.52
Timing model + greedy	2663.969	57010.33	9351.137	8412.265	60765.20	2272.59
Running time improvement	9.15 ×	1.13 ×	8.45 ×	9.04 ×	2.17 ×	44.07 ×

Table 4.3: Average running time (in ms) over 100 instances for the optimizer and the (adjusted) timing model combined with the greedy algorithm.

4.2 Performance Simulated Annealing

In this section we discuss the performance of simulated annealing on the instances in Table 4.1. There are a number of alternate strategies for simulated annealing, such as different neighbourhood functions. We differentiate the following strategies:

- Neighborhood function
  - Move-k
  - Clustering
  - Toolbit clusters

- Evaluation of partitions
    - Optimizer
    - Timing model
- Simulated Annealing parameters
    - Initial temperature
    - Cooling schedule
    - Epoch length
- Partial toolbit setup
    - No partial toolbit setup
    - Partial toolbit setup using estimated number of robots
    - Fixed partial toolbit setup
- Part packages
    - Package size

Evaluating all combinations of these strategies would take too much time. Instead, we choose a 'standard' scenario, choosing a value for each of the dimensions above. Then we change one of the dimensions and evaluate both scenarios. If the latter evaluation gives a better result, we can conclude that the change is positive in that scenario. We leave the analysis of combinations of these scenarios as future work. The standard scenario is the following:

- Neighborhood function: Move-k.
- Evaluation of partitions: Adjusted timing model.
- Simulated annealing parameters: Initial temperature based on average increase, geometric cooling schedule, epoch length 1.
- Partial toolbit setup: Partial toolbit setup using estimated number of robots.
- Part packages: Package size based on average processing time per robot times 0.7.

Table 4.4 displays each of the scenarios that have been tested.

We now evaluate each of these scenarios, varying with the evaluations function (either optimizer or timing model) and the number of evaluations used within the simulated annealing framework.

Based on the results in Table 4.5 and Table 4.6 we conclude that the clustering and toolbit cluster approach do not yield the desired improvements compared to scenario STANDARD. As for the partial toolbit setup, the differences are relatively small, except for instance d. Scenario NO_PTS performs significantly better compared to STANDARD on instance d. The differences for PP_SIZE_NO_CORR are relatively small as well, but in general it performs slightly worse than scenario STANDARD. Introducing an epoch length of five improves the solution for some instances, while providing a worse solution for others.

Scenario ID	Change w.r.t. STANDARD
STANDARD	-
CLUST	Neighborhood function: clustering
TBCLUST	Neighborhood function: toolbit clusters
NO_PTS	Partial toolbit setup: none
FIXED_PTS	Partial toolbit setup: fixed
PP_SIZE_NO_CORR	Part package size: average processing time per robot (no correction with a factor of ε)
EPOCH_LENGTH	Simulated annealing uses an epoch length of 5 evaluations; the total number of evaluations stays the same.

Table 4.4: The different scenarios tested.

Scenario ID	Instance a	Instance b	Instance c	Instance d	Instance e	Instance f
STANDARD	50.852	172.808	122.307	4138.512	49.735	217.015
CLUST	140.624	322.454	165.598	3502.26	71.368	374.578
TBCLUST	137.925	376.919	196.740	3368.649	62.525	265.248
NO_PTS	50.852	172.808	122.543	3628.231	50.925	220.370
FIXED_PTS	50.852	172.808	122.307	4138.512	49.735	217.015
PP_SIZE_NO_CORR	50.852	172.808	131.326	4372.639	51.031	218.230
EPOCH_LENGTH	50.852	181.295	125.258	3181.265	48.326	208.385

Table 4.5: Performance of 1000 evaluations of our model, evaluated with the optimizer. Cycle times are given in seconds.

Scenario ID	Instance a	Instance b	Instance c	Instance d	Instance e	Instance f
STANDARD	61.068	196.7172	97.839	3214.173	49.922	176.290
CLUST	66.100	230.344	98.453	3396.506	43.792	188.842
TBCLUST	85.563	417.888	100.941	6016.929	57.995	203.943
NO_PTS	61.410	220.265	100.189	3048.594	51.023	189.135
FIXED_PTS	61.410	220.265	97.839	3214.173	51.023	189.135
PP_SIZE_NO_CORR	61.068	196.717	97.840	3214.173	49.922	176.290
EPOCH_LENGTH	61.068	201.477	101.017	2926.368	48.951	179.250

Table 4.6: Performance of 1000 evaluations of our model, evaluated with the adjusted timing model. Cycle times are given in seconds.

Using only 100 evaluations instead of 1000, we obtain the results in Table 4.7 when evaluated with the optimizer and Table 4.8 when evaluated with the timing model. As for the scenarios, we generally make the same conclusions as we did for the 1000-evaluations case. That is, CLUST and TBCLUST perform worse than STANDARD, while the solutions provided by NO_PTS and FIXED_PTS lie relatively close to the solution provided by STANDARD in general. Scenarios PP_SIZE_NO_CORR and EPOCH_LENGTH provide mixed results.

As for the impact of the number of evaluations, the results are mixed. In general performing 1000 evaluations provides better results for instance a and b, while yielding worse for instance c and d and roughly equal cycle times for instance e and f. This might be explained by the ‘randomness’ of the timing model. One can observe from Table 4.6 and Table 4.8 that

Scenario ID	Instance a	Instance b	Instance c	Instance d	Instance e	Instance f
STANDARD	55.827	189.512	101.236	2911.557	49.816	219.653
CLUST	211.301	417.115	435.143	3344.063	168.69	573.092
TBCLUST	95.784	202.017	293.66	3470.197	81.46	323.965
NO_PTS	55.827	189.512	101.430	3180.032	50.415	215.512
FIXED_PTS	55.827	189.512	101.236	2911.557	49.816	219.653
PP_SIZE_NO_CORR	49.728	189.512	115.27	4474.704	50.347	196.811
EPOCH_LENGTH	50.192	183.895	106.68	3290.061	53.979	209.567

Table 4.7: Performance of 100 evaluations of our model, evaluated with the optimizer. Cycle times are given in seconds.

Scenario ID	Instance a	Instance b	Instance c	Instance d	Instance e	Instance f
STANDARD	62.925	197.63	99.7497	3431.07	50.1459	222.572
CLUST	74.2734	303.96	116.901	3779.97	49.834	205.822
TBCLUST	75.4678	316.156	103.648	5860.41	45.602	245.514
NO_PTS	64.2554	220.606	103.782	3405.31	54.0745	224.478
FIXED_PTS	49.443	197.63	120.17	3221.66	55.2856	216.492
PP_SIZE_NO_CORR	62.925	197.63	99.7497	3431.07	50.1459	222.572
EPOCH_LENGTH	62.925	197.63	100.616	3337.69	50.8301	220.977

Table 4.8: Performance of 100 evaluations of our model, evaluated with the adjusted timing model. Cycle times are given in seconds.

simulated annealing with 1000 evaluations finds better results than with 100 evaluations when evaluated with the timing model, but only marginally better. Those slightly better solutions may be worse when evaluated with the optimizer, which might explain the worse solutions in Table 4.5 compared to those in Table 4.7.

Scenario ID	Instance a	Instance b	Instance c	Instance d	Instance e	Instance f
STANDARD	47.450	164.873	99.156	2889.219	48.584	184.144
CLUST	84.327	218.007	108.090	3132.335	53.476	265.248
TBCLUST	49.880	505.267	104.404	3268.282	72.189	402.652
NO_PTS	46.402	163.040	99.097	2993.600	48.031	191.885
FIXED_PTS	42.378	185.181	103.425	2835.676	48.530	192.760
PP_SIZE_NO_CORR	46.402	156.396	99.097	2851.863	48.031	182.280
EPOCH_LENGTH	46.381	164.196	98.357	2750.107	48.640	189.353

Table 4.9: Performance of 100 evaluations of the optimizer, evaluated with the optimizer. Cycle times are given in seconds.

In Table 4.9 the performance of simulated annealing using 100 evaluations evaluated with the optimizer is displayed. We observe that in general a better solution is found using the optimizer compared to our model using 100 evaluations, which is in line with our expectations. Since our model is merely an approximation of the actual optimizer, we expect that simulated annealing using the optimizer requires less evaluations to obtain the same solution than our model. Even with 1000 evaluations using our model, the results are worse than those found by the optimizer using 100 evaluations. This means either that we need more evaluations with our model, or our model is not sufficiently precise (causing the 'randomness' in the cycle time of the solutions returned by our model).

As for the different scenarios, the indirect neighboring functions do not give the desired improvements in line with our other results. Scenario NO_PTS gives slightly better results in general, except for instance d. Scenario FIXED_PTS gives mixed results, whereas scenario PP_SIZE_NO_CORR gives the best results overall. Scenario EPOCH_LENGTH performs well on most instances compared to scenario STANDARD, except performing slightly worse on instance e and f.

Scenario ID	Instance a	Instance b	Instance c	Instance d	Instance e	Instance f
Valor MSS Process Preparation	52.263	168.940	105.055	2714.353	51.917	226.582
Best overall	42.378	156.396	98.357	2750.107	48.031	182.280
Improvement	20.44%	7.43%	6.38%	-1.32%	7.49%	19.55%
PP_SIZE_NO_CORR 100 evaluations with optimizer	46.402	156.396	99.097	2851.863	48.031	182.280
Improvement	12.88%	7.43%	5.67%	-5.07%	7.49%	19.55%

Table 4.10: Comparison of performance with respect to Valor MSS Process Preparation. Cycle times are given in seconds.

To conclude our analysis of the algorithms, Table 4.10 displays a comparison of our algorithms with the current system in use, Valor MSS Process Preparation. Both the best overall solution found and the results using the best scenario are displayed.

Chapter 5

Discussion

In this chapter we discuss to which extent the problem under study has been solved and propose ideas and areas for future work.

We have developed a simulated annealing algorithm for load balancing among the iX machine and iFlex machine using the natural neighborhood function of moving k part packages, as well as indirect neighborhood functions. Since evaluating partitions using K&S's optimizer software is relatively slow, we constructed a model simulating the software, by means of a greedy algorithm assigning part packages to placement robots and a timing model evaluating the assignment from part packages to placement robots.

While the timing model computes a cycle time sufficiently fast, the greedy algorithm takes quite some time to compute a solution, reducing the number of evaluations that can be performed within a certain amount of time. Replacing the greedy algorithm by a faster algorithm would increase the number of evaluations. On the other hand, since the assignment from part packages to machines determines the cycle time, the performance of the algorithm assigning part packages to machines should be sufficient as well. Further research towards this tradeoff is required to find a better balancing algorithm.

The performance of the indirect neighboring functions based on clustering is considerably lower than the performance with the natural neighborhood function. Recall that the indirect neighboring functions require either an estimate for a set of part packages on a machine, or an assignment from those part packages to placement robots and an estimate for part packages on placement robots. Currently the greedy algorithm is used for this purpose, providing another reason to replace the greedy algorithm by another algorithm.

For now, our research shows that the techniques used do yield improvements in some cases, but for consistent high-quality solutions these techniques need to be improved or new techniques have to be developed. Our algorithms can be seen as a basis toward a well-performing algorithm.

Our approach does not take all details of the machines into account. In particular, our algorithms largely ignore the existence of index steps. We do assign part packages to placement robots within the greedy algorithm, but do not decide in which index step components are placed. We expect that incorporating index steps both within our algorithms and our timing model would yield a more accurate model of the optimizer. In turn, this would reduce some of the randomness of the results returned by simulated annealing (using an inaccurate model for simulated annealing may cause the final solution to be relatively bad when evaluated using the optimizer).

Recall that the iX machine and iFlex machine inherently differ in a number of aspects: An iFlex machine has two placement heads per robot arm whereas those of the iX machine have only one. Within our algorithms and timing model, we simplified these operations by assuming there is only one placement head and dividing the result by two. From the results on the accuracy of the timing model, it seems this simplification does not withhold us from constructing a sufficiently accurate timing model. However, the complexity introduced by those two placement heads is not captured within the toolbit exchanges as well, where one toolbit exchange can make a relatively large difference in cycle time.

We now propose a number of potential research areas for future work. First of all, more research could be performed for finding a good neighboring function for simulated annealing. The neighborhood function using toolbit clusters has the downside that toolbit clusters only increase in size. Putting a cap on how large a toolbit cluster is allowed to be helps to obtain reasonable solutions, but an additional mutator that decreases the size of those toolbit clusters would reduce the severity of the problem and may yield better results.

As for the partial toolbit setup introduced in Section 3.2.3, one could experiment with starting to place the rarest toolbits instead of the most common ones. If done properly, these rare toolbits are placed on exactly one (or a few) placement robots, which may yield a better partial toolbit setup.

Out of the wide range of optimization techniques, we chose simulated annealing as the framework for our approach. As future work one could try other optimization techniques and see how these perform. These may include stochastic optimization techniques such as genetic algorithms and tabu search. For genetic algorithms, depending on the crossovers used there may be too much variety within the solutions: A crossover of two partitions intuitively does not necessarily yield a better solution. The main problem with tabu search is that it cannot deal with many local optima, while we expect that our optimization landscape contains these local optima. As future work one could implement these methods nevertheless, or use any other stochastic optimization techniques.

Another approach is to use more direct algorithms that compute a partition (either among machines or placement robots) directly, such as our greedy algorithm. These direct approaches could be based on existing algorithms for multi-way number partitioning or an adaption of the Karmarkar-Karp algorithm (Karmarkar & Karp, 1982). Since our problem is multidimensional due to different board types and different index steps, one could incorporate ideas used for vector scheduling and its dual problem vector bin packing in these direct algorithms. Such a direct algorithm can either be used by itself, or within the simulated annealing framework with a clustering neighborhood function replacing the greedy algorithm.

A virtually endless effort can be spent in tuning the simulated annealing algorithm. Examples of parameters that could be tuned are the initial temperature of simulated annealing, the cooling schedule used, the different mutators used and their combination. In general tuning the parameters of simulated annealing is a common problem within the analysis of these algorithms, and good parameters tend to differ across different domains.

To conclude, we have provided the basis for an algorithm distributing the workload among the iX and the iFlex machine. While in some cases the cycle time is better when compared those of the current system, for a consistent high-quality output our techniques either need to be extended or some components need to be replaced by others.

References

Aarts, E., Korst, J., & Michiels, W. (2005). Simulated annealing. In *Search methodologies* (pp. 187–210). Springer.

Bansal, N., Caprara, A., & Sviridenko, M. (2009). A new approximation method for set covering problems, with applications to multidimensional bin packing. *SIAM Journal on Computing*, *39*(4), 1256–1278.

Bansal, N., Vredeveld, T., & van der Zwaan, R. (2014). Approximating vector scheduling: almost matching upper and lower bounds. In *Latin american symposium on theoretical informatics* (pp. 47–59).

Chekuri, C., & Khanna, S. (1999). On multi-dimensional packing problems. In *Soda* (Vol. 99, pp. 185–194).

Davis, E., & Jaffe, J. M. (1981). Algorithms for scheduling tasks on unrelated processors. *Journal of the ACM*, *28*(4), 721–736.

Graham, R. L., Lawler, E. L., Lenstra, J. K., & Rinnooy Kan, A. H. G. (1979). Optimization and approximation in deterministic sequencing and scheduling: a survey. *Annals of discrete mathematics*, *5*, 287–326.

Hariri, A., & Potts, C. N. (1991). Heuristics for scheduling unrelated parallel machines. *Computers & operations research*, *18*(3), 323–331.

Horowitz, E., & Sahni, S. (1976). Exact and approximate algorithms for scheduling nonidentical processors. *Journal of the ACM*, *23*(2), 317–327.

Jansen, K., & Porkolab, L. (2001). Improved approximation schemes for scheduling unrelated parallel machines. *Mathematics of Operations Research*, *26*(2), 324–338.

Johnson, D. S., Aragon, C. R., McGeoch, L. A., & Schevon, C. (1989). Optimization by simulated annealing: an experimental evaluation; part i, graph partitioning. *Operations research*, *37*(6), 865–892.

Johnson, D. S., Aragon, C. R., McGeoch, L. A., & Schevon, C. (1991). Optimization by simulated annealing: an experimental evaluation; part ii, graph coloring and number partitioning. *Operations research*, *39*(3), 378–406.

Karmarkar, N., & Karp, R. M. (1982). *The differencing method of set partitioning* (Tech. Rep.). Technical Report UCB/CSD 82/113, Computer Science Division, University of California, Berkeley.

Kim, D.-W., Kim, K.-H., Jang, W., & Chen, F. F. (2002). Unrelated parallel machine scheduling with setup times using simulated annealing. *Robotics and Computer-Integrated Manufacturing*, *18*(3), 223–231.

Kirkpatrick, S., Vecchi, M. P., & Gelatt, C. D. (1983). Optimization by simmulated annealing. *science*, *220*(4598), 671–680.

Korf, R. E. (2009). Multi-way number partitioning. In *Ijcai* (pp. 538–543).

Lenstra, J. K., Shmoys, D. B., & Tardos, É. (1990). Approximation algorithms for scheduling unrelated parallel machines. *Mathematical programming*, *46*(1-3), 259–271.

Low, C. (2005). Simulated annealing heuristic for flow shop scheduling problems with unrelated parallel machines. *Computers & Operations Research*, *32*(8), 2013–2025.

Martello, S., Soumis, F., & Toth, P. (1997). Exact and approximation algorithms for makespan minimization on unrelated parallel machines. *Discrete applied mathematics*, *75*(2), 169–188.

Meyerson, A., Roytman, A., & Tagiku, B. (2013). Online multidimensional load balancing. In *Approximation, randomization, and combinatorial optimization. algorithms and techniques* (pp. 287–302). Springer.

Pop, P. C., & Matei, O. (2013). A genetic algorithm approach for the multidimensional two-way number partitioning problem. In *International conference on learning and intelligent optimization* (pp. 81–86).

Potts, C. N. (1985). Analysis of a linear programming heuristic for scheduling unrelated parallel machines. *Discrete Applied Mathematics*, *10*(2), 155–164.

Potts, C. N., & Strusevich, V. A. (2009). Fifty years of scheduling: a survey of milestones. *Journal of the Operational Research Society*, *60*(1), S41–S68.

Rekiek, B., Dolgui, A., Delchambre, A., & Bratcu, A. (2002). State of art of optimization methods for assembly line design. *Annual Reviews in control*, *26*(2), 163–174.

van Rooij, J. M. M., & Bodlaender, H. L. (2008). Design by measure and conquer, a faster exact algorithm for dominating set. *Symposium on Theoretical Aspects of Computer Science (Bordeaux)*, 657–668.

Ruml, W., Ngo, J. T., Marks, J., & Shieber, S. M. (1996). Easily searched encodings for number partitioning. *Journal of Optimization Theory and Applications*, *89*(2), 251–291.

Suman, B., & Kumar, P. (2006). A survey of simulated annealing as a tool for single and multiobjective optimization. *Journal of the operational research society*, 1143–1160.

Thompson, W. A., Jr., & Endriss, J. (1961). The required sample size when estimating variances. *The American Statistician*, *15*(3), 22–23.

van de Velde, S. L. (1993). Duality-based algorithms for scheduling unrelated parallel machines. *ORSA Journal on Computing*, *5*(2), 192–205.

Ying, K.-C., Lee, Z.-J., & Lin, S.-W. (2012). Makespan minimization for scheduling unrelated parallel machines with setup times. *Journal of Intelligent Manufacturing*, *23*(5), 1795–1803.