Column generation algorithms for the wafer-to-wafer integration problem

[1], Trivikram Dokka[*1], Yves Crama[†2], Guillerme Duvillié[‡3], and Frits C.R. Spieksma[§4]

[1]Department of Management Science, Lancaster University, United Kingdom.
[2]HEC Management School,University of Liège, Belgium.
[3]Université Libre de Bruxelles, Brussels, Belgium.
[4]Eindhoven University of Technology, the Netherlands.

December 5, 2019

Abstract

We consider the wafer-to-wafer integration problem that arises in the manufacturing of integrated circuits. We propose an exact, state-of-the-art branch-and-price optimization algorithm for this problem, and we derive a price-and-branch heuristic from this algorithm. We have implemented these two algorithms, as well as a simple sequential heuristic algorithms, and we have conducted extensive experiments to test their performance. The results allow us to establish the range of the size of the instances that can be solved to optimality by our branch-and-price algorithm. We also identify ranges of the different instance parameters for which the two heuristics perform relatively well.

1 Introduction

Three-dimensional integration is an important innovation in the manufacturing of stacked integrated circuits. Compared to traditional technology where transistors are connected using only a single layer, three-dimensional integration allows multiple layers of transistors to be bonded. This shortens the length of the connections, thereby improving performance in terms of inter-connect delays and efficiency. For a precise description of the technological intricacies, and the advantages of three-dimensional integration technology, we refer to Garrou et al. [11] and Taouil [21].

[*]Corresponding author. Email: t.dokka@lancaster.ac.uk
[†]Email: yves.crama@uliege.be
[‡]Email: guillerme.duvillie@ulb.ac.be
[§]Email: f.c.r.spieksma@tue.nl.

A key step in the production of three-dimensional integrated circuits is stacking. According to Reda et al. [15], there are three different ways of stacking: wafer-to-wafer, die-to-wafer, and die-to-die. Of these approaches, wafer-to-wafer stacking offers the highest potential for maximizing throughput; moreover, for some applications, wafer-to-wafer is in fact the only option (Reda et al. [15]). However, a serious drawback of wafer-to-wafer stacking is its low production yield. The main motivation of this paper is to study the computational performance of methods that maximize yield in the wafer-to-wafer integration process.

Let us now give an informal description of this yield maximization problem in wafer-to-wafer integration. An individual wafer consists of, say p, dies. As we assume that the wafers are pre-tested before stacking, we know, for each wafer, which of its p dies are functioning (i.e., good), and which are not functioning (i.e., bad). In other words, each wafer can be represented by a binary string of length p, where 0's indicate good dies, and 1's indicate bad dies. Further, we are given m batches of wafers, where each batch consists of n wafers. The objective is to form n wafer stacks by integrating one wafer from each batch in each stack, while maximizing the yield, i.e., while minimizing the total number of bad dies in the resulting stacks. For this reason we call the above problem the wafer-to-wafer integration problem, or WWI in short (see Section 2 for a precise description).

Related literature

WWI has received much attention from various fields: engineering, computer science and operations research. Reda et al. [15], Singh [17], Taouil and Hamdioui [22], Taouil et al. [24], and Verbree et al. [25] propose and experimentally analyze so-called sequential heuristics to optimize the yield. Given an ordering of the batches, these heuristics repeatedly assign wafers from the next batch to the partial wafer stacks formed from previous iterations. Dokka et al. [8] study the worst-case performance of these heuristics. Bougeret et al. [3] focus on the complexity and approximability of WWI.

Reda et al. [15] already observed that WWI is a special case of a broader class of problems, namely (axial) multi-dimensional assignment problems (see also Dokka [6], Dokka et al. [8], Harzi et al [14], etc.). Sequential heuristics for general multi-dimensional assignment problems have been studied by Bandelt et al. [1]. Surveys on this class of problems can be found in Chapter 10 of Burkard et al. [4] and in Spieksma [19].

For the sake of completeness, let us mention that a number of variants of WWI have also been investigated. For example, Singh [18] describes a variant where rotations of wafers are allowed. Clearly, this gives more possibilities to minimize the number of bad dies in the resulting wafer stacks. Another variant concerns a dynamic version of WWI, where the available batches of wafers change over time (dynamic repositories). Taouil et al. [23] investigate how to best maximize yield in a setting where the repositories are replenished over time. In the sequel, we do not consider such variants and we concentrate on the basic formulation of the WWI problem.

Our contribution

In this work, we analyze the performance of an exact, branch-and-price algorithm based on a natural integer programming formulation of WWI. We use column generation to solve the linear programs that arise, and we investigate the corresponding pricing problem. Based on this column generation algorithm, we further design a so-called price-and-branch heuristic which only uses the variables that are present with a positive value in the optimal solution of the linear programming relaxation of WWI. We also consider a sequential heuristic that can be seen as the standard method for solving instances of WWI in practice. All these algorithms are implemented, and we perform a computational study on different classes of randomly generated instances to assess how these algorithms fare with respect to quality of the solution found and computation time needed. By doing so, we are able to establish the boundaries of the size of the instances that can be solved exactly by our state-of-the-art branch-and-price algorithm. In addition, we are able to assess the quality of the solutions found by the heuristic approaches, by comparing them to the optimal value of the linear programming relaxation.

The paper is organized as follows. In Section 2, we give a precise definition of WWI and we present its integer programming formulation. Section 3 describes the branch-and-price algorithm in detail. Sections 4 and 5 introduce the price-and-branch heuristic and the sequential heuristic. We provide details about the generation of the instances in Section 6. The outcome of applying all algorithms on the generated instances is extensively reported in Section 7. We conclude in Section 8.

2 The wafer-to-wafer integration problem

2.1 Problem statement

We now formally state the *wafer-to-wafer integration problem.* The input of the problem is defined by m disjoint sets $V_1, \ldots, V_m$, where each set V_k $(1 \leq k \leq m)$ contains the same number n of p-dimensional binary vectors. Each binary vector models a wafer, and each set V_k stands for a batch of wafers. All wafers in a batch are meant to be identical, except for occasional (and unwanted) defects. We sometimes refer to components of a binary vector as *positions.* The *cost function* $c(\cdot) : \{0,1\}^p \to \mathbb{Z}_+$ associates with each p-dimensional vector the sum of its components (that is, its Hamming weight): $c(u) = \sum_{\ell=1}^{p} u_\ell$.

Let $K = V_1 \times \ldots \times V_m$. A *feasible m-tuple* is an m-tuple of vectors $(u^1, u^2, \ldots, u^m) \in K$ (representing a stack of m wafers), and a *feasible assignment* for K is a set A of n feasible m-tuples such that each element of $V_1 \cup \ldots \cup V_m$ appears in exactly one m-tuple of A. We define the component-wise maximum operator $\vee$ as follows: for every pair of vectors $v, w \in \{0,1\}^p$,

$$v \vee w = (\max(v_1, w_1), \max(v_2, w_2), \ldots, \max(v_p, w_p)).$$

Given a feasible m-tuple $a = (u^1, u^2, \ldots, u^m)$, we define the *representative vector* of a as $v(a) = u^1 \vee u^2 \vee \cdots \vee u^m$.

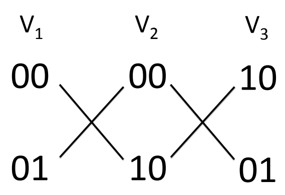


Figure 1: A WWI-3 instance with $m = 3, n = p = 2$

Now, the cost of a feasible m-tuple $a = (u^1, \ldots, u^m)$ is defined as the cost of its representative vector, i.e., with a slight abuse of notations, $c(a) := c(v(a)) := c(u^1 \vee \ldots \vee u^m)$ (it represents the number of defective dies in the stack of wafers $(u^1, \ldots, u^m)$). The cost of a feasible assignment A is the sum of the costs of its m-tuples: $c(A) = \sum_{a \in A} c(a) = \sum_{(u^1,\ldots,u^m) \in A} c(u^1 \vee \ldots \vee u^m)$.

With this terminology, the *wafer-to-wafer assignment problem* (WWI) is to find a feasible assignment for K with minimum cost.

Example 1. *An instance of WWI with $m = 3, n = p = 2$ is displayed in Figure 1. The optimal solution of the instance, of value equal to 2, is achieved by assigning the first vector of V_1, the second vector of V_2, and the first vector of V_3 to the same triple, thus producing the representative vector $(1, 0)$ with cost $c(1, 0) = 1$; the remaining three vectors form a second triple with cost $c(0, 1) = 1$. The cost of this optimal assignment is 2.*

2.2 Integer programming formulation

As previously stated, WWI can be viewed as a multi-dimensional axial assignment problem [19]. This leads to a straightforward set covering formulation of WWI, described as follows.

For each feasible m-tuple $a \in K$ and $u \in V_k$ ($k \in \{1, \ldots, m\}$), we write $u \in a$ if u is the projection of a on V_k, that is, if a contains the vector u from V_k. We define the decision variables x_a, for all $a \in K$, where $x_a = 1$ if m-tuple a is selected in the assignment, and $x_a = 0$ otherwise. The set covering formulation (AIP) of WWI is now:

$$\text{(AIP)} \qquad \min \sum_{a \in K} c(a)\, x_a \tag{1}$$

$$\text{subject to} \sum_{a: u \in a} x_a \geq 1 \qquad \forall u \in \bigcup_{k=1}^{m} V_k \tag{2}$$

$$x_a \in \{0, 1\} \qquad \forall a \in K. \tag{3}$$

The set covering constraints (2) ensure that each vector u is in at least one m-tuple. In view of the objective (1) and of the structure of the problem, each vector will actually be covered exactly once in some optimal solution of AIP. (We could alternatively set up a set partitioning formulation, where (2) are replaced by equality constraints. But the continuous relaxation of the set covering formulation has a better numerical behavior [2].)

The number of variables and number of constraints of AIP are n^m and mn, respectively. As we will see in Section 7, the linear relaxation of AIP is rather tight, but the exponential size of the formulation prevents the solution of large scale instances, due in particular to heavy memory requirements. Several compact, polynomial-size integer programming formulations of WWI have been proposed to address this drawback, either for the general case (see, e.g., [6–8]) or for the case of fixed dimension p (see, e.g., [3,8,9]). However, preliminary experiments revealed that these formulations are weak and do not form the basis of competitive algorithms, even when $m = 3$. These observations motivate the study of more efficient approaches based on formulation AIP, to be developed in subsequent sections.

3 An exact branch-and-price algorithm

In this section we propose an exact branch-and-price (B&P) algorithm for the solution of AIP. B&P is a variant of branch-and-bound: branching on binary variables produces an enumeration tree where each node is associated with a restriction of the original problem, and a column generation algorithm is used in order to solve the linear relaxation of the restricted problem associated with each node (see, e.g., [2]). We discuss next the main ingredients of our implementation.

3.1 Branching

We rely on a well-known branching strategy proposed by Ryan and Foster [2,16]. Suppose that the solution of the LP-relaxation of AIP is fractional. The classical branching rule that forces a fractional variable x_a to take value either 1 or 0 (and thus forces the corresponding m-tuple to be respectively part of, or excluded from, the solution) leads to a very unbalanced enumeration tree. On the other hand, when the optimal solution is fractional, it can be shown that there exists a pair (u, v), $u \in V_i, v \in V_j, i \neq j$ such that

$$0 < \sum_{a: u \in a, v \in a} x_a < 1.$$

Then, the Ryan-Foster branching rule forces, in one branch, $u \in V_i$ and $v \in V_j$ to be part of the same m-tuple (branch 1), and, in the other branch, $u \in V_i$ and $v \in V_j$ to be part of two different m-tuples (branch 2). To fix our notations, let us generically denote by F the set of pairs of vectors that are forbidden to be part of the same m-tuple in a given node of the tree, i.e., $F = \{(w, w') |\ w, w' \in \bigcup_{k=1}^{m} V_k,$ and w, w' cannot be in a same m-tuple$\}$. (Initially, F only contains those pairs (w, w') such that w and w' are in a same set V_k.) Then, to deal with branch 1, we set $F := F \cup \{(w, v) : w \in V_i \setminus \{u\}\} \cup \{(u, w') : w' \in V_j \setminus \{v\}\}$: this guarantees that any generated m-tuple which contains u must also contain v, and vice versa. In branch 2, we set $F := F \cup \{(u, v)\}$, thereby prohibiting vectors u and v to be in a same m-tuple.

Thus, in any node of the tree, we are actually solving a modified version of AIP, to be denoted rAIP in the sequel, where the set of feasible assignments is restricted to a subset $\overline{K} \subseteq K$, with $\overline{K} = \{a \in K : \text{ for all } u, v \in a, (u,v) \notin F\}$. A general formulation of rAIP is

$$\text{(rAIP)} \qquad \min \sum_{a \in \overline{K}} c(a)\, x_a \tag{4}$$

$$\sum_{a : u \in a} x_a \geq 1 \qquad \forall u \in \bigcup_{k=1}^{m} V_k \tag{5}$$

$$x_a \in \{0,1\} \qquad \forall a \in \overline{K}. \tag{6}$$

3.2 Column generation

In the branch-and-price framework, the linear relaxation of rAIP is solved by column generation at each node of the enumeration tree. An arbitrary iteration of the column generation algorithm starts with a master problem (MP) of the form

$$\text{(MP)} \qquad \min \sum_{a \in S} c(a)\, x_a \tag{7}$$

$$\sum_{a : u \in a} x_a \geq 1 \qquad \forall u \in \bigcup_{k=1}^{m} V_k \tag{8}$$

$$0 \leq x_a \leq 1 \qquad \forall a \in S, \tag{9}$$

where $S \subseteq \overline{K}$ is the subset of m-tuples indexing the variables (columns) currently included in MP. Let us denote by λ_u, $u \in \bigcup_{k=1}^{m} V_k$, the dual variables associated with inequalities (8). Given an optimal solution of MP, the pricing problem consists in deciding whether there exists a variable x_a ($a \in \overline{K}$) with negative reduced cost, that is, whether there exists an m-tuple $a \in \overline{K}$ such that

$$\sum_{u \in a} \lambda_u - c(a) > 0. \tag{10}$$

This pricing problem can be formulated as an integer program. We introduce binary variables s_u, indicating whether vector $u \in \bigcup_{k=1}^{m} V_k$ is selected in the m-tuple a, and binary variables q_ℓ indicating whether at least one of the vectors selected in the m-tuple has a 1 in position ℓ ($1 \leq \ell \leq p$). We then obtain the following integer programming formulation of the pricing problem:

$$\text{(PPIP)} \qquad \max \sum_{k=1}^{m} \sum_{u \in V_k} \lambda_u s_u - \sum_{\ell=1}^{p} q_\ell \tag{11}$$

$$\sum_{u \in V_k} s_u = 1 \qquad \forall k = 1, \ldots, m \tag{12}$$

$$\sum_{u \in V_k} u_\ell s_u \leq q_\ell \qquad \forall \ell = 1, \ldots, p,\ k = 1, \ldots, m \tag{13}$$

$$s_u + s_v \leq 1 \qquad \forall (u,v) \in F \tag{14}$$

$$s_u, q_\ell \in \{0,1\} \qquad \forall u \in \bigcup_{k=1}^{m} V_k,\ \ell = 1,\dots,p. \tag{15}$$

The optimal value of PPIP is positive if and only if the answer to the pricing question is yes.

Clearly, the pricing problem (and hence, PPIP) can be solved by a straightforward enumerative algorithm with complexity $O(pn^m)$ (simply generate each feasible m-tuple). On the other hand, it is unlikely that the pricing problem can be solved by an exact algorithm with complexity polynomial in m, n and p. To see this, consider the special case of PPIP that arises when (i) $F = \emptyset$ (hence, constraints (14) disappear), and (ii) $\lambda_u = 0$ for all u. The resulting problem is to find an m-tuple a with minimum cost $c(a)$. The decision version of this problem was shown in [3, 10] to be NP-complete.

3.3 Implementation

We have used the SCIP 5.0.1 framework to implement a branch-and-price algorithm for WWI [12, 13]. Even though SCIP is single-threaded when driving the branch-and-price process, if offers great modularity in several stages of the process. In particular, it allowed us to implement a multithreaded algorithm to solve the pricing problem, as we now explain.

In a first attempt, the pricing problem was solved by simply passing the PPIP formulation to the CPLEX ILP solver. It turned out that the large amount of variables in PPIP (in particular, for large values of p) leads to poor performance of the ILP solvers and thus to prohibitive running times for the branch-and-price algorithm. The choice has therefore been made to solve the pricing problem by exhaustive enumeration: that is, we successively generate all feasible m-tuples a and, for each of them, we verify whether the inequality (10) is satisfied. While this naive algorithm obviously has exponential $O(pn^m)$ complexity, it nevertheless offers interesting opportunities:

- it can be easily parallelized and optimized,
- it offers great adaptability, no matter the adopted branching rule,
- it remains tractable for reasonably small values of m.

We take advantage of the enumeration order to parallelize and optimize the pricing process. Without going into all details of the implementation, note that an m-tuple can be encoded as a word of length m on an alphabet with n symbols: if each of $z_1, \dots, z_m$ is a symbol in $\{1, \dots, n\}$, then the word $(z_1, \dots, z_m)$ encodes the feasible m-tuple containing the z_i-th vector in set V_i (where each set V_i is arbitrarily ordered).

Example 2. *For an instance of WWI with $m = 3, n = 7$, the word $(5, 2, 6)$ encodes the triple consisting of the fifth vector of V_1, the second vector of V_2, and the sixth vector of V_3.*

The m-tuples can then be enumerated according to the lexicographic order $<_L$:

$$(z_1, \ldots, z_m) <_L (z'_1, \ldots, z'_m) \text{ if and only if, for some index } i,$$
$$z_1 = z'_1, z_2 = z'_2, \ldots, z_{i-1} = z'_{i-1}, \text{and } z_i < z'_i.$$

To take advantage of parallelization, the enumeration of the n^m m-tuples is split into n enumerations of n^{m-1} $(m-1)$-tuples, one for each vector of the set V_1 (that is, for each possible value of z_1). These n jobs are dispatched among the different threads of the pool.

Moreover, the representative vector $v(a)$ of each m-tuple a is computed dynamically during the enumeration process, in order to reduce the memory requirements. Indeed, due to the lexicographic enumeration order, all the m-tuples z sharing the same initial prefix $(z_1, \ldots, z_{m-1})$ are successively generated. It is therefore possible to compute the representative vector of the $(m-1)$-tuple associated with this prefix only once for all m-tuples sharing the prefix. Such a representative vector is kept temporarily into memory during the enumeration of these m-tuples. All these "tricks" considerably reduce both memory consumption and computation.

In the initial iterations of the column generation process, it may happen that very many columns with negative reduced cost are identified. In an attempt to limit the size of the master problem, therefore, the number of columns each thread can add to the restricted master problem is limited by an empirically defined threshold (namely, 100 columns in our experiments). Furthermore, we prioritize and we add first those columns with the most negative reduced costs, that is, with the largest possible value of the left-hand side of inequality (10).

Finally, let us remark that the parallelization of the enumeration process aims at reducing the clock wall time of the branch-and-price algorithm, but does not reduce the number of generated m-tuples. On the other hand, parallelization requires additional overhead (CPU time) to manage the job pool, to thread dedicated memory, and even to add columns to the formulation. For these reasons, both the total CPU time (including overhead) and the clock wall time of the solution process will be mentioned, whenever relevant, in our discussion of computational results in Section 7.

4 A price-and-branch heuristic

Even though a branch-and-price algorithm can reduce the memory requirements imposed by the AIP formulation, our computational experiments show that its capability to solve large instances to optimality remains limited. In this section, we propose a heuristic price-and-branch (P&B) algorithm to address this limitation.

4.1 Principle

The basic principle of the price-and-branch algorithm has been used by many authors and appears to be part of the "folklore" of column generation approaches. It consists, first, in relying on the pricing strategy of branch-and-price to solve the linear relaxation of AIP to optimality or, in some variants, to near-optimality. When the column generation process stops, it has produced a specific instance of the master problem (7)-(9) (associated with a subset of columns S), that we call MP*. Now, it is intuitively tempting to assume that MP* is a tight approximation of AIP, meaning that it contains many columns (i.e., m-tuples) that could be used in an optimal solution of AIP (i.e., in an optimal assignment). Reinstating the constraints $x_a \in \{0, 1\}$ for all $a \in S$ in MP* leads to an integer programming problem that we call IMP*. Solving IMP* is a heuristic strategy which cannot guarantee optimality, but which is likely to lead to a good solution of WWI.

The difficulty of solving the IMP* depends, among other factors, on the number of columns generated in the pricing phase. Adding many columns typically improves the quality of the model, but may result in large running times during the branching phase due to the size of the linear programs to be solved. On the other hand, adding too few columns increases the likelihood to miss the optimal solution of AIP. To deal with these issues we adopt the following strategy:

- **Initial columns:** We start with n tuples which form a feasible assignment. Several possibilities offer themselves for this initial set. A natural choice is to define the i-th m-tuple to include the i-th vector from each set, for $i = 1, \ldots, n$. Another possible choice is to include in the initial formulation the m-tuples produced by a heuristic solution of WWI, such as the SHH solution described in Section 5. In our computational experiments, this choice did not significantly affect the performance of the algorithm. (We report on the results obtained with the first alternative here above.)

- **Pricing phase:** As in the branch-and-price algorithm, the pricing problem is solved by enumeration over all m-tuples, that is, we check for each $m-$tuple whether inequality (10) is satisfied. In each round of pricing we only add a prespecified number of columns (300 columns in our experiment). This pricing strategy allows us to solve the LP relaxation of AIP without massive increase of the running time. This phase ends with a formulation MP* of the master problem.

- **Branching phase:** In this phase, we solve the IP restriction of MP*, that is, IMP*. For this purpose, we first implemented a branching algorithm based on the Ryan-Foster rule, as in the branch-and-price algorithm. However, it turned out that simply launching the branch-and-cut IP solver of CPLEX on IMP* leads to the same result in much shorter time. Hence, in Section 7, we only report the results obtained by using the CPLEX IP solver in the P&B solving phase.

5 Sequential heavy heuristic

We present here a simple sequential matching heuristic which provides a baseline for the estimation of our algorithmic results. This class of heuristics has been investigated in [1, 5] for related multi-dimensional assignment problems, and has been more specifically proposed in [15] for the WWI problem. Its worst-case performance has been analyzed in [9]. The idea of sequential matching heuristics is to extend a partial solution, obtained by solving the bipartite matching problem between any two sets V_i, V_j, by solving again a bipartite matching problem between the set of vectors corresponding to the partial solution and another set V_k. This process is repeated until all sets have been matched. Let us define the *weight* of a set V_i as the total number of 1's of the vectors contained in the set, that is: $\text{weight}(V_i) = \sum_{u \in V_i} c(u)$. It was observed by [17] that the performance of sequential heuristics is improved when the sets V_i are ordered by non-increasing weight. Following [9], we call this variant *sequential heavy heuristic* (SHH). We describe it more formally in Algorithm 1.

Algorithm 1 Sequential heavy heuristic (SHH)

```
reorder $V_1, \dots, V_m$ so that $\text{weight}(V_i) \geq \text{weight}(V_j)$ when $i < j$;
let $H_1 := V_1$;
**for** $i = 2$ to $m$ **do**
  solve a bipartite matching problem between $H_{i-1}$ and $V_i$ based on the costs $c(u^1 \vee \dots \vee u^{i-1} \vee v)$, for all $(u^1, \dots, u^{i-1}) \in H_{i-1}$ and $v \in V_i$; let $H_i$ be the resulting assignment for $V_1 \times V_2 \times \dots \times V_i$;
**end for**
output $H_m$.
```

6 Instances

Let us now describe the instances on which the experimental study is based. We consider two classes of instances (uniform and negative binomial), with the first one having three subclasses of different sparsity (percentage of 1's).

Uniform: There are three subclasses of instances in this class: sparse (US), very sparse (UVS), and ultra-sparse (UUS). Each of these subclasses contains random instances generated using a Bernoulli distribution, where the probability of an element being one (failure probability of a die) is given as $p_{US} = 0.10$ for the instances in US, $p_{UVS} = 0.05$ for UVS, and $p_{UUS} = 0.01$ for UUS, respectively. The state of each die is independent of the state of the other dies.

Negative Binomial (NB): The classical negative binomial distribution is frequently used in the literature (see [15, 20]) to generate defect wafer maps. The main idea here is to reproduce the clustering of defects that is often observed in practice. Many authors argued that this effect is best captured by the negative binomial model. Inspired by these arguments, our instance generation procedure for this class of instances is as follows:

- Consider each vector (wafer) as a rectangular block, as in Figure 2a. For example, when $p = 100$, the vector can be arranged as a 10×10 rectangular block.

- Divide this block into sub-blocks of identical size sb. For example, a 10×10 block may be divided into four 5×5 sub-blocks, each containing $sb = 25$ elements (dies).
- Choose a sparsity parameter β, which represents the maximum number of defects to be generated in each sub-block. In our instances we used $\beta = 7$, meaning that each block contains at most 7 elements equal to 1. As an illustration, in Figures 2a and 2b, we compare a vector (wafer) generated with $\beta = 7$ and $\beta = 14$, respectively.
- Let $b(i)$ be the number of defects to be placed in sub-block i, where $b(i)$ is uniformly generated between 0 and β.
- In order to generate $b(i)$ defects (1's) in block i, we draw $b(i)$ integers, say $k_1, k_2, \ldots, k_{b(i)}$ in the interval $[0, sb-1]$, from the negative binomial distribution with success probability p_{NB}; that is, each k_j is the number of failures before a success in a sequence of Bernoulli draws with probability p_{NB} (values of k_j larger than or equal to sb are discarded). In our experiments, we used $p_{NB} = 0.152$.
- The integers $k_1+1, k_2+1, \ldots, k_{b(i)}+1$ indicate the positions where the defects are placed in sub-block i.

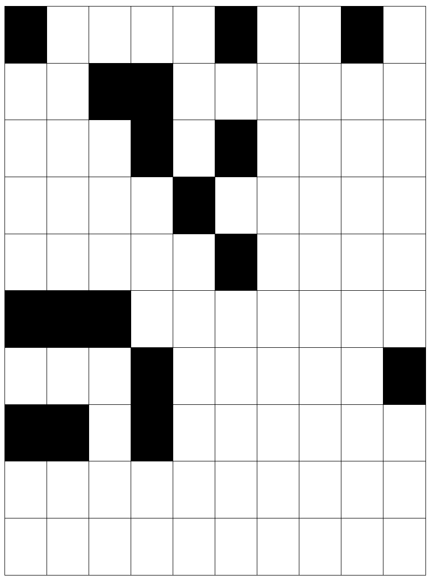
We observed empirically that the choice of parameters $\beta = 7$ and $p_{NB} = 0.152$ leads to instances where the probability of an element being one (failure probability of a die) is roughly 0.15.

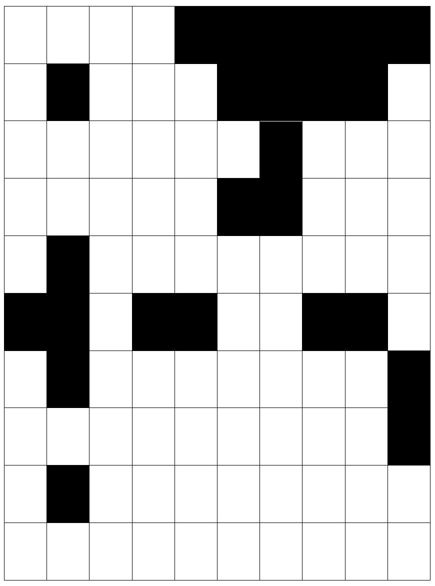
For each combination of parameters $m \in \{3, 5, 7\}$, $n \in \{15, 25, 35\}$ and $p \in \{100, 200, 400, 800\}$, we generate 10 instances in each subclass US, UVS, UUS, and NB. We thus have a total of at most 1440 instances (as we will see in the next section, not all combinations of parameters have been considered in the experiments). Hereafter, we use the notation US(m, n, p) (resp., UVS(m, n, p), UUS(m, n, p), NB(m, n, p)) for the set of US (resp., UVS, UUS, NB) instances with parameters m, n, p. We also use the wildcard character $*$ to consider broader sets of instances where one or more parameters are not fixed. For instance, the set NB$(*, j, *)$ includes all negative binomial instances with $m \in \{3, 5, 7\}$, $n = j$, and $p \in \{100, 200, 400, 800\}$.

7 Computational results

7.1 Branch-and-price algorithm

In this section, we focus on the computational performance of the branch-and-price exact algorithm described in Section 3. The experiments have been conducted on an Intel(R) Xeon(R) CPU E5-2630 v3 2.40GHz with 16 cores. We have set a time limit of 60000 seconds of CPU time. Both the actual CPU time and the corresponding wall clock time are mentioned in the result tables.


(a) Wafer map with $\beta = 7$


(b) Wafer map with $\beta = 14$

The memory limit has been set to 20 Gigabytes and SCIP version 5.0.1 has been used to run the branch-and-price algorithm, while the branch-and-cut (B&C) algorithm of ILOG CPLEX (version 12.7.1) has been used to provide a point of comparison.

The discussion of our results is divided into two parts. Among all instances, a set of 640 instances are small enough to be solved to optimality by giving the full AIP formulation (1)-(3) to CPLEX B&C. We refer to them as *easy instances.* They include all the instances with $m = 3$, as well as all instances with $m = 5$ and $n = 15$. The remaining instances define the set of *hard instances.*

7.1.1 Easy instances

Table 1 summarizes the performance of CPLEX branch-and-cut algorithm (B&C) and of the branch-and-price algorithm (B&P) on each subclass of easy instances. It contains the average CPU and wall clock running times (both in seconds), the average number of nodes in the branching tree, and the average integrality *int_gap* between the optimal value (OPT) and the rounded-up optimal value of the linear relaxation of AIP ($\lceil LP \rceil$):

$$int\_gap = \frac{\text{OPT} - \lceil LP \rceil}{\lceil LP \rceil} \times 100\%. \tag{16}$$

All averages are computed over 10 instances in each subclass.

Let us first remark that the AIP formulation is extremely tight, as witnessed by the small value of the integrality gap. (In fact, the lower bound $\lceil LP \rceil$ is equal to the optimal

		CPU (s)		Wall Clock (s)		# nodes		*int_gap*
	p	B&C	B&P	B&C	B&P	B&C	B&P	
NB	100	0.215	0.066	0.079	0.096	0.0	3.1	0.0000%
	200	0.251	0.071	0.086	0.092	0.0	2.8	0.0116%
	400	0.253	0.085	0.095	0.090	0.0	4.1	0.0109%
	800	0.196	0.070	0.073	0.084	0.0	2.9	0.0000%
US	100	0.228	0.079	0.075	0.094	0.0	3.3	0.0000%
	200	0.215	0.069	0.072	0.088	0.0	3.0	0.0140%
	400	0.213	0.080	0.072	0.087	0.0	4.0	0.0000%
	800	0.235	0.079	0.076	0.087	0.0	3.3	0.0064%
UVS	100	0.223	0.077	0.081	0.102	0.0	3.0	0.0000%
	200	0.233	0.077	0.081	0.086	0.0	3.3	0.0000%
	400	0.199	0.075	0.066	0.084	0.0	3.0	0.0000%
	800	0.233	0.090	0.079	0.096	0.0	3.3	0.0000%
UUS	100	0.239	0.077	0.084	0.107	0.0	4.4	0.0000%
	200	0.242	0.085	0.087	0.103	0.0	4.0	0.0000%
	400	0.227	0.085	0.073	0.098	0.0	4.0	0.0000%
	800	0.222	0.076	0.076	0.093	0.0	3.0	0.0000%

(a) $m = 3$, $n = 15$

		CPU (s)		Wall Clock (s)		# nodes		*int_gap*
	p	B&C	B&P	B&C	B&P	B&C	B&P	
NB	100	0.993	0.301	0.318	0.262	0.0	6.2	0.0595%
	200	1.157	0.332	0.366	0.251	0.0	8.5	0.0208%
	400	1.463	0.479	0.422	0.352	0.0	12.7	0.0302%
	800	1.135	0.579	0.336	0.390	0.0	17.2	0.0163%
US	100	1.001	0.282	0.310	0.255	0.0	5.7	0.0157%
	200	0.933	0.274	0.292	0.240	0.0	4.8	0.0083%
	400	0.963	0.361	0.336	0.267	0.0	9.8	0.0039%
	800	0.956	0.469	0.322	0.321	0.0	12.4	0.0137%
UVS	100	0.937	0.215	0.286	0.200	0.0	3.3	0.0000%
	200	1.064	0.235	0.326	0.217	0.0	3.8	0.0000%
	400	1.045	0.313	0.350	0.258	0.0	6.0	0.0076%
	800	1.012	0.35	0.348	0.292	0.0	5.5	0.0073%
UUS	100	1.051	0.165	0.332	0.164	0.0	4.4	0.0000%
	200	0.942	0.209	0.284	0.203	0.0	3.7	0.0000%
	400	1.116	0.201	0.411	0.192	0.0	3.0	0.0000%
	800	1.026	0.267	0.335	0.230	0.0	3.9	0.0000%

(b) $m = 3$, $n = 25$

		CPU (s)		Wall Clock (s)		# nodes		*int_gap*
	p	B&C	B&P	B&C	B&P	B&C	B&P	
NB	100	3.768	0.792	1.057	0.546	0.0	9.0	0.0210%
	200	3.773	1.181	1.078	0.708	0.0	14.8	0.0297%
	400	3.401	1.028	0.919	0.617	0.0	12.4	0.0096%
	800	3.193	1.661	0.902	0.856	0.0	21.9	0.0175%
US	100	3.660	0.700	1.013	0.480	0.0	8.5	0.0379%
	200	3.728	1.091	1.018	0.649	0.0	15.1	0.0237%
	400	3.191	0.892	0.866	0.555	0.0	10.6	0.0085%
	800	3.219	1.567	0.901	0.808	0.0	19.2	0.0153%
UVS	100	3.456	0.527	0.945	0.427	0.0	3.8	0.0000%
	200	3.310	0.677	0.849	0.491	0.0	5.3	0.0235%
	400	3.612	0.938	0.924	0.609	0.0	9.8	0.0109%
	800	3.176	0.967	0.916	0.623	0.0	8.0	0.0053%
UUS	100	3.120	0.413	0.874	0.343	0.0	4.7	0.0000%
	200	2.997	0.421	0.809	0.338	0.0	4.6	0.0000%
	400	3.819	0.479	1.312	0.385	0.0	3.0	0.0000%
	800	3.176	0.698	0.777	0.491	0.0	4.7	0.0000%

(c) $m = 3$, $n = 35$

		CPU (s)		Wall Clock (s)		# nodes		*int_gap*
	p	B&C	B&P	B&C	B&P	B&C	B&P	
NB	100	122.869	22.890	33.755	4.052	35.3	104.9	0.2407%
	200	108.049	95.598	29.837	12.550	58.3	217.1	0.1906%
	400	151.122	347.423	39.398	48.923	436.1	906.6	0.1788%
	800	141.250	254.766	37.645	34.535	446.9	560.4	0.1141%
US	100	104.360	32.702	28.608	5.678	21.5	157.4	0.2537%
	200	113.738	105.287	31.423	12.932	82.0	250.9	0.2270%
	400	111.369	91.006	31.265	11.980	42.2	201.0	0.1428%
	800	115.120	78.440	31.394	10.508	58.7	143.1	0.0940%
UVS	100	97.881	15.728	27.115	2.890	0.0	65.8	0.1857%
	200	100.444	43.951	27.36	5.698	34.5	110.7	0.1741%
	400	108.470	41.940	29.108	5.308	26.0	88.0	0.1150%
	800	116.235	67.598	31.306	8.998	21.4	124.5	0.0755%
UUS	100	105.482	1.979	30.262	0.671	0.0	5.5	0.0000%
	200	93.671	3.258	25.606	0.658	0.0	7.5	0.0000%
	400	88.100	2.537	21.435	0.486	0.0	3.7	0.0000%
	800	85.971	5.903	19.84	0.907	0.0	7.1	0.0369%

(d) $m = 5$, $n = 15$

Table 1: Branch-and-cut vs. branch-and-price performance (average CPU time, wall clock time, number of nodes in the branching tree, integrality gap) on easy instances

value for 451 out of these 640 easy instances.)

As a consequence, for all instances with $m = 3$ and for all instances in UUS$(5, 15, *)$, B&C never enters the branching process: it finds the optimal solution after preprocessing and root node solving. However, despite the smaller size of its branching tree, branch-and-cut is slower than branch-and-price on almost all easy instances, with the exception of the instances in NB$(5, 15, 400)$ and NB$(5, 15, 800)$ for CPU time, and of the instances in NB$(5, 15, 400)$ for clock wall time.

Next, we can observe that the integrality gap increases with m and n, as does the running time of both branch-and-cut and branch-and-price. This behavior is clearly expected, as the number of variables in AIP increases with these two parameters. Moreover, for branch-and-price, larger values of m and n also lead to a more difficult pricing problem, since the number of m-tuples to enumerate is equal to n^m.

In fact, the running time of branch-and-cut on easy instances appears to be mostly determined by m and n (it is roughly constant for fixed values of these parameters), whereas the performance of branch-and-price appears to be affected by other factors as well, in particular, by the value of p (i.e., the size of the wafers) and by the number of nodes in the branching tree.

The impact of p on the running time of branch-and-price is partially due to a (slight) increase of the integrality gap, but also to the fact that, in order to avoid memory issues, the cost of each m-tuple is dynamically computed in the pricing problem (see Section 3.3), rather than stored in memory. Code profiling shows indeed that an average 25% of computation time is spent on counting the number of components equal to one in the m-tuples. As an illustration of the influence of parameter p, observe for example that the branch-and-price algorithm takes more time to solve the instances in US$(5, 15, 800)$ than those in US$(5, 15, 100)$, even though the branching tree has fewer nodes in the former case. The influence of p is way less noticeable when solving the full model by branch-and-cut, since in this case, the cost of each m-tuple is precomputed and hard-coded in the objective function coefficients of the model. (The dependence of the branch-and-price algorithm on the vector size p could perhaps be alleviated by storing all m-tuples costs in memory, but this creates memory problems for larger instances.)

In general, however, the main factor affecting the performance of branch-and-price appears to be the size of the branching tree. For the instances in NB$(5, 15, *)$, for example, the correlation between the (CPU or wall clock) running time and the number of nodes is almost perfect (equal to 0.99). Once again, this trend is less noticeable with branch-and-cut: even when it must resort to branching, the bulk of the running time of B&C goes into the preprocessing work performed at the root node.

The quality of the lower bound, the number of nodes (and, by way of consequence, the running time) of branch-and-price is directly impacted by the sparsity of the instances. This is quite apparent when comparing US, UVS and UUS instances with the same parameters m and n.

	$m = 5$, $n = 25$				$m = 5$, $n = 35$			
p	NB	US	UVS	UUS	NB	US	UVS	UUS
100	9	9	10	10	0	0	2	10
200	6	4	10	10	0	0	0	10
400	1	4	9	10	0	0	0	9
800	1	6	9	10	0	0	0	5
Total	17	23	38	40	-	-	2	34

Table 2: Branch-and-price algorithm: Number of instances solved to optimality

7.1.2 Hard instances

In this section we consider instances for which CPLEX B&C does not return any solution due to the heavy memory consumption of the full AIP formulation. Thus we only discuss the results returned by the branch-and-price algorithm. As a first indication, Table 2 provides a synthetic overview of the number of instances solved to optimality within 60000 seconds of CPU time for $m = 5$ and $n \in \{25, 35\}$. (Remember that there are 10 instances in each class.)

Table 3 gives more details about the performance of the branch-and-price algorithm on the instances solved to optimality within the time limit. The algorithm solves 118 among the 160 instances with $m = 5$ and $n = 25$, and 36 among the 160 instances with $m = 5$ and $n = 35$. These are mostly sparse instances with a small number of components per vector. These results confirm the sensitivity of the branch-and-price algorithm to the values of n, p, and to the sparsity of the instances, as already observed for easy instances in the previous section. One can actually notice in a more pronounced fashion the influence of sparsity on the quality of the lower bound (the integrality gap given by Equation (16) increases with p), on the number of nodes in the branching tree and on the running time of the algorithm: the wall clock solving time ranges from a few seconds for ultra-sparse (UUS) instances to several thousand seconds for the densest instances. (The integrality gap is zero for 55 of the 154 hard instances solved to optimality, all of them of the UUS type.)

Let us now turn to the instances that could not be solved to optimality within the time limit. Table 4 shows the performance of the branch-and-price algorithm when restricted to these instances. Beside the number of instances in each subclass and the average number of nodes in the branching tree, the table displays two measures of the (average) optimality gap for each subclass of instances, namely, *diff* and *gap*: if *ub* denotes the best value of a feasible solution obtained by branch-and-price, and if *lb* denotes the best lower bound computed on the optimal value of an instance, then we define the *absolute difference* $diff = ub - \lceil lb \rceil$, and the *relative gap*

$$gap = \frac{ub - \lceil lb \rceil}{\lceil lb \rceil} \times 100\%. \tag{17}$$

The majority of the instances in Table 4 are those with $m = 5$ and $n = 35$. Indeed, because exhaustive enumeration with complexity $O(n^m)$ is used for pricing, even a small

	p	# inst	CPU (s)	Wall Clock (s)	# nodes	*int_gap*
NB	100	9	21823.310	3240.590	14784.3	0.4110%
	200	6	25746.407	2318.499	7797.7	0.2555%
	400	1	33650.790	2952.017	9464.0	0.1793%
	800	1	45496.160	4059.979	12222.0	0.1358%
US	100	9	9774.341	1271.886	6421.2	0.4042%
	200	4	24816.468	2269.256	7815.2	0.2603%
	400	4	18374.928	1560.585	5158.5	0.1715%
	800	6	32250.569	2922.909	8786.8	0.1188%
UVS	100	10	2505.699	261.169	1500.3	0.3710%
	200	10	4274.250	366.582	1088.1	0.2060%
	400	9	13588.724	1169.891	3723.1	0.1688%
	800	9	27070.183	2400.674	7287.8	0.1318%
UUS	100	10	26.658	5.106	15.4	0.0000%
	200	10	49.316	5.563	12.8	0.0345%
	400	10	102.810	9.768	21.8	0.0727%
	800	10	407.384	37.367	70.4	0.0813%

(a) $m = 5, n = 25$

	p	# inst	CPU (s)	Wall Clock (s)	# nodes	*int_gap*
UVS	100	2	1880.822	2309.616	2712.5	0.4451%
UUS	100	10	633.143	79.192	77.8	0.0000%
	200	10	411.365	36.397	22.7	0.0000%
	400	9	11507.609	889.296	486.556	0.0994%
	800	5	17249.189	1325.007	672.0	0.1020%

(b) $m = 5,\ n = 35$

Table 3: Branch-and-price algorithm: average performance on instances solved to optimality (computed over the number of instances reported in column '# inst')

	p	# inst	*gap*	*diff*	# nodes
NB	100	1	0.232 %	2.000	36960.0
	200	4	0.173 %	3.250	18869.7
	400	9	0.119 %	4.667	17115.1
	800	9	0.090 %	7.334	16108.4
US	100	1	0.249 %	2.000	38572.0
	200	6	0.175 %	3.000	17997.3
	400	6	0.129 %	4.667	17502.3
	800	4	0.070 %	5.250	16231.7
UVS	400	1	0.098 %	2.000	19516.0
	800	1	0.048 %	2.000	16534.0

(a) $m = 5,\ n = 25$

	p	# inst	*gap*	*diff*	# nodes
NB	100	10	1.844 %	21.800	6699.100
	200	10	1.818 %	47.100	2900.800
	400	10	1.391 %	75.400	2482.400
	800	10	1.125 %	127.000	2465.500
US	100	10	1.816 %	19.700	6798.700
	200	10	1.012 %	24.000	2812.100
	400	10	1.046 %	52.700	2395.900
	800	10	0.588 %	61.700	2271.400
UVS	100	8	0.742 %	4.250	6912.500
	200	10	0.401 %	5.100	2786.900
	400	10	0.370 %	10.100	2318.000
	800	10	0.570 %	32.700	2343.900
UUS	400	1	0.174 %	1.000	3135.000
	800	5	0.147 %	1.800	2518.000

(b) $m = 5,\ n = 35$

Table 4: Branch-and-price algorithm: average performance on instances reaching the time limit (computed over the number of instances reported in column '# inst')

increase of either m or n badly worsens the overall performance. This is highlighted by the low number of nodes explored in the branching process when $n = 35$ (between 2700 and 7000 nodes, depending mostly on p, the number of dies per wafer) as compared with the case when $n = 25$ (between 16000 and 40000 nodes, depending again on p).

Even when it runs out of time, the branch-and-price algorithm is able to produce solutions and lower bounds of high quality for the instances in Table 4: the average gap between the best integral solution and the best lower bound is very small and never exceeds 1.85%. It is even smaller for instances with $n = 25$, where it never exceeds 0.25%: the enumeration being more efficient on instances with $n = 25$, a higher number of nodes is explored before the time limit, and this potentially allows the algorithm to improve both the lower bound and the best known solution. (This phenomenon is even more evident on the absolute difference *diff*, which increases significantly when the number of wafers goes from $n = 25$ to $n = 35$.)

Interestingly, for given values of m and n and for a fixed sparsity class, the gap tends to decrease when p increases, while *diff* simultaneously increases. This is due to the fact that the number of defective dies grows with p, which leads to larger values of the objective function and hence, to larger absolute differences between upper and lower bounds.

In conclusion, the branch-and-price algorithm is able to handle larger instances without facing the memory issues that arise when trying to solve the complete AIP formulation by branch-and-cut. It allows us to solve some of these instances to optimality (especially when $m = 5$ and $n = 25$) and yields good quality solutions for others. This confirms, in particular, that the AIP formulation is quite tight, as we already observed when we examined the case of easier instances in Section 7.1.1.

However, the computation time of branch-and-price remains high, which could make this algorithm unsuitable for solving larger instances and for industrial purposes. Therefore, in the next section, we take a look at the performance of the price-and-branch heuristic introduced in Section 4.

7.2 Price-and-branch algorithm

In this section we examine the performance of the price-and-branch heuristic P&B. The experiments have been conducted on an Intel Core i7-7700HQ CPU 2.80GHz, 4 Cores with 16 Gigabytes of RAM. As in earlier sections, ILOG CPLEX version 12.7.1 has been used to solve the linear and integer programming problems (with the default settings and a maximum of two solver threads).

We first focus on the quality of the solution returned by P&B as compared with the optimal solution when the latter is known. Next, we consider the improvement brought by P&B over the simple heuristic SHH.

7.2.1 Easy instances

The subset of easy instances is the same as described in Section 7.1.1. Recall that both branch-and-cut (CPLEX) and branch-and-price were able to solve these instances to

	NB($m,n,*$)				Uniform($m,n,*$)			
	3	3	3	5	3	3	3	5
	15	25	35	15	15	25	35	15
ub = OPT	40	37	34	17	119	118	118	95
ub = OPT+1	0	3	6	15	1	2	2	20
ub = OPT+2	0	0	0	5	0	0	0	5
ub = OPT+3	0	0	0	2	0	0	0	0
ub = OPT+4	0	0	0	1	0	0	0	0

Table 5: Price-and-branch algorithm: Number of easy instances with ub = OPT + Δ, for $\Delta = 0, \ldots, 4$

optimality, so that we know their optimal value, say OPT. The running time of price-and-branch for these instances is approximately 0.8 second for the instances with $m = 3$, and 1 second for $m = 5$, $n = 15$, after which P&B returns a feasible assignment with value ub. As shown in Table 5, the optimal value is obtained by P&B for 90% of these instances (ub = OPT for 578 out of 640 instances). For the remaining 62 instances, the solution delivered by P&B is almost optimal: ub lies most of the time within 1 unit of the optimal value, and never exceeds OPT+4. The difference between ub and the optimal value tends to increase with the problem size, but remains very small in all cases.

7.2.2 Hard instances

We now turn to hard instances which could not be solved by the CPLEX branch-and-cut solver due to insufficient memory (as in Section 7.1.2), or which could not even be solved by branch-and-price: these are all 120 instances with $(m, n) = (5, 25)$, (5,35), and (7,15). Table 6 presents the results obtained by the price-and-branch algorithm on this set of instances. In all cases, a time limit of 1200 seconds is set for the branching phase where the IP version of MP* (i.e., IMP*) is solved by CPLEX (see Section 4.1; we do not set any time limit for the pricing phase). In many cases, IMP* cannot be solved to optimality within the time limit, and the best integer solution obtained upon termination is then reported. We denote again by ub the value of this heuristic solution, and we define gap as

$$gap = \frac{ub - \lceil LP \rceil}{\lceil LP \rceil} \times 100\%. \tag{18}$$

Table 6 displays the average value of the gap as well as the average running time of P&B. As we can see, for all the instances considered here, the gap remains extremely small (at most 1.5%), and it tends to decrease when either p or the sparsity increases. P&B actually finds an optimal solution for 40 UUS instances (gap = 0%). This is consistent with previous observations, and demonstrates the excellent quality of the heuristic solution and of the lower bound delivered by P&B. The running time of the algorithm is relatively short, except in some cases (e.g., UUS(7,15,100)) where the pricing phase lasts very long, which explains that the reported time exceeds the limit of 1200 seconds.

	p	*gap*	CPU (s)	SHH *gap*	SHH impr.
NB	100	0.58%	72.49	3.80%	84.22%
	200	0.39%	106.17	2.68%	84.97%
	400	0.29%	287.73	1.90%	84.47%
	800	0.21%	213.99	1.34%	83.67%
US	100	0.59%	41.69	3.93%	85.20%
	200	0.38%	56.09	2.60%	85.14%
	400	0.28%	64.44	1.71%	82.99%
	800	0.16%	39.11	1.13%	85.91%
UVS	100	0.40%	18.03	4.53%	91.06%
	200	0.40%	28.33	3.06%	86.30%
	400	0.24%	25.71	1.85%	86.85%
	800	0.19%	35.54	1.38%	86.22%
UUS	100	0.00%	212.86	3.28%	100.00%
	200	0.00%	75.06	4.26%	100.00%
	400	0.07%	28.79	3.42%	98.04%
	800	0.09%	18.35	1.91%	94.89%

(a) $m = 5, n = 25$

	p	*gap*	CPU (s)	SHH *gap*	SHH impr.
NB	100	1.05%	610.92	4.07%	73.91%
	200	0.76%	641.35	2.87%	73.32%
	400	0.56%	644.48	2.06%	72.70%
	800	0.46%	645.45	1.47%	67.78%
US	100	647.14	0.88%	4.19%	77.80%
	200	645.65	0.67%	2.94%	76.15%
	400	645.84	0.49%	1.96%	73.32%
	800	643.45	0.37%	1.31%	71.71%
UVS	100	0.62%	248.96	4.87%	86.66%
	200	0.46%	408.50	3.21%	84.98%
	400	0.33%	536.80	2.06%	83.97%
	800	0.27%	491.76	1.42%	80.64%
UUS	100	0.00%	1380.74	4.02%	100.00%
	200	0.04%	257.78	5.24%	99.33%
	400	0.14%	81.82	3.69%	96.12%
	800	0.17%	100.08	2.23%	92.35%

(b) $m = 5, n = 35$

Table 6: Price-and-branch and sequential heavy heuristic results

	p	*gap*	CPU (s)	SHH *gap*	SHH impr.
NB	100	1.49%	900.21	4.98%	67.43%
	200	1.01%	1127.14	3.58%	70.33%
	400	1.00%	1261.02	2.47%	58.05%
	800	1.20%	1273.44	1.81%	43.04%
US	100	1.25%	749.66	4.99%	74.72%
	200	0.99%	911.35	3.69%	72.19%
	400	0.86%	1070.97	2.41%	63.11%
	800	0.67%	989.41	1.80%	62.23%
UVS	100	1.05%	149.42	6.65%	84.04%
	200	0.99%	911.35	3.69%	72.19%
	400	0.56%	361.10	2.94%	80.40%
	800	0.40%	520.42	1.99%	79.53%
UUS	100	0.00%	3333.87	3.25%	100.00%
	200	0.07%	768.51	5.26%	98.75%
	400	0.03%	256.03	3.96%	99.38%
	800	0.16%	132.20	2.48%	92.89%

(c) $m = 7, n = 15$

Table 6: Price-and-branch and sequential heavy heuristic results

Since P&B is a heuristic, we find it interesting to compare its quality with the fast and simple SHH procedure (see Section 5). We observed that P&B always improves upon SHH, except for two instances out of 120. The columns labeled "SHH gap" and "SHH impr." in Table 6 display the (average) gap associated with SHH solutions, and the (average) percentage of the gap between the SHH upper bound and the lower bound that is closed by P&B, respectively. We can see that P&B almost completely closes the SHH gap in ultra-sparse instances, and closes 70%–90% of the gap, on average, for most of the other instances. Of course, this improvement comes with a clear trade-off in running time, but nevertheless demonstrates the potential benefit of using P&B as an alternative heuristic procedure.

8 Conclusion

In this paper, we have presented several exact and heuristic approaches for the solution of the wafer-to-wafer integration problem. Little work has been previously published concerning the exact solution of this difficult problem, and efficient algorithms could potentially improve the quality of the proposed wafer stacks.

Our results confirm that the AIP formulation is tight. But its size grows quickly with m (the number of batches) and n (the number of wafers per batch), and this leads to memory overflows when we feed this formulation to an ILP solver, even for relatively small instances. Column generation techniques are able to partially alleviate the memory issues and yield more efficient algorithms. While exact resolution by branch-and-price can be used to tackle instances of moderate size, it fails to scale very well for larger sizes. Our price-and-branch heuristic algorithm, on the other hand, turns out to produce very high quality solutions without requiring too much computation time, even for large instances involving 5 batches and 35 wafers per batch, or 7 batches and 15 wafers per batch.

Both branch-and-price and price-and-branch require the solution of a hard pricing problem, which amounts to selecting an optimal m-tuple (or stack) of wafers according to some appropriately defined objective function. In our implementation, this pricing problem is solved by an exhaustive enumeration procedure. While the efficiency of this procedure can be considerably improved by taking advantage of parallelization, it still remains the main bottleneck of our approaches. Streamlining the enumeration, or developing more efficient optimization algorithms for the pricing problem would open new perspectives for the solution of the wafer-to-wafer integration problem.

References

[1] H. Bandelt, Y. Crama, and F. Spieksma. Approximation algorithms for multi-dimensional assignment problems with decomposable costs. *Discrete Applied Mathematics*, 49:25–50, 1994.

[2] C. Barnhart, E. L. Johnson, G. L. Nemhauser, M. W. P. Savelsbergh, and P. H. Vance. Branch-and-price: Column generation for solving huge integer programs. *Operations Research*, 46:316–329, 1998.

[3] M. Bougeret, V. Boudet, T. Dokka, G. Duvillié, and R. Giroudeau. On the complexity of wafer-to-wafer integration. *Discrete Optimization*, 22:255 – 269, 2016.

[4] R. Burkard, M. Dell'Amico, and S. Martello. *Assignment Problems.* SIAM, Philadelphia, 2009.

[5] Y. Crama and F. Spieksma. Approximation algorithms for three-dimensional assignment problems with triangle inequalities. *European Journal of Operational Research*, 60:273–279, 1992.

[6] T. Dokka. *Algorithms for multi-index assignment problems.* PhD thesis, Katholieke Universiteit Leuven, Belgium, 2013.

[7] T. Dokka. Multi-dimensional vector assignment problems: integer programming approaches. *61st UK OR Society conference, Lancaster*, 2018.

[8] T. Dokka, M. Bougeret, V. Boudet, R. Giroudeau, and F. Spieksma. Approximation algorithms for the wafer to wafer integration problem. In T. Erlebach and G. Persiano, editors, *Proceedings of the 10th Workshop on Approximation and Online Algorithms (WAOA2012)*, pages 286–297. Springer, 2013.

[9] T. Dokka, Y. Crama, and F. Spieksma. Multi-dimensional vector assignment problems. *Discrete Optimization*, 14:111–125, 2014.

[10] G. Duvillié. *Approximability, parameterized complexity and solving strategies for some multidimensional assignment problems.* PhD thesis, Université de Montpellier, France, 2016.

[11] P. Garrou, C. Bower, and P. Ramm (editors). *Handbook of 3D Integration: Volume 1 - Technology and Applications of 3D Integrated Circuits.* Wiley, 2008.

[12] Ambros Gleixner, Leon Eifler, Tristan Gally, Gerald Gamrath, Patrick Gemander, Robert Lion Gottwald, Gregor Hendel, Christopher Hojny, Thorsten Koch, Matthias Miltenberger, Benjamin Müller, Marc E. Pfetsch, Christian Puchert, Daniel Rehfeldt, Franziska Schlösser, Felipe Serrano, Yuji Shinano, Jan Merlin Viernickel, Stefan Vigerske, Dieter Weninger, Jonas T. Witt, and Jakob Witzig. The SCIP Optimization Suite 5.0. Technical report, Optimization Online, December 2017.

[13] Ambros Gleixner, Leon Eifler, Tristan Gally, Gerald Gamrath, Patrick Gemander, Robert Lion Gottwald, Gregor Hendel, Christopher Hojny, Thorsten Koch, Matthias Miltenberger, Benjamin Müller, Marc E. Pfetsch, Christian Puchert, Daniel Rehfeldt, Franziska Schlösser, Felipe Serrano, Yuji Shinano, Jan Merlin Viernickel, Stefan Vigerske, Dieter Weninger, Jonas T. Witt, and Jakob Witzig. The SCIP Optimization Suite 5.0. ZIB-Report 17-61, Zuse Institute Berlin, December 2017.

[14] M. Harzi, H. Abusenenh, and S. Krichen. Solving the yield optimization problem for wafer to wafer 3D integration process. *Procedia - Social and Behavioral Sciences*, 195:1905–1914, 2015.

[15] S. Reda, L. Smith, and G. Smith. Maximizing the functional yield of wafer-to-wafer integration. *IEEE Transactions on VLSI Systems*, 17:1357–1362, 2009.

[16] D.M. Ryan and B.A. Foster. An integer programming approach to scheduling. In A. Wren, editor, *Computer Scheduling of Public Transport*, pages 269–280. North-Holland Publishing Company, 1981.

[17] E. Singh. Wafer ordering heuristic for iterative wafer matching in W2W 3D-SICs with diverse die yields. In *3D-TEST, First IEEE International Workshop on Testing Three-Dimensional Stacked Integrated Circuits.* IEEE, 2010. Extended summary.

[18] E. Singh. Exploiting rotational symmetries for improved stacked yields in W2W 3D-SICs. In *Proceedings of the 29th IEEE VLSI Test Symposium (VTS'11)*, pages 32–37. IEEE, 2011.

[19] F. Spieksma. Multi-index assignment problems: complexity, approximation, applications. In P.M. Pardalos and L. Pitsoulis, editors, *Nonlinear Assignment Problems: Algorithms and Applications*, pages 1–12. Kluwer Academic Publisher, 2000.

[20] C. H. Stapper. Small-area fault clusters and fault tolerance in vlsi circuits. *IBM Journal of Research and Development*, 33:174–177, 1989.

[21] M. Taouil. *Yield and cost analysis for 3D stacked ICs.* PhD thesis, TU Delft, The Netherlands, 2014.

[22] M. Taouil and S. Hamdioui. Layer redundancy based yield improvement for 3D wafer-to-wafer stacked memories. *IEEE European Test Symposium*, pages 45–50, 2011.

[23] M. Taouil, S. Hamdioui, and E. Marinissen. Yield improvement for 3D wafer-to-wafer stacked ICs using wafer matching. *ACM Transactions on Design Automation of Electronic Systems,*, page DOI: http://dx.doi.org/10.1145/2699832, 2015.

[24] M. Taouil, S. Hamdioui, J. Verbree, and E. Marinissen. On maximizing the compound yield for 3D wafer-to-wafer stacked ICs. In IEEE, editor, *IEEE International Test Conference*, pages 183–192, 2010.

[25] J. Verbree, E. Marinissen, P. Roussel, and D. Velenis. On the cost-effectiveness of matching repositories of pre-tested wafers for wafer-to-wafer 3D chip stacking. *IEEE European Test Symposium*, pages 36–41, 2010.