Technical University of Munich
School of Computation, Information and Technology
Chair of Electronic Design Automation

Implementation and Analysis of the One-Pass Architectural Synthesis for Continuous-Flow Microfluidic Lab-on-a-Chip Systems

Research Internship Report

Hui Deng

Technical University of Munich
School of Computation, Information and Technology
Chair of Electronic Design Automation

Implementation and Analysis of the One-Pass Architectural Synthesis for Continuous-Flow Microfluidic Lab-on-a-Chip Systems

Research Internship Report

Hui Deng

Advisor :	Meng Lian
Advising Professor :	Prof. Dr.-Ing. Ulf Schlichtmann
Topic issued :	22.04.2024
Working period :	22.04.2024 - 20.09.2024

Hui Deng
Arcisstraße 21
80333 München

Abstract

In recent years, the emergence of continuous-flow microfluidic technology has revolutionized fields such as biochemistry and biomedicine . On these microscale lab-on-chip systems, complex biochemical analyses—such as DNA analysis and drug discovery—can be performed automatically and efficiently without the need for manual intervention. Over the past few years, the automated design of such chips has become a significant research focus due to the high complexity of analytical protocols and chip architectures, with numerous studies dedicated to the design automation of these chips. The current mainstream approach divides the chip design process into four independent tasks: binding, scheduling, placement, and routing. However, the lack of communication between these separate stages often leads to failures in the automated design process.

To address these issues, a novel automated design flow called BigIntegr for continuous-flow microfluidic lab-on-chip systems was proposed in [1]. The paper claims that this process integrates all design steps into a unified "organic whole," allowing tasks such as binding, scheduling, placement, and routing to be seamlessly synchronized and executed in a combinatorial manner, thereby eliminating gaps between the design stages. Consequently, efficient and cost-effective biochip architectures can be generated without the need for design adjustments or modifications. However, the mathematical derivations provided in [1] are somewhat vague, and the lack of practical implementation code raises concerns about the validity of the data and the feasibility of the model.

To investigate and assess the feasibility of this approach, this thesis analyzes and improves the model based on the mathematical derivations provided in [1] and successfully re-implements the model. According to the test results, the effectiveness and practical value of the modified automated flow have been demonstrated.

`Keywords: design automation, code implementation, Continuous-flow microfluidic biochips (CFMBs)`

Contents

List of Figures

List of Tables

1. Introduction

1.1. Motivation

The automation of Continuous-flow microfluidic biochips (CFMBs) architecture design is a complex multi-objective optimization problem. To better optimize key metrics such as assay completion time, chip area, channel length, and channel intersections, the design flow of CFMBs is typically divided into several stages, including binding, scheduling, placement, and routing and the first two stages are part of the architecture-level synthesis, while the latter stages are referred to as physical-level synthesis [1]. To efficiently automate the generation of cost-effective biochip architectures, it is essential to systematically consider all design tasks, as the outcomes of these steps are closely interrelated. Designing each chip step in isolation introduces significant information shifts in subsequent steps as shown in figure 1.1, which may lead to a decline in solution quality or even design failure. To address these challenges, [1] claims to propose a novel approach to CFMB design automation. This method synthesizes all four steps simultaneously, executing them in a unified manner and enhancing the exchange of information between tasks [1]. By doing so, it eliminates the gaps between different design phases, ensuring the correctness and efficiency of the final solution while avoiding unnecessary delays in design convergence [1].

[1] presents this new CFMB design automation flow as an Integer Linear Programming (ILP) problem and lists the corresponding mathematical expressions. However, upon further examination, many of these mathematical formulations are logically unclear and ambiguous, and several corner cases are not addressed. Additionally, no implementation code for the proposed flow is provided.

Inspired by these observations, this thesis aims to verify the validity of the model proposed in [1] through code replication. The effectiveness and practicality of the model were evaluated across multiple design tasks. Furthermore, this thesis improves the model to better accommodate various corner cases, enhancing its robustness and applicability.

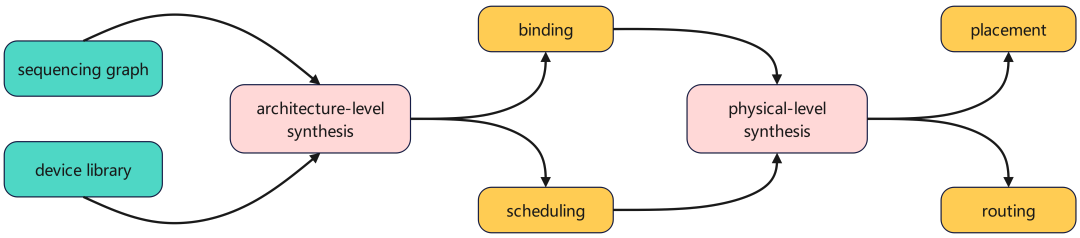


Figure 1.1.: chip synthesis flow [1].

1.2. Objective and contribution

To re-implement and improve the chip synthesis process proposed in [1], this thesis employs C++ as the tool for code implementation and Gurobi as the linear programming solver, designing a miniature design automation software based on BigIntegr. This software accepts text data as input, where the data must be formatted to include device information, the protocol of the assay to be realized, modeled as a directed acyclic graph $G(O, E)$ and design constraints like maximal chip area. Based on the input, the software automatically designs the chip architecture according to the specified requirements and outputs the results in text format.

Through extensive testing, this thesis improves and validates the effectiveness of the re-implemented synthesis design and proposes methods to address corner cases.

1.3. Organization

The organization of the rest part of the paper is as follows. Section 2 provides an overview of continuous-flow microfluidic biochips and the background of the automated design of such chip architectures, enabling readers to better understand the implementation of the synthesis process. Section 3 outlines the mathematical expression behind the model, along with the code details, and identifies shortcomings in the model proposed in [1] while offering improvements. Section 4 presents the development process and details of the automated design software implemented in this thesis. Section 5 presents the test results, and Section 6 summarizes the findings and suggests areas for future improvements and further exploration.

2. Background

In this section, the principles of continuous-flow microfluidic biochips and the foundational background of automated chip architecture design are introduced to provide readers with a clearer understanding of the subsequent implementation of the synthesis process.

2.1. Continuous-flow microfluidic biochips introduction

Continuous-flow microfluidic biochips (CFMBs), often referred to as lab-on-a-chip systems, have garnered significant attention from both academic and industrial researchers over the past ten years [2]. On these microscale platforms, complex biochemical analyses, such as point-of-care diagnostics [3], can be conducted automatically and concurrently. Unlike traditional laboratory equipment that relies on manual operation, CFMBs offer distinct benefits, such as reduced size, improved reliability and precision, as well as lower usage of expensive samples and reagents [4].

Fluid flow control in CFMBs is achieved through the precise manipulation of flexible membranes, referred to as microvalves [1]. These microvalves are constructed from polydimethylsiloxane (PDMS), a commonly utilized elastomer [5]. By using microvalves as building blocks, various microfluidic devices, including mixers and chambers, can be developed [7]. Through the programming of predetermined control sequences, a wide array of biochemical assays can be executed automatically on the biochip [1].

The schematic and control principle of the biochip are illustrated in Figure 2.1 [6]. Microvalves are positioned at the intersections of the control and flow channels. When the valve is closed, an external pressure source injects air into the control channel through the control port, pushing the membrane downward into the flow channel, thereby blocking fluid movement [1]. Conversely, once the pressure is released, fluid transport within the flow channel resumes [1].

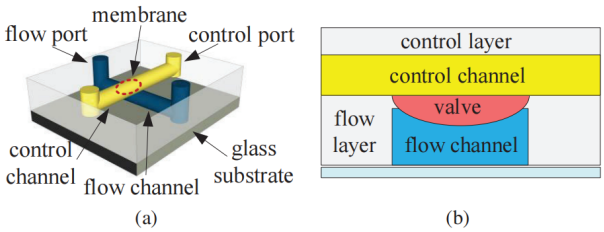


Figure 2.1.: Structure of a continuous-flow microfluidic biochip (a) and front view of the structure (b) [6].

2.2. One-Pass Architectural Synthesis for CFMBs introduction

2.2.1. limitations on current biochip design automation methods

With the advancements in microfabrication technology, the integration density of biochips can now reach nearly one million valves per square centimeter [8], significantly increasing the complexity of biochip design. To facilitate large-scale integration processes for biochips, recent years have seen a focus on researching and developing automated design methodologies for highly complex biochips [9]. The current mainstream research on chip design typically involves executing and optimizing one design task independently, such as resource binding, operation scheduling, device placement, and channel routing [1]. However, due to the independent nature of these steps—with no exchange of design results or constraint information—these methods often lead to a decline in the quality of chip architecture design or even impractical solutions [1]. For instance, as demonstrated in Figures 3 and 4 of [1], when design steps are executed separately—such as considering architecture-level synthesis and physical-level synthesis independently—an optimal binding and scheduling scheme for the bioassay may be derived during the architecture-level synthesis process, but the design may fail during the physical-level synthesis process [1]. This indicates that an optimal solution at one stage does not necessarily guarantee the overall optimal design, and the lack of information exchange between steps can easily result in suboptimal solutions or design failures.

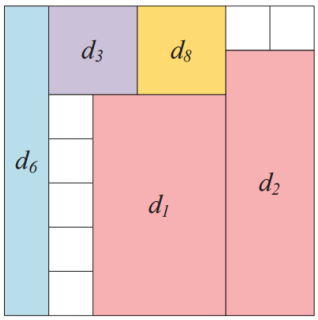


Figure 2.2.: Design failure occurring when performing the physical-level synthesis based on optimal architecture-level synthesis [1].

2.2.2. One-Pass Architectural Synthesis

To overcome the aforementioned issues, [1] proposes a one-pass synthesis flow called BigIntegr for CFMBs, which considers all design steps of CFMBs simultaneously [1]. This approach aims to eliminate the information gaps that arise when design steps are executed independently.

As illustrated in Figure 2.3, upon providing the appropriate input data, the proposed one-pass synthesis flow successfully generates the biochip architecture automatically.

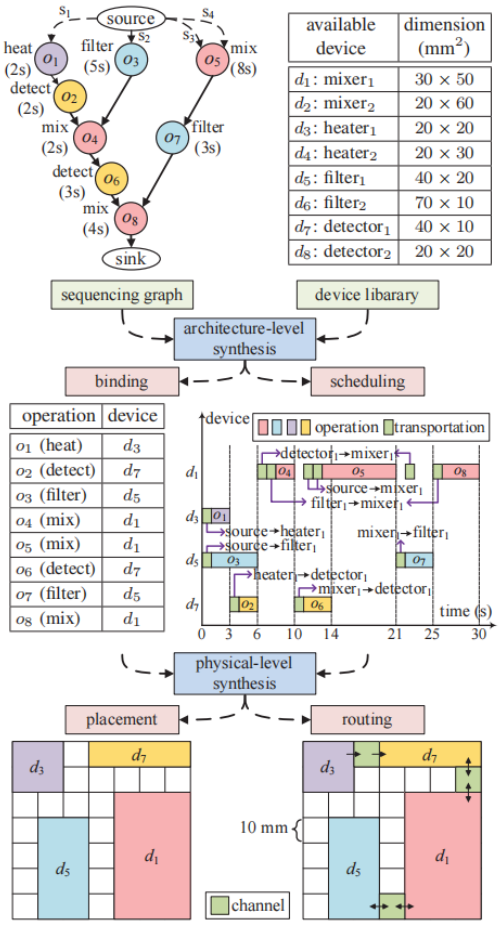


Figure 2.3.: Illustration of a complete one-pass synthesis flow for biochip architecture [1].

The automatic design of a chip architecture for a given bioassay using BigIntegr requires three types of input, which include:

1) Sequencing graph showing the protocol of the assay to be realized $G(O, E)$, where a vertex $o_i \in O$ represents a biochemical operation with a weight indicating its duration and an edge $e_{i,j} \in E$ specifies the dependency between operations;

2) A device library D used to facilitate the execution of operations in G, where a device $d_i \in D$ is represented by a rectangular box of size $w_i \times h_i$ featuring input/output ports along its boundary;

3) Design constraints like the maximal chip area, device resource and so on [1].

BigIntegr simultaneously analyzes and executes all design steps to achieve the chip architecture design automatically. Consequently, the objectives of the aforementioned design steps can be outlined as follows:

1) Binding: Determine a binding function $\Phi : O \rightarrow D$ such that each operation in the sequencing graph is assigned to a specific device for execution;

2) Scheduling: Establish the start and end times for each operation in G, ensuring that the completion time of the bioassay is minimized;

3) Placement: Position the assigned devices at precise locations on the chip plane, ensuring that the chip area adheres to the specified constraints;

4) Routing: Design optimal connections between devices to minimize chip costs, including channel length and intersections [1].

The proposed one-pass synthesis flow is modeled as an Integer Linear Programming (ILP) problem, wherein each of the design steps is expressed through a series of mathematical inequalities. The specific mathematical formulation of BigIntegr will be discussed in detail in Chapter 3.

3. Analysis of BigIntegr

As the mathematical model of BigIntegr presented in [1] are not sufficiently clear, this chapter focuses on discussing and analyzing the formulation and improvement of the BigIntegr's mathematical expressions. Additionally, it discusses the corner cases and limitations that were not adequately covered in [1].

3.1. Mathematical constraints analysis

The proposed one-pass synthesis flow is formulated as an Integer Linear Programming (ILP) problem, meaning the model is composed of a series of linear constraints. The improved mathematical model of BigIntegr presented in this study consists of 32 constraints, with each constraint comprising multiple specific and precise mathematical expressions that can be used as Gurobi input [10]. The optimization objective is also included within these constraints. Our constraints collectively address the following design tasks in chip design automation, including resource binding, operation scheduling, device placement, and channel routing, within a one-pass synthesis process. In addition, the symbols used in [1] to represent the design results at each stage of the complete synthesis flow are shown in Figure 3.1. Besides, the symbols used in the improved model are not included here and will be introduced in detail later.

Symbols frequently used in the synthesis flow
• $b_{i,k}$ (binding): A 0-1 variable representing whether operation o_i in the sequencing graph is bound to a device d_k in D. For the sake of simplicity, hereinafter we denote the allocated device of an operation o_i by $d(o_i)$;
• t_i^s and t_i^e (scheduling): The start time and end time of operation o_i in the generated scheduling, respectively;
• t^E (scheduling): The completion time of the given bioassay;
• $out(o_i)$ (scheduling): The resulting fluid of operation o_i;
• $\rho_{j,i}$ (scheduling): A 0-1 variable representing whether a storage is needed when transporting $out(o_j)$ from $d(o_j)$ to $d(o_i)$;
• $p_{j,i,k}$, k=1,2,3 (scheduling): 1) When a storage is needed, i.e., $\rho_{j,i} = 1$, $p_{j,i,1}$ and $p_{j,i,2}$ represent the transportation tasks of $out(o_j)$ from $d(o_j)$ to a storage and from the storage to $d(o_i)$, respectively. 2) When storage is not needed, i.e., $\rho_{j,i} = 0$, $p_{j,i,3}$ represents the transportation task of $out(o_j)$ from $d(o_j)$ to $d(o_i)$;
• $t^s_{p_{j,i,k}}$ and $t^e_{p_{j,i,k}}$ (scheduling): The start time and end time of transportation task $p_{j,i,k}$;
• (x_S^{lb}, y_S^{lb}) and (x_S^{ru}, y_S^{ru}) (placement): The lower-left and upper-right coordinates of a storage d_s, respectively;
• (x_i^{lb}, y_i^{lb}) and (x_i^{ru}, y_i^{ru}) (placement): The lower-left and upper-right coordinates of device $d(o_i)$, respectively;
• $g_{x,y}^{d(o_i)}$ (placement): A 0-1 variable representing whether a grid cell $g_{x,y}$ is occupied by device $d(o_i)$;
• $g_{x,y}^{p_{j,i,k}}$ (routing): A 0-1 variable representing whether a grid cell $g_{x,y}$ is occupied by the transportation task $p_{j,i,k}$.

Figure 3.1.: Symbols frequently used in the synthesis flow [1].

The first constraint addresses the binding of biochemical operations. In [1], it is stated that each biochemical operation $o_i \in O$ must be assigned to a specific device in D. However, the constraints presented in [1] only ensure that each operation is bound to one device in D, without imposing any restrictions on the type of device to which it is bound. To this end, the revised constraint is formulated as follows:

$$\sum_{d_k \in D} b_{i,k} = 1, \quad \sum_{d_k \in D_p} b_{i,k} = 1, \quad \forall o_i \in O, \tag{3.1}$$

where D represents the entire device library while D_p denotes the set of devices of the same type as the operation o_i.

Constraints (2) in [1] is part of the scheduling task. it ensures that the transportation tasks $P_{j,i,1}$ and $P_{j,i,2}$ must be performed sequentially, if device d_{o_i} is still occupied by another operation when transporting $out(o_j)$ and storage is required to temporarily cache $out(o_j)$ [1]. Since the mathematical expressions in [1] are accurate, they are omitted in this section.

Constraint (3) in [1] ensures that the corresponding **multi-input** tasks of operation o_i must be performed separately. Since the description in [1] is vague, constraint (3) is linearized using *big M method* [11] as

$$\begin{aligned} &\forall e_{j,i}, e_{h,i} \in E : \\ &\qquad t^e_{P_{j,i,2}} \le t^s_{P_{h,i,2}} + (1 - \lambda_{j,h,i}) \cdot M, \\ &\qquad t^e_{P_{h,i,2}} \le t^s_{P_{j,i,2}} + (1 - \lambda_{j,h,i}) \cdot M, \\ &\qquad t^e_{P_{j,i,3}} \le t^s_{P_{h,i,3}} + (1 - \lambda_{j,h,i}) \cdot M, \\ &\qquad t^e_{P_{h,i,3}} \le t^s_{P_{j,i,3}} + (1 - \lambda_{j,h,i}) \cdot M, \end{aligned} \tag{3.2}$$

where $\lambda_{j,h,i}$ is a 0-1 variable indicating the input order of $out(o_j)$ and $out(o_h)$ regarding $d(o_i)$.

Constraint (4), (5) and (6) are part of the scheduling task. Their functionalities are described as follows:
1) Constraint (4) ensures that the execution of o_i can only begin once all preceding input tasks have been completed;
2) Constraint (5) ensures that the execution time of each operation adheres to the specified duration in the sequencing graph, where t_{o_i} is the duration of operation o_i;
3) Constraint (6) ensures that the transportation of $out(o_i)$ can only begin once the execution of o_i has been completed [1]. Since the mathematical expressions in [1] are accurate, they are omitted in this section.

Constraint (7) in [1] ensures that the corresponding **multi-output** tasks of operation o_i must be performed separately. Since the description in [1] is vague, this study linearize it using *big*

M method [11] into four inequalities as

$$\begin{aligned}
&\forall e_{i,h}, e_{i,q} \in E : \\
&\qquad t^e_{P_{i,q,1}} \le t^s_{P_{i,h,1}} + (1-\mu_{i,h,q}) \cdot M, \\
&\qquad t^e_{P_{i,h,1}} \le t^s_{P_{i,q,1}} + (1-\mu_{i,h,q}) \cdot M, \\
&\qquad t^e_{P_{i,q,3}} \le t^s_{P_{i,h,3}} + (1-\mu_{i,h,q}) \cdot M, \\
&\qquad t^e_{P_{i,h,3}} \le t^s_{P_{i,q,3}} + (1-\mu_{i,h,q}) \cdot M,
\end{aligned} \tag{3.3}$$

where $\mu_{i,h,q}$ is a 0-1 variable indicating the output order of o_i.

Constraint (8) in [1] is part of the scheduling task. it ensures that all operations must be completed before the bioassay is finished[1]. Since the mathematical expressions in [1] are accurate, they are omitted in this section.

Constraint (9) in [1] can achieved by setting boundaries of Gurobi variables. Constraints (10) and (11) in [1] ensure that all allocated devices, including the storage, are placed within the specified chip area and do not overlap. Since the description in [1] is vague, we linearize constraint (10) using *big M method* [11] as follows:

$$\begin{aligned}
&\forall o_i, o_j \in O, i \ne j : \\
&\qquad \sum_{d \in D} S_{(i,j),k} \ge 1 - (1 - \bar{q}_s_{i,j}) \cdot M, \\
&\qquad \sum_{d \in D} S_{(i,j),k} \le q_s_{i,j} \cdot M, \\
&\qquad\qquad \bar{q}_s_{i,j} = q_s_{i,j} \\
&\qquad\qquad x^{ru}_i \le x^{lb}_j + (1 - q_r_d_{i,j} + \bar{q}_s_{i,j}) \cdot M, \\
&\qquad\qquad x^{ru}_j \le x^{lb}_i + (1 - q_l_d_{i,j} + \bar{q}_s_{i,j}) \cdot M, \\
&\qquad\qquad y^{ru}_i \le y^{lb}_j + (1 - q_u_d_{i,j} + \bar{q}_s_{i,j}) \cdot M, \\
&\qquad\qquad y^{ru}_j \le y^{lb}_i + (1 - q_b_d_{i,j} + \bar{q}_s_{i,j}) \cdot M, \\
&q_r_d_{i,j} + q_l_d_{i,j} + q_u_d_{i,j} + q_b_d_{i,j} \ge 1 - \bar{q}_s_{i,j} \cdot M, \\
&\qquad\qquad x^{ru}_s \le x^{lb}_j + q_r_s_{s,j} \cdot M, \\
&\qquad\qquad x^{ru}_j \le x^{lb}_s + q_l_s_{s,j} \cdot M, \\
&\qquad\qquad y^{ru}_s \le y^{lb}_j + q_u_s_{s,j} \cdot M, \\
&\qquad\qquad y^{ru}_j \le y^{lb}_s + q_b_s_{s,j} \cdot M, \\
&q_r_s_{s,j} + q_l_s_{s,j} + q_u_s_{s,j} + q_b_s_{s,j} \le 3,
\end{aligned} \tag{3.4}$$

where $q\_r\_d_{i,j}$, $q\_l\_d_{i,j}$, $q\_u\_d_{i,j}$, $q\_b\_d_{i,j}$, $q\_r\_s_{s,j}$, $q\_l\_s_{s,j}$, $q\_u\_s_{s,j}$, $q\_b\_s_{s,j}$ are 0-1 variables indicating whether devices and storage overlap in the right, left, up, or down directions, respectively. Here, binary variables $q\_s_{i,j}$, $\bar{q}\_s_{i,j}$ indicates whether $\sum_{d \in D} S_{(i,j),k}$

is larger than zero or not. Further, constraint (11) ensures $S_{(i,j),k}$ is a 0-1 variable representing whether o_i and o_j are bound to a device d_k, which is correct in [1] and omitted in this section.

Constraint (12) in [1] ensures that if two operations o_i and o_j are bound to the same device, and o_i executed first, the execution of o_j can only begin once the fluids from o_i have been fully removed. We linearize constraint (12) using big M method [11] as follows:

$$\begin{aligned}
&\forall e_{u,j}, e_{i,h} \in E : \\
&\qquad t_j^s \le t_i^e + q_ex_{i,j} \cdot M, \\
&\qquad t_{P_{u,j,2}}^s \ge t_{P_{i,h,1}}^s - (2 - q_s_{i,j} - q_ex_{i,j}) \cdot M, \\
&\qquad t_{P_{u,j,2}}^s \ge t_{P_{i,h,3}}^s - (2 - q_s_{i,j} - q_ex_{i,j}) \cdot M, \\
&\qquad t_{P_{u,j,3}}^s \ge t_{P_{i,h,1}}^s - (2 - q_s_{i,j} - q_ex_{i,j}) \cdot M, \\
&\qquad t_{P_{u,j,3}}^s \ge t_{P_{i,h,2}}^s - (2 - q_s_{i,j} - q_ex_{i,j}) \cdot M,
\end{aligned} \tag{3.5}$$

where $q\_ex_{i,j}$ is 0-1 variable representing the sequential execution order of o_i and o_j on the same device.

Constraints (13), (14), (15), (16), and (17) in [1] are part of the binding and placement task. Their functionalities are described as follows:
1) Constraint (13) ensures that if a device d_k is assigned to an operation o_i in the binding scheme, the placement of d_k on the chip must match its dimensions, where w_k and h_k are the width and height of d_k;
2) Constraint (14) ensures that if a storage is needed, the placement of storage on the chip must match its dimensions, where w_s and h_s are the width and height of storage;
3) Constraint set 15 ensures that l_s is a 0-1 variable representing whether a storage is introduced;
4) Constraint (16) ensures that the total area occupied by the allocated devices in the binding stage does not exceed the available chip area;
5) Constraint (17) ensures that l_k is a 0-1 variable indicating whether d_k is allocated for executing operations [1]. Since the mathematical expressions in [1] are accurate, they are omitted in this section.

Constraints (18)–(23) in [1] are part of the routing task and represent the most complex portion of BigIntegr, as well as the least clearly described section in [1]. To improve and express constraint (18) more accurately, this study first introduces a method to represent $g_{x,y} \in N_{d_k}$, For any grid $g_{x,y}$ not occupied by a device, if any of its neighboring grids is occupied by device d_k, then $g_{x,y}$ is considered a neighbor of device d_k. The neighboring grids of $g_{x,y}$ refer to the four adjacent grids (above, below, left, and right). Thus, constraint (18) can be revised and expanded as follows:

- these inequalities firstly ensure that binary variable $g_{x,y}^{d(o_j)}$ indicates whether a grid $g_{x,y}$

is occupied by a device $d(o_j)$:

$$\begin{aligned}
\forall e_{j,i} \in E, \quad \forall g_{x,y} \in R: \\
x &\geq x_j^{ru} + u - q_g_l_{x,y}^{d(o_j)} \cdot M, \\
y &\geq y_j^{ru} + u - q_g_b_{x,y}^{d(o_j)} \cdot M, \\
x &\leq x_j^{lb} + q_g_r_{x,y}^{d(o_j)} \cdot M, \\
y &\leq y_j^{lb} + q_g_u_{x,y}^{d(o_j)} \cdot M, \\
g_{x,y}^{d(o_j)} &\geq q_g_l_{x,y}^{d(o_j)} + q_g_b_{x,y}^{d(o_j)} + q_g_r_{x,y}^{d(o_j)} + q_g_u_{x,y}^{d(o_j)} - 3,
\end{aligned} \tag{3.6}$$

- these inequalities ensure that binary variable $q\_N_{x,y}^{d(o_j)}$ is 0-1 variable indicating whether a grid $g_{x,y}$ is neighbor of device $d(o_j)$, i.e. $g_{x,y} \in N_{d(o_j)}$:

$$\begin{aligned}
x &\leq x_j^{ru} + \bar{g}_{x,y}^{d(o_j)} \cdot M, \\
y &\leq y_j^{ru} + \bar{g}_{x,y}^{d(o_j)} \cdot M, \\
x - u &\leq x_j^{lb} - \bar{g}_{x,y}^{d(o_j)} \cdot M, \\
y - u &\leq y_j^{lb} - \bar{g}_{x,y}^{d(o_j)} \cdot M, \\
g_{x-u,y}^{d(o_j)} + g_{x+u,y}^{d(o_j)} + g_{x,y-u}^{d(o_j)} + g_{x,y+u}^{d(o_j)} &\leq q_N_\exists N_{x,y}^{d(o_j)} \cdot M, \\
g_{x-u,y}^{d(o_j)} + g_{x+u,y}^{d(o_j)} + g_{x,y-u}^{d(o_j)} + g_{x,y+u}^{d(o_j)} &\geq 1 - (1 - q_N_\exists N_{x,y}^{d(o_j)}) \cdot M, \\
q_N_{x,y}^{d(o_j)} &\geq \bar{g}_{x,y}^{d(o_j)} + q_N_\exists N_{x,y}^{d(o_j)} - 1, \\
q_N_{x,y}^{d(o_j)} &\leq q_N_\exists N_{x,y}^{d(o_j)}, \\
\bar{q}_N_{x,y}^{d(o_j)} &\geq \bar{g}_{x-u,y}^{d(o_j)} + \bar{g}_{x+u,y}^{d(o_j)} + \bar{g}_{x,y-u}^{d(o_j)} + \bar{g}_{x,y+u}^{d(o_j)} + \bar{g}_{x,y}^{d(o_j)} - 5, \\
\bar{g}_{x,y}^{d(o_j)} + g_{x,y}^{d(o_j)} &= 1, \\
q_N_{x,y}^{d(o_j)} + \bar{g}_{x,y}^{d(o_j)} &= 1,
\end{aligned} \tag{3.7}$$

- these inequalities ensure that $\sum_{g_{x,y} \in N_{d(o_j)}} g_{x,y}^{P_{j,i,1}} = 1$:

$$\begin{aligned}
q^{d(o_j)}_N_g_{x,y}^{P_{j,i,1}} &\leq g_{x,y}^{P_{j,i,1}}, \\
q^{d(o_j)}_N_g_{x,y}^{P_{j,i,1}} &\leq q_N_{x,y}^{d(o_j)}, \\
q^{d(o_j)}_N_g_{x,y}^{P_{j,i,1}} &\geq g_{x,y}^{P_{j,i,1}} + q_N_{x,y}^{d(o_j)} - 1, \\
Sum^{d(o_j)}_N_g_{x,y}^{P_{j,i,1}} &= \sum_{g_{x,y} \in R} q^{d(o_j)}_N_g_{x,y}^{P_{j,i,1}}, \\
Sum^{d(o_j)}_N_g_{x,y}^{P_{j,i,1}} &\leq 1 + (1 - \rho_{j,i}) \cdot M, \\
Sum^{d(o_j)}_N_g_{x,y}^{P_{j,i,1}} &\geq 1 - (1 - \rho_{j,i}) \cdot M,
\end{aligned} \tag{3.8}$$

- similarly, these inequalities ensure that $\sum_{g_{x,y} \in N_{ds}} g_{x,y}^{P_{j,i,1}} = 1$:

$$
\begin{aligned}
x &\geq x_s^{ru} + u - q\_g\_l_{x,y}^{ds} \cdot M, \\
y &\geq y_s^{ru} + u - q\_g\_b_{x,y}^{ds} \cdot M, \\
x &\leq x_s^{lb} + q\_g\_r_{x,y}^{ds} \cdot M, \\
y &\leq y_s^{lb} + q\_g\_u_{x,y}^{ds} \cdot M, \\
g_{x,y}^{ds} &\geq q\_g\_l_{x,y}^{ds} + q\_g\_b_{x,y}^{ds} + q\_g\_r_{x,y}^{ds} + q\_g\_u_{x,y}^{ds} - 3, \\
x &\leq x_s^{ru} + \bar{g}_{x,y}^{ds} \cdot M, \\
y &\leq y_s^{ru} + \bar{g}_{x,y}^{ds} \cdot M, \\
x - u &\leq x_s^{lb} - \bar{g}_{x,y}^{ds} \cdot M, \\
y - u &\leq y_s^{lb} - \bar{g}_{x,y}^{ds} \cdot M, \\
g_{x-u,y}^{ds} + g_{x+u,y}^{ds} + g_{x,y-u}^{ds} + g_{x,y+u}^{ds} &\leq q\_N\_\exists N_{x,y}^{ds} \cdot M, \\
g_{x-u,y}^{dS} + g_{x+u,y}^{dS} + g_{x,y-u}^{dS} + g_{x,y+u}^{dS} &\geq 1 - (1 - q\_N\_\exists N_{x,y}^{ds}) \cdot M, \\
q\_N_{x,y}^{ds} &\geq \bar{g}_{x,y}^{ds} + q\_N\_\exists N_{x,y}^{ds} - 1, \\
q\_N_{x,y}^{ds} &\leq q\_N\_\exists N_{x,y}^{ds}, \\
\bar{q}\_N_{x,y}^{ds} &\geq \bar{g}_{x-u,y}^{ds} + \bar{g}_{x+u,y}^{ds} + \bar{g}_{x,y-u}^{ds} + \bar{g}_{x,y+u}^{ds} + \bar{g}_{x,y}^{ds} - 5, \\
\bar{g}_{x,y}^{ds} + g_{x,y}^{ds} &= 1, \\
q\_N_{x,y}^{ds} + \bar{g}_{x,y}^{ds} &= 1, \\
q^{ds}\_N\_g_{x,y}^{P_{j,i,1}} &\leq g_{x,y}^{P_{j,i,1}}, \\
q^{ds}\_N\_g_{x,y}^{P_{j,i,1}} &\leq q\_N_{x,y}^{ds}, \\
q^{ds}\_N\_g_{x,y}^{P_{j,i,1}} &\geq g_{x,y}^{P_{j,i,1}} + q\_N_{x,y}^{ds} - 1, \\
Sum^{ds}\_N\_g_{x,y}^{P_{j,i,1}} &= \sum_{g_{x,y} \in R} q^{ds}\_N\_g_{x,y}^{P_{j,i,1}}, \\
Sum^{ds}\_N\_g_{x,y}^{P_{j,i,1}} &\leq 1 + (1 - \rho_{j,i}) \cdot M, \\
Sum^{ds}\_N\_g_{x,y}^{P_{j,i,1}} &\geq 1 - (1 - \rho_{j,i}) \cdot M.
\end{aligned}
\tag{3.9}
$$

The meaning of variables symbols are described as follows:

- $q\_g\_l_{x,y}^{d(o_j)}$, $q\_g\_b_{x,y}^{d(o_j)}$, $q\_g\_r_{x,y}^{d(o_j)}$, $q\_g\_u_{x,y}^{d(o_j)}$ are 0-1 variables indicating whether a grid is positioned to the left, right, above, or below a device's right, lower, left, or upper boundaries, respectively.

- $g_{x,y}^{d(o_j)}$ is 0-1 variable indicating whether a grid $g_{x,y}$ is occupied by a device $d(o_j)$.

- $\bar{g}_{x,y}^{d(o_j)}$ is 0-1 variable indicating whether a grid $g_{x,y}$ is not occupied by a device $d(o_j)$.

- u denotes the length of a grid cell, which is typically defined as 1 unit.
- x, y is the cartesian coordinates of the upper-right corner of the grid, and $(x-u, y)$, $(x+u, y)$, $(x, y-u)$, $(x, y+u)$ represents the cartesian coordinates of the upper-right corners of the neighboring grids of $g_{x,y}$.
- $q\_N\_\exists N_{x,y}^{d(o_j)}$ is 0-1 variable indicating whether there exists a neighbor grid of a grid $g_{x,y}$ is neighbor of device $d(o_j)$.
- $q\_N_{x,y}^{d(o_j)}$ is 0-1 variable indicating whether a grid $g_{x,y}$ is neighbor of device $d(o_j)$.
- $\bar{q}\_N_{x,y}^{d(o_j)}$ is 0-1 variable indicating whether a grid $g_{x,y}$ is not neighbor of device $d(o_j)$.
- $q^{d(o_j)}\_N\_g^{P_{j,i,k}}$ is 0-1 variable representing whether a grid $g_{x,y}$ is neighbor of device $d(o_j)$ and occupied by transportation $P_{j,i,k}$ with $k = 1, 2, 3$.
- $Sum^{d(o_j)}\_N\_g_{x,y}^{P_{j,i,k}}$ with $k = 1, 2, 3$ is integer variable indicating sum of $q^{d(o_j)}\_N\_g^{P_{j,i,k}}$ with $k = 1, 2, 3$ of all grids in chip.
- symbols with indices such as ds have similar meanings to the aforementioned variables but refer to storage instead of the device $d(o_j)$.
- symbols with indices such as $d(o_i)$ have similar meanings to the aforementioned variables but refer to device that has parent operations instead of the device $d(o_j)$ that has child operations.

It is important to note that the constraints applied to $d(o_j)$ must also be applied to $d(o_i)$, although they are not listed here.

Constraint (19) in [1] is also unclear. Based on the representation of $g_{x,y} \in N_{d_k}$ mentioned in constraint (18), we introduce the following constraints to represent $g_{x,y} \notin N_{d(o_j)} \cup N_S$.

$$\begin{aligned} &\forall e_{j,i} \in E, \quad \forall g_{x,y} \in R: \\ &\qquad q\_notN_{x,y}^{N_{d(o_j)},N_S} \geq \bar{q}\_N_{x,y}^{d(o_j)} + \bar{q}\_N_{x,y}^{ds} - 1, \\ &\qquad q\_notN_{x,y}^{N_{d(o_j)},N_S} \leq \bar{q}\_N_{x,y}^{d(o_j)}, \\ &\qquad q\_notN_{x,y}^{N_{d(o_j)},N_S} \leq \bar{q}\_N_{x,y}^{ds}, \end{aligned} \tag{3.10}$$

where $q\_notN_{x,y}^{N_{d(o_j)},N_S}$ is 0-1 variable indicating $g_{x,y} \notin N_{d(o_j)} \cup N_S$. After that, constraint (19) can be described as

$$\begin{aligned} g_{x-u,y}^{P_{j,i,1}} + g_{x+u,y}^{P_{j,i,1}} + g_{x,y-u}^{P_{j,i,1}} + g_{x,y+u}^{P_{j,i,1}} &\leq 2 + (2 - g_{x,y}^{P_{j,i,1}} - q\_notN_{x,y}^{N_{d(o_j)},N_S}) \cdot M, \\ g_{x-u,y}^{P_{j,i,1}} + g_{x+u,y}^{P_{j,i,1}} + g_{x,y-u}^{P_{j,i,1}} + g_{x,y+u}^{P_{j,i,1}} &\geq 2 - (2 - g_{x,y}^{P_{j,i,1}} - q\_notN_{x,y}^{N_{d(o_j)},N_S}) \cdot M. \end{aligned} \tag{3.11}$$

Constraints (20) and (21) in [1] have also been improved. These constraints are similar to constraints (18) and (19), representing the routing method between device $d(o_j)$ and the storage, but specifically for transaction $P_{j,i,2}$ instead of $P_{j,i,1}$. Therefore, the details are not elaborated here.

Constraints (22) and (23) in [1] have likewise been refined. These constraints are slightly different to constraints (18) and (19), representing the routing method between device $d(o_j)$ and device $d(o_i)$ regarding transaction $P_{j,i,3}$. Thus, Constraint (22) can be formulated as

$$
\begin{aligned}
&\forall e_{j,i} \in E, \quad \forall g_{x,y} \in R: \\
&\qquad q^{d(o_j)}\_N\_g_{x,y}^{P_{j,i,3}} \le g_{x,y}^{P_{j,i,3}}, \\
&\qquad q^{d(o_j)}\_N\_g_{x,y}^{P_{j,i,3}} \le q\_N_{x,y}^{d(o_j)}, \\
&\qquad q^{d(o_j)}\_N\_g_{x,y}^{P_{j,i,3}} \ge g_{x,y}^{P_{j,i,3}} + q\_N_{x,y}^{d(o_j)} - 1, \\
&\qquad Sum^{d(o_j)}\_N\_g_{x,y}^{P_{j,i,3}} = \sum_{g_{x,y} \in R} q^{d(o_j)}\_N\_g_{x,y}^{P_{j,i,3}}, \\
&\qquad Sum^{d(o_j)}\_N\_g_{x,y}^{P_{j,i,3}} \le 1 + (\bar{q}\_s_{i,j} + \rho_{j,i}) \cdot M, \\
&\qquad Sum^{d(o_j)}\_N\_g_{x,y}^{P_{j,i,3}} \ge 1 - (\bar{q}\_s_{i,j} + \rho_{j,i}) \cdot M, \\
&\qquad q^{d(o_i)}\_N\_g_{x,y}^{P_{j,i,3}} \le g_{x,y}^{P_{j,i,3}}, \\
&\qquad q^{d(o_i)}\_N\_g_{x,y}^{P_{j,i,3}} \le q\_N_{x,y}^{d(o_i)}, \\
&\qquad q^{d(o_i)}\_N\_g_{x,y}^{P_{j,i,3}} \ge g_{x,y}^{P_{j,i,3}} + q\_N_{x,y}^{d(o_i)} - 1, \\
&\qquad Sum^{d(o_i)}\_N\_g_{x,y}^{P_{j,i,3}} = \sum_{g_{x,y} \in R} q^{d(o_i)}\_N\_g_{x,y}^{P_{j,i,3}}, \\
&\qquad Sum^{d(o_i)}\_N\_g_{x,y}^{P_{j,i,3}} \le 1 + (\bar{q}\_s_{i,j} + \rho_{j,i}) \cdot M, \\
&\qquad Sum^{d(o_i)}\_N\_g_{x,y}^{P_{j,i,3}} \ge 1 - (\bar{q}\_s_{i,j} + \rho_{j,i}) \cdot M.
\end{aligned}
\tag{3.12}
$$

Constraint (23) can be formulated as

$$
\begin{aligned}
&\forall e_{j,i} \in E, \quad \forall g_{x,y} \in R: \\
&q\_notN_{x,y}^{N_{d(o_j)},N_{d(o_i)}} \ge \bar{q}\_N_{x,y}^{d(o_j)} + \bar{q}\_N_{x,y}^{d(o_i)} - 1, \\
&q\_notN_{x,y}^{N_{d(o_j)},N_{d(o_i)}} \le \bar{q}\_N_{x,y}^{d(o_j)}, \\
&q\_notN_{x,y}^{N_{d(o_j)},N_{d(o_i)}} \le \bar{q}\_N_{x,y}^{d(o_i)}, \\
&g_{x-u,y}^{P_{j,i,3}} + g_{x+u,y}^{P_{j,i,3}} + g_{x,y-u}^{P_{j,i,3}} + g_{x,y+u}^{P_{j,i,3}} \le 2 + (2 - g_{x,y}^{P_{j,i,1}} - q\_notN_{x,y}^{N_{d(o_j)},N_{d(o_i)}}) \cdot M, \\
&g_{x-u,y}^{P_{j,i,3}} + g_{x+u,y}^{P_{j,i,3}} + g_{x,y-u}^{P_{j,i,3}} + g_{x,y+u}^{P_{j,i,3}} \ge 2 - (2 - g_{x,y}^{P_{j,i,3}} - q\_notN_{x,y}^{N_{d(o_j)},N_{d(o_i)}}) \cdot M,
\end{aligned}
\tag{3.13}
$$

where $q\_notN_{x,y}^{N_{d(o_j)},N_{d(o_i)}}$ is 0-1 variable indicating $g_{x,y} \notin N_{d(o_j)} \cup N_{d(o_i)}$.

Constraints (24), (25) in [1] are part of the scheduling task. Their functionalities are described as follows:

1) Constraint (24) ensures that scheduling should be linked to the routing results to accurately calculate the transportation latencies $P_{j,i,k}$ between devices when determining the start times of operations;
2) Constraint (25) ensures that transportation tasks sharing the same grid cells cannot be executed simultaneously; Since the mathematical expressions in [1] are accurate, they are omitted in this section.

Constraints (26) and (27) in [1] are part of the routing and placement tasks. Constraints (26) ensures that flow channels cannot pass through grid cells that are already occupied by devices during the routing process. However, the constraints presented in [1] lack clarity and do not account for storage considerations. Therefore, this study corrects and improves constraint (26) as

$$\begin{aligned} &\forall o_j, o_i \in O, \quad \forall g_{x,y} \in R: \\ &\quad g_{x,y}^{d(o_j)} + g_{x,y}^{c} + g_{x,y}^{d_S} \leq 1 \\ &\quad g_{x,y}^{d(o_i)} + g_{x,y}^{c} + g_{x,y}^{d_S} \leq 1, \end{aligned} \tag{3.14}$$

and correct constraint (27) as

$$\begin{aligned} &\forall g_{x,y} \in R, \quad \forall e_{j,i} \in E: \\ &\quad 1 - (1 - g_{x,y}^{c}) \cdot M \leq \sum_{e_{j,i} \in E} (g_{x,y}^{P_{j,i,1}} + g_{x,y}^{P_{j,i,2}} + g_{x,y}^{P_{j,i,3}}), \\ &\sum_{e_{j,i} \in E} (g_{x,y}^{P_{j,i,1}} + g_{x,y}^{P_{j,i,2}} + g_{x,y}^{P_{j,i,3}}) \leq g_{x,y}^{c} \cdot M, \end{aligned} \tag{3.15}$$

where $g_{x,y}^{c}$ is a 0-1 variables representing whether grid $g_{x,y}$ is occupied by a flow channel.

Constraints (28) and (29) in [1] represent the optimization objectives and are formulated in this study as

$$min(\alpha t^{E} + \beta \sum_{g_{x,y}} g_{x,y}^{c} + \gamma \sum_{g_{x,y}} g_{x,y}^{r}), \tag{3.16}$$

where α, β and γ are three weighting factors and are set to 0.3, 0.3, and 0.4 respectively.

Constraints (30), (31), and (32) in [1] are additional improvements and supplements introduced in this study, ensuring that $g_{x,y}^{r}$ is a 0-1 variable representing whether grid $g_{x,y}$ forms a channel intersection on the chip. Constraints (30) and (31) can be formulated as follows:

$$\begin{aligned} &\forall e_{j,i}, \quad e_{j_1,i_1}, e_{j_2,i_2} \in E, \quad \forall g_{x,y} \in R, \quad \forall g_{x',y'} \in N_{g_{x,y}}: \\ &q^{g_{x-u,y}} \_c_{i_2,j_2}^{i_1,j_1} + q^{g_{x+u,y}} \_c_{i_2,j_2}^{i_1,j_1} + q^{g_{x,y-u}} \_c_{i_2,j_2}^{i_1,j_1} + q^{g_{x,y+u}} \_c_{i_2,j_2}^{i_1,j_1} \leq (1 - q^{g_{x,y}} \_cross_{i_2,j_2}^{i_1,j_1}) \cdot M, \\ &q^{g_{x-u,y}} \_c_{i_2,j_2}^{i_1,j_1} + q^{g_{x+u,y}} \_c_{i_2,j_2}^{i_1,j_1} + q^{g_{x,y-u}} \_c_{i_2,j_2}^{i_1,j_1} + q^{g_{x,y+u}} \_c_{i_2,j_2}^{i_1,j_1} \geq 1 - q^{g_{x,y}} \_cross_{i_2,j_2}^{i_1,j_1} \cdot M, \end{aligned}$$

$$
\begin{aligned}
q^{g_{x,y}}\_c_{i_2,j_2}^{i_1,j_1} &\geq 1-(2-g_{(x,y),(i_1,j_1)}^{c}-g_{(x,y),(i_2,j_2)}^{c})\cdot M,\\
g_{(x,y),(i_1,j_1)}^{c}+g_{(x,y),(i_2,j_2)}^{c} &\geq 2-(1-q^{g_{x,y}}\_c_{i_2,j_2}^{i_1,j_1})\cdot M,\\
g_{x,y}^{r} &\leq q^{g_{x,y}}\_cross_{i_2,j_2}^{i_1,j_1},\\
g_{x,y}^{r} &\leq q^{g_{x,y}}\_c_{i_2,j_2}^{i_1,j_1},\\
g_{x,y}^{r} &\geq q^{g_{x,y}}\_cross_{i_2,j_2}^{i_1,j_1}+q^{g_{x,y}}\_c_{i_2,j_2}^{i_1,j_1}-1,\\
g_{x,y}^{P_{j,i,1}}+g_{x,y}^{P_{j,i,2}}+g_{x,y}^{P_{j,i,3}} &\leq g_{(x,y),(i,j)}^{c}\cdot M,\\
g_{x,y}^{P_{j,i,1}}+g_{x,y}^{P_{j,i,2}}+g_{x,y}^{P_{j,i,3}} &\geq 1-(1-g_{(x,y),(i,j)}^{c})\cdot M.
\end{aligned}
\tag{3.17}
$$

The meaning of variables symbols are described as follows:

- $g_(x-u,y)$, $g_(x+u,y)$, $g_(x,y-u)$, $g_(x,y+u)$ represents neighboring grids of $g_{x,y}$, i.e. $g_{x',y'} \in N_{g_{x,y}}$. $N_{g_{x,y}}$ is the set of neighbor cells of $g_{x,y}$ on the grid.

- $q^{g_{x,y}}\_c_{i_2,j_2}^{i_1,j_1}$ is 0-1 variables representing if grid $g_{x,y}$ is occupied by channel for transportation $P_{i_1,j_1,k}$ and $P_{i_2,j_2,k}$.

- $g_{(x,y),(i,j)}^{c}$ is 0-1 variables representing if grid $g_{x,y}$ is occupied by channel for transportation $P_{i,j,k}$.

- $q^{g_{x,y}}\_cross_{i_2,j_2}^{i_1,j_1}$ is 0-1 variable representing if each neighboring grid $g_{x',y'}$ of $g_{x,y}$ is occupied for no more than one transportation, e.g. transportation $P_{i_1,j_1,k}$ and $P_{i_2,j_2,k}$.

- $g_{x,y}^{r}$ is 0-1 variable representing whether grid cell forms a channel intersection.

Constraint (32) addresses and restricts the corner cases where $g_(x',y')$ is outside the chip boundaries, and these limitations can be achieved by setting the boundaries for Gurobi variables.

The improved constraints mentioned above constitute the ILP mathematical model for the one-pass synthesis flow.

3.2. Corner cases analysis

The model presented in [1] does not account for certain corner cases. For example, constraint (12) assumes that every operation has both input and output, without considering input/output devices. Such devices only have input or output, and thus the model needs to be adjusted. Additionally, constraint (26) does not take into account that storage also occupies space, which has been corrected in this study.

Furthermore, constraint (30) does not introduce or restrict $g^c_{(x,y),(i,j)}$. Therefore, this study introduces constraint (31) to limit $g^c_{(x,y),(i,j)}$, which is defined as a 0-1 variable representing whether grid $g_{x,y}$ is occupied by a channel for transportation $P_{i,j,k}$.

4. Implementation of design automation softawre based on BigIntegr

In this section, the methods and details of developing design automation software based on BigIntegr using C++ and Gurobi [10] are presented. The software takes text data as input, which must be formatted to include device information, the protocol of the assay to be realized, modeled as a directed acyclic graph $G(O, E)$ and design constraints like maximal chip area. Based on the input, the software automatically generates the biochip architecture as required and outputs the results in text format.

4.1. Environment configurations

To successfully develop and execute the proposed design automation software, it is essential to first consider the code compilation platform, compiler, code editing tools, and the installation and use of Gurobi. Accordingly, the objectives of the aforementioned environment configurations can be summarized as follows:

- In this study, Ubuntu 20.04 is chosen as the compilation platform because, when using common g++ compilers such as MinGW64 on Windows 10, incompatibilities with the Gurobi solver library such as undefined reference may arise. Based on testing and information from the official documentation, using Linux platform as the compilation platform eliminates these compatibility issues with Gurobi [10].

- For the g++ compiler, this study utilizes GCC version 9.4.0 (Ubuntu 9.4.0-1ubuntu1 20.04) to leverage the features of C++11. If the g++ compiler version is too low, errors may occur during the compilation of libraries or header files. To avoid such issues, it is recommended to use GCC version 4.8.0 or higher. The command `g++ -v` can be used in the Linux terminal to verify the successful installation of the compiler and to check the compiler version.

- This study utilizes Visual Studio Code (VS Code) as the code editor due to its support for various extensions that enable real-time detection of syntax and spelling errors, as well as its efficiency in managing C++ source and header files. Upon initial configuration, it is necessary to include the Gurobi library in the compiler's search path and specify the g++ compiler to prevent compilation errors.

- The installation of Gurobi [10] requires the registration of an academic account on the official website, followed by obtaining a license. Upon downloading and extracting the

package, the environment variables must be configured in accordance with the official documentation, and the license must be activated. To utilize Gurobi [10], the path to the Gurobi library should be appropriately included in the g++ compilation commands. For instance, the command used to compile the code into an executable file is as follows:

```
g++ -std=c++11 -m64 -g -o BigInteger_3 MILP_BigInteger_3.cpp
-I/home/ge23qoc/FP_code/gurobi11.0.2_linux64/gurobi1102/linux64/include
-L/home/ge23qoc/FP_code/gurobi11.0.2_linux64/gurobi1102/linux64/lib
-lgurobi_c++ -lgurobi110 -lpthread -lm
```

Once the development environment is configured and the necessary tools are installed, program development can commence.

4.2. Software development

The successful design and development of chip design automation software using C++ requires careful consideration of several components, including the necessary C++ libraries, custom header files developed for this study, CPP files based on BigIntegr, and input/output files. The development process for this software can be summarized in four key steps:

1) Identify and select the required C++ standard libraries, and create custom header files to support the implementation of core software functions.

2) Since the input data is provided in the form of text files, it is essential to define a specific input format to streamline the development of data reading functions.

3) Develop the main CPP files based on BigIntegr to implement the essential functionality of the software.

4) Design test cases and generate output files for validation and performance assessment.

4.2.1. Design of custom header files

The header file should include the following components:

- The necessary C++ standard libraries required for the software.
- Gurobi variable Class definitions needed for integration with the Gurobi solver.
- Various class definitions necessary for the implementation of BigIntegr.
- Organization and declaration of variables used to store design results at each stage of the complete synthesis flow in each class.

- Declarations of relevant functions and global variables, including the global parameters required for the Gurobi solver's optimization process.

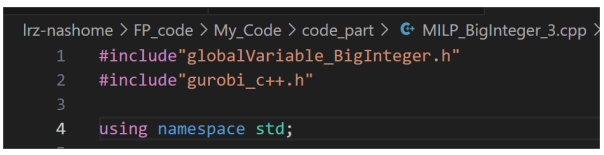
Thus, two header files named `class_BigInteger.h` and `globalVariable_BigInteger.h` are created to develop the software.

As illustrated in Figure 4.1, the C++ standard libraries required for the software, primarily sourced from the Standard Template Library (STL), have been incorporated into `class_BigInteger.h`. These libraries are essential for defining the variables necessary for the model's construction and for enabling the utilization of various functions required throughout the development process.

Figure 4.1.: The necessary C++ standard libraries required for the software.

To avoid multiple inclusion of header files, a header guard can be employed, or alternatively, as shown in Figure 4.2, `class_BigInteger.h` can be included within `globalVariable_BigInteger.h`. The main CPP file of the software only needs to include `globalVariable_BigInteger.h` and the Gurobi C++ library header file `gurobi_c++.h`.

(a) header files of `globalVariable_BigInteger.h`



(b) header files of `BigInteger.cpp`

Figure 4.2.: Header files included in each files.

After including the necessary libraries, as shown in Figure 4.3, the variable classes required for the Gurobi solver, as well as other classes, where variable members used to store design results at each stage of the complete synthesis flow, must be declared.

```
/* class  declaration */
class var;
class operation;
class device;
class storage;
class grid;
class chip;

/* part of gurobi API, don't change it */
class var
{
        public:
                var();
                int typ, index; // Variable type (0: binary, 1: integer, 2: continuous) and index
                double lb, ub;// Lower bound (lb) and upper bound (ub) of the variable
                set<int> cons_i;// Set of constraint indices associated with this variable
                // Map of constraint indices to their coefficients
                map<int,double> coef_cons; // cons_i, coef
                int intrval; // Integer value of the variable
                double dourval; // Double value of the variable
                /* quadratic */
                map<int, set<var*>> coef_setvar;
                map<pair<int, var*>, double> coef_varcons;
                string var_name; // variable name
};
var::var()
{
        typ=-1;
        intrval=-1;
        dourval=-1;
}
```

Figure 4.3.: Class declaration.

Figure 4.3 illustrates the implementation of the variable class required by the Gurobi solver. Each variable has its own type and value boundaries, and the results computed by the Gurobi solver are stored in the intrval member.

The classes declared for operations, devices, storage, grids and chip primarily include collections of Gurobi variables, categorized and established according to the mathematical expressions of BigIntegr. These variables are instantiated in the `BigInteger.cpp` file and are used to store the design results at each stage of the complete synthesis flow.

Taking the device class as an example, as illustrated in Figure 4.4, the device class includes variables for storing information such as the name and dimensions of a device. It also contains Gurobi variables representing the chip architecture design results associated with the device. In the inputRead function of the `BigInteger.cpp` file, these classes are instantiated based on the input to represent specific devices. The final design results for each device, at every stage of the complete synthesis flow, are stored in the Gurobi variables associated with the respective device instances.

Declarations of relevant functions and global variables, including the global parameters required for the Gurobi solver's optimization process, are included in `globalVariable_BigInteger.h`.

```
class device
{
        public:
                device();
                /* input information that helps generating variables and constraints */
                //constant input needed in constraints 13
                //updated in funcRead
                string name; //device name
                string device_type;
                int deviceW, deviceH; //width and Height of device

                /**
                 * model variables
                 * variables in map created by funcPrepare
                 * */
                // model variables for constraint 9
                // x^lb_d, x^ru_d, y^lb_d, y^ru_d
                var *x_lb, *x_ru, *y_lb, *y_ru; //belowLeft point and upperRight point address
                // model variables for constraint 10, 11, 22
                // S_i_j_dk
                map<pair<operation*, operation*>, var*> sameBound_i_j_k; //binary variable
                // model variables for constraint 16, 17
                // l_k
                var *deviceUsed; //binary variable showing if the device are used
};
device::device()
{
        deviceW = -1;
        deviceH = -1;
}
```

Figure 4.4.: Device class declaration.

As shown in Figure 4.5, eight functions related to the creation of BigIntegr are declared. These functions are used for reading from the input file, creating and modifying objects not generated by the input, generating model variables, building model constraints, displaying debugging information on the screen, displaying output information on the screen, writing output to files, and invoking the Gurobi solver to obtain the design results.

```
#include"class_BigInteger.h"
/* function declaration */
void funcReadFile(const char*, const char*); //read from input file
void funcPrepare(); //create objects that not created by read and modify
void funcModel_Var(); // generate model variables
void funcModel_Cons();// build model constraints
void funcScreenMessage_readPart(); //read info on screen
void funcScreenMessage_outputPart(); //output info on screen
void funcWriteFile(const char*); //write output into files
void funcGurobi(int, double, double, double, double, int, int);//gurobi api decalration
```

Figure 4.5.: Function declaration.

Figure 4.6 illustrates the global variables required for building the software, as well as the parameters and global variables necessary for constructing the constraints of BigIntegr. The instances generated by the previously mentioned classes are stored within the collections repre-

sented by these global variables. Additionally, the information pertaining to BigIntegr is stored in the relevant global variables. This information forms the basis for the one-pass synthesis flow used by the Gurobi solver to solve the model. Once the Gurobi solver is invoked, the design results at each stage of the complete synthesis flow are stored in the Gurobi variables contained within the instances held by the global variables.

```
/* global funcModel_Var */
var *minExeTime; //t^E: part of optimization objective

/* funcReadFile */
//chip object
chip *chipObject;
//operation set
set<operation*> setOperation; //all operations needed in the flow
map<string, operation*> operationMap; //identify operation according to key: operation name string
pair<operation*, operation*> edge;
set<pair<operation*, operation*>> setEdge; //edge from input file specifies the dependency between operations
//device set
set<device*> setDevice; //all available devices in the flow
map<string, device*> deviceMap; //identify device according to key: device name string
//storage
storage *storageObject;
//grid set
grid *generalGrid; //used for store info to help generate real grid
set<grid*> setGrid; //set contains all grid
set<pair<int, int>> gridAddress; //grid cell upper right point address
map<pair<int, int>, grid*> gridMap; //identify grid according to key: element of set gridAddress



/* MILP Model parameters */
int nCons, objcons;
map<int,int> mapcs; // use cons index to map sense: 1=greater_equal, 0=equal, -1=less_equal
map<int,double> mapcc; // use cons index to map right side constant// constants on the right side of constraints to corresponding cons
map<int,set<var*> > mapcv; // use cons index to map corresponding vars//contained variables to corresponding constraints(constraint ir
vector<var*> vvar; //whole variables in model
set<int> qcons;
```


Figure 4.6.: Global variables.

4.2.2. Design of input formats

The input for the chip design automation software based on the BigIntegr is provided in a specific text format. Figure 4.7 presents an example of such a text input, which includes design constraints, such as the chip's name and dimensions, the device library containing information such as the storage device's name, length, and height, as well as the details of the sequencing graph. The information provided in Figure 4.7 indicates the following:

1) the design is carried out on a chip named chip1, with the chip layout restricted to a 7x7 unit grid;

2) The storage device is named storage, with a width of 2 units and a height of 1 unit;

3) a device named d1 functions as a mixer, with a width of 3 units and a height of 1 unit;
a device named d2 functions as a mixer, with a width of 4 units and a height of 2 units;
a device named d3 functions as a heater, with a width of 1 unit and a height of 2 units;

a device named d4 functions as a detector, with a width of 1 unit and a height of 1 unit;
Devices named dio1 and dio2 serve as input/output ports, each with a width of 1 unit and a height of 1 unit;

4) o1 represents a mixing operation with an execution time of 2 seconds and can be bound to two devices, d1 and d2;
o2 represents a heating operation with an execution time of 3 seconds and can be bound to a single device, d3;
o3 represents a heating operation with an execution time of 1 second and can be bound to a single device, d4;
io1 and io2 represent input/output operations with an execution time of 1 second each, and can be bound to two devices, dio1 and dio2, respectively;

5) io1 has no parent operations and one child operation, o1;
o1 has one parent operation, io1, and one child operation, o2;
o2 has one parent operation, o1, and one child operation, o3;
o3 has one parent operation, o2, and one child operation, io2;
io2 has one parent operation, o3, and no child operations

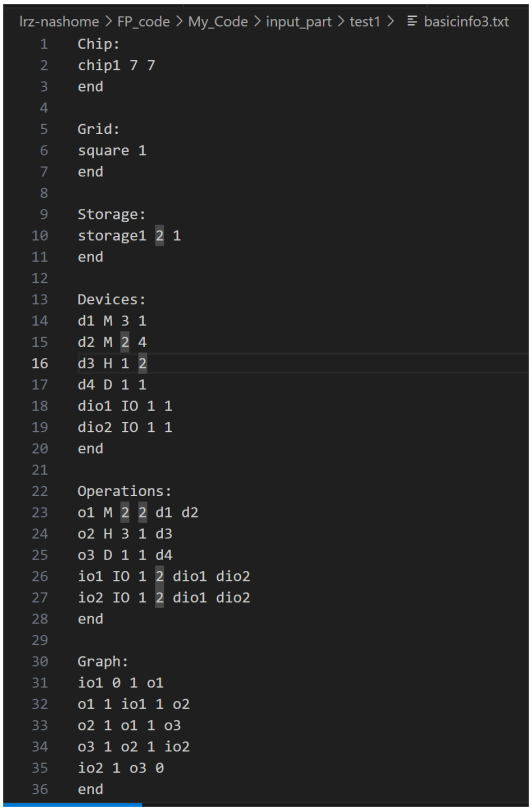


Figure 4.7.: Input format.

In practice, the device library, sequencing graph, and chip area information are supplied to the software using this input format, allowing for the automated design and generation of the chip

architecture.

4.2.3. Software functions implementation

After defining the header files and input format, the next step is to develop the CPP files, which implement the various functionalities of the software. As shown in Figure 4.8, aside from the namespace and macro definitions, the first implementation is the main function, which demonstrates the complete execution process of the software.

```
4  using namespace std;
5
6  #define Mval 1000000
7  #define u 1
8  //#define flowSpeed (generalGrid->gridL)
9  #define flowSpeed 1
10
11 int main(int argc, char* argv[])
12 {
13     cout<<"Start building model instances"<<endl;
14     funcReadFile(argv[1],argv[2]);
15     funcPrepare();
16     cout<<"Start building variables:"<<endl;
17     funcModel_Var();
18     cout<<"End building variables. "<<vvar.size()<<" variables."<<endl;
19     cout<<"Start building constraints:"<<endl;
20     funcModel_Cons();
21     cout<<"End building constraints. "<<nCons<<" constraints."<<endl;
22     funcGurobi(0, 0.05, atoi(argv[3]), 0.0, 0.0, 1, 1); // min/max, heu, time, absgap, gap, dis, focus (quick feas.)
23     //funcScreenMessage_readPart();
24     funcScreenMessage_outputPart();
25     //funcWriteFile(argv[4]);
26 }
```

Figure 4.8.: Main function of the software.

The software first reads the input data from the text file and performs some preprocessing before beginning model construction. Initially, Gurobi variables are created to store the results at each stage of the complete synthesis flow. The mathematical constraints of the model are then established, as the proposed one-pass synthesis flow is formulated as an Integer Linear Programming (ILP) problem, meaning the model consists of a series of mathematical constraint expressions. Once the model is constructed, the software invokes the Gurobi solver to solve the problem, with the final results output both in text format and displayed on the screen.

The following parts provide a detailed explanation of the functionality and implementation of each component function:

- Figure 4.9 illustrates the structure of part of the input-reading function, which reads information from the input and then constructs the corresponding class instances, subsequently used for creating and storing Gurobi variables.

- Figure 4.10 illustrates the structure of part of the pre-processing function, which creates an instance of the grid class for each grid on the chip based on the Cartesian coordinates

```
4021  /* read file function */
4022  void funcReadFile(const char* filename_basicInfo, const char* filename_complement)
4023  {
4024      //read basic info of operation, device, chip, storage and grid
4025      fstream ReadFile_basicInfo;
4026      ReadFile_basicInfo.open(filename_basicInfo, fstream::in);
4027      while(1)
4028      {
4029          string strtmp;
4030          getline(ReadFile_basicInfo,strtmp); //read one line from file
4031          if(ReadFile_basicInfo.eof()) break; //if EOF is readed, break the loop
4032          if(strtmp.compare("Chip:")==0) //chip info
4033          {
4034              while(1)
4035              {
4036                  ReadFile_basicInfo>>strtmp; //read first string of each line
4037                  if(strtmp.compare("end")==0) break; // if the part is end, break the loop
4038                  chipObject = new chip; //creat a chip object
4039                  chipObject->name = strtmp;
4040                  int tempValue0;
4041                  ReadFile_basicInfo>>tempValue0;
4042                  chipObject->chipW = tempValue0; //read chip width and assignment
4043                  ReadFile_basicInfo>>tempValue0;
4044                  chipObject->chipH = tempValue0; //read chip height and assignment
4045              }
4046          }
4047          if(strtmp.compare("Grid:")==0) //grid info
4048          {
```

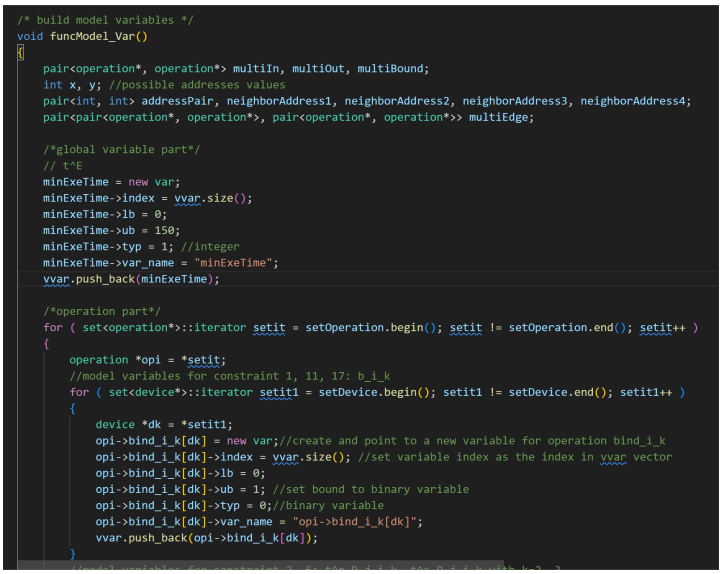
Figure 4.9.: Input-reading function of the software.

of the upper-right corner. Additionally, it constructs a set of neighboring grids for each grid to facilitate subsequent processing.

```
/* create objects that not created by read and modify */
void funcPrepare()
{
    // create grid instances
    int x, y; //possible addresses values
    pair<int, int> addressPair, neighborAddress1, neighborAddress2, neighborAddress3, neighborAddress4;
    for ( x = 0; x <= (chipObject->chipW)/(generalGrid->gridL) + 1; x++ )
    {
        for ( y = 0; y <= (chipObject->chipH)/(generalGrid->gridL) + 1; y++ ) //create grids larger than chip
        {
            addressPair = make_pair(x, y);
            gridAddress.insert(addressPair);
            grid *cell = new grid;
            cell->gridType = generalGrid->gridType;
            cell->gridL = generalGrid->gridL;
            cell->address = addressPair;
            gridMap[addressPair] = cell;
            // if( ((x != (chipObject->chipW)/(generalGrid->gridL) + 1) && x != 0) &&
            //  ((y != (chipObject->chipH)/(generalGrid->gridL) + 1) && y != 0) )
            // {
            //  setGrid.insert(cell); //grids in chip are in setGrid
            // }
        }
    }
    //set neighborGrid
    for ( x = 1; x <= (chipObject->chipW)/(generalGrid->gridL); x++ )
    {
        for ( y = 1; y <= (chipObject->chipH)/(generalGrid->gridL); y++ ) //create grids larger than chip
        {
            addressPair = make_pair(x, y);
            neighborAddress1 = make_pair(x - 1, y);
            neighborAddress2 = make_pair(x, y - 1);
            neighborAddress3 = make_pair(x, y + 1);
            neighborAddress4 = make_pair(x + 1, y);
            setGrid.insert(gridMap[addressPair]); //grids in chip are in setGrid
```

Figure 4.10.: Prepare function of the software.

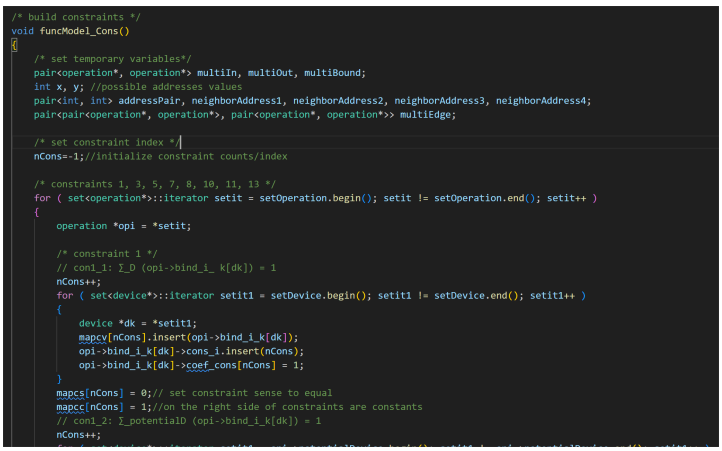
- Figure 4.11 illustrates the structure of part of the model-variable construction function. This function creates Gurobi variables associated with the design for each class instance, assigning parameters such as index, type, and boundaries to facilitate solving by the Gurobi Solver. For example, minTimeExe represents the total execution time of the bioassay and is part of the chip architecture's optimization objective, where a smaller value is preferred. In this function, the software creates a global Gurobi variable for minTimeExe, with a boundary of 0 to 150 and an integer type.



```
/* build model variables */
void funcModel_Var()
{
    pair<operation*, operation*> multiIn, multiOut, multiBound;
    int x, y; //possible addresses values
    pair<int, int> addressPair, neighborAddress1, neighborAddress2, neighborAddress3, neighborAddress4;
    pair<pair<operation*, operation*>, pair<operation*, operation*>> multiEdge;

    /*global variable part*/
    // t^E
    minExeTime = new var;
    minExeTime->index = vvar.size();
    minExeTime->lb = 0;
    minExeTime->ub = 150;
    minExeTime->typ = 1; //integer
    minExeTime->var_name = "minExeTime";
    vvar.push_back(minExeTime);

    /*operation part*/
    for ( set<operation*>::iterator setit = setOperation.begin(); setit != setOperation.end(); setit++ )
    {
        operation *opi = *setit;
        //model variables for constraint 1, 11, 17: b_i_k
        for ( set<device*>::iterator setit1 = setDevice.begin(); setit1 != setDevice.end(); setit1++ )
        {
            device *dk = *setit1;
            opi->bind_i_k[dk] = new var;//create and point to a new variable for operation bind_i_k
            opi->bind_i_k[dk]->index = vvar.size(); //set variable index as the index in vvar vector
            opi->bind_i_k[dk]->lb = 0;
            opi->bind_i_k[dk]->ub = 1; //set bound to binary variable
            opi->bind_i_k[dk]->typ = 0;//binary variable
            opi->bind_i_k[dk]->var_name = "opi->bind_i_k[dk]";
            vvar.push_back(opi->bind_i_k[dk]);
        }
```

Figure 4.11.: Variable construction function of the software.

- Figure 4.12 illustrates the structure of part of the model-constraints construction function, which builds each mathematical constraint for BigIntegr. As shown, nCons represents the index of the current constraint for the Gurobi solver. After transforming the mathematical expression, variables are placed on the left side of the inequality, with constants on the right. Finally, the constraint sense is set, where mapcs[nCons] is assigned a value of 1 for greater_equal, 0 for equal, and -1 for less_equal.

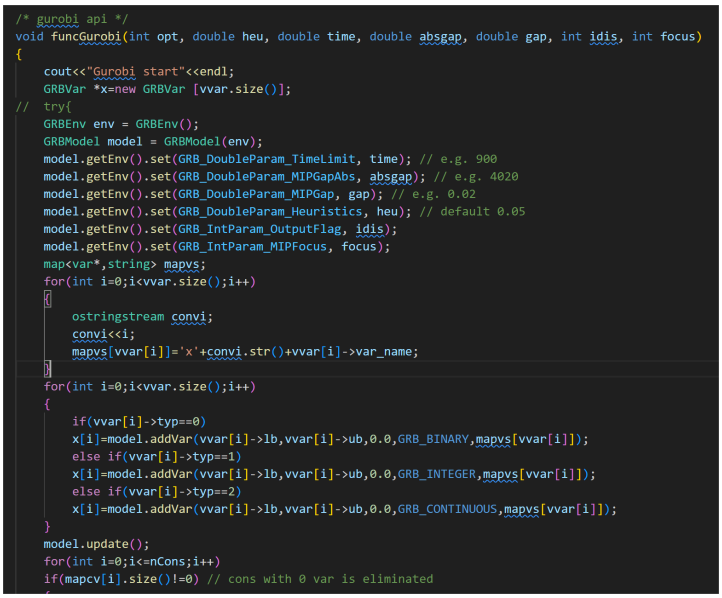


```
/* build constraints */
void funcModel_Cons()
{
    /* set temporary variables*/
    pair<operation*, operation*> multiIn, multiOut, multiBound;
    int x, y; //possible addresses values
    pair<int, int> addressPair, neighborAddress1, neighborAddress2, neighborAddress3, neighborAddress4;
    pair<pair<operation*, operation*>, pair<operation*, operation*>> multiEdge;

    /* set constraint index */
    nCons=-1;//initialize constraint counts/index

    /* constraints 1, 3, 5, 7, 8, 10, 11, 13 */
    for ( set<operation*>::iterator setit = setOperation.begin(); setit != setOperation.end(); setit++ )
    {
        operation *opi = *setit;

        /* constraint 1 */
        // con1_1: Σ_D (opi->bind_i_ k[dk]) = 1
        nCons++;
        for ( set<device*>::iterator setit1 = setDevice.begin(); setit1 != setDevice.end(); setit1++ )
        {
            device *dk = *setit1;
            mapcv[nCons].insert(opi->bind_i_k[dk]);
            opi->bind_i_k[dk]->cons_i.insert(nCons);
            opi->bind_i_k[dk]->coef_cons[nCons] = 1;
        }
        mapcs[nCons] = 0;// set constraint sense to equal
        mapcc[nCons] = 1;//on the right side of constraints are constants
        // con1_2: Σ_potentialD (opi->bind_i_k[dk]) = 1
        nCons++;
```

Figure 4.12.: Constraints construction function of the software.

- Figure 4.13 illustrates the structure of part of the Gurobi invocation function. This function submits BigIntegr, constructed by the model-constraints construction function, to the Gurobi solver for computation. The results at each stage of the complete synthesis flow are then stored in the result members of the respective Gurobi variables, which were

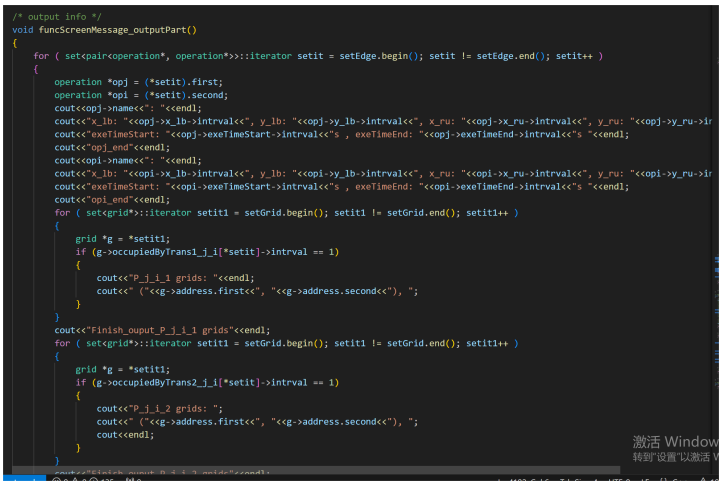
created by the model-variable construction function.



```
/* gurobi api */
void funcGurobi(int opt, double heu, double time, double absgap, double gap, int idis, int focus)
{
    cout<<"Gurobi start"<<endl;
    GRBVar *x=new GRBVar [vvar.size()];
//  try{
    GRBEnv env = GRBEnv();
    GRBModel model = GRBModel(env);
    model.getEnv().set(GRB_DoubleParam_TimeLimit, time); // e.g. 900
    model.getEnv().set(GRB_DoubleParam_MIPGapAbs, absgap); // e.g. 4020
    model.getEnv().set(GRB_DoubleParam_MIPGap, gap); // e.g. 0.02
    model.getEnv().set(GRB_DoubleParam_Heuristics, heu); // default 0.05
    model.getEnv().set(GRB_IntParam_OutputFlag, idis);
    model.getEnv().set(GRB_IntParam_MIPFocus, focus);
    map<var*,string> mapvs;
    for(int i=0;i<vvar.size();i++)
    {
        ostringstream convi;
        convi<<i;
        mapvs[vvar[i]]='x'+convi.str()+vvar[i]->var_name;
    }
    for(int i=0;i<vvar.size();i++)
    {
        if(vvar[i]->typ==0)
        x[i]=model.addVar(vvar[i]->lb,vvar[i]->ub,0.0,GRB_BINARY,mapvs[vvar[i]]);
        else if(vvar[i]->typ==1)
        x[i]=model.addVar(vvar[i]->lb,vvar[i]->ub,0.0,GRB_INTEGER,mapvs[vvar[i]]);
        else if(vvar[i]->typ==2)
        x[i]=model.addVar(vvar[i]->lb,vvar[i]->ub,0.0,GRB_CONTINUOUS,mapvs[vvar[i]]);
    }
    model.update();
    for(int i=0;i<=nCons;i++)
    if(mapcv[i].size()!=0) // cons with 0 var is eliminated
```

Figure 4.13.: Gurobi API function.

- Figure 4.14 shows a code snippet of the output-on-screen function. This function outputs the results stored in the members of the Gurobi variables, representing the results at each stage of the complete synthesis flow, to the terminal. Optionally, the output can also be saved in a text file.



```
/* output info */
void funcScreenMessage_outputPart()
{
    for ( set<pair<operation*, operation*>>::iterator setit = setEdge.begin(); setit != setEdge.end(); setit++ )
    {
        operation *opj = (*setit).first;
        operation *opi = (*setit).second;
        cout<<opj->name<<": "<<endl;
        cout<<"x_lb: "<<opj->x_lb->intrval<<", y_lb: "<<opj->y_lb->intrval<<", x_ru: "<<opj->x_ru->intrval<<", y_ru: "<<opj->y_ru->ir
        cout<<"exeTimeStart: "<<opj->exeTimeStart->intrval<<"s , exeTimeEnd: "<<opj->exeTimeEnd->intrval<<"s "<<endl;
        cout<<"opj_end"<<endl;
        cout<<opi->name<<": "<<endl;
        cout<<"x_lb: "<<opi->x_lb->intrval<<", y_lb: "<<opi->y_lb->intrval<<", x_ru: "<<opi->x_ru->intrval<<", y_ru: "<<opi->y_ru->ir
        cout<<"exeTimeStart: "<<opi->exeTimeStart->intrval<<"s , exeTimeEnd: "<<opi->exeTimeEnd->intrval<<"s "<<endl;
        cout<<"opi_end"<<endl;
        for ( set<grid*>::iterator setit1 = setGrid.begin(); setit1 != setGrid.end(); setit1++ )
        {
            grid *g = *setit1;
            if (g->occupiedByTrans1_j_i[*setit]->intrval == 1)
            {
                cout<<"P_j_i_1 grids: "<<endl;
                cout<<" ("<<g->address.first<<", "<<g->address.second<<"), ";
            }
        }
        cout<<"Finish_ouput_P_j_i_1 grids"<<endl;
        for ( set<grid*>::iterator setit1 = setGrid.begin(); setit1 != setGrid.end(); setit1++ )
        {
            grid *g = *setit1;
            if (g->occupiedByTrans2_j_i[*setit]->intrval == 1)
            {
                cout<<"P_j_i_2 grids: ";
                cout<<" ("<<g->address.first<<", "<<g->address.second<<"), ";
                cout<<endl;
            }
        }
```

Figure 4.14.: Output function.

The six functions described above constitute the complete functionality of the design automation software. With appropriate and well-structured input, the software can generate the detailed chip architecture, which is then output in text format.

4.2.4. Design of output formats and test case

The complexity of the test case is closely related to the design time. To reduce computation time, a simplified test case, as shown in Figure 4.7, was used as mentioned earlier. The results obtained from the calculation are presented in Figure 4.15.

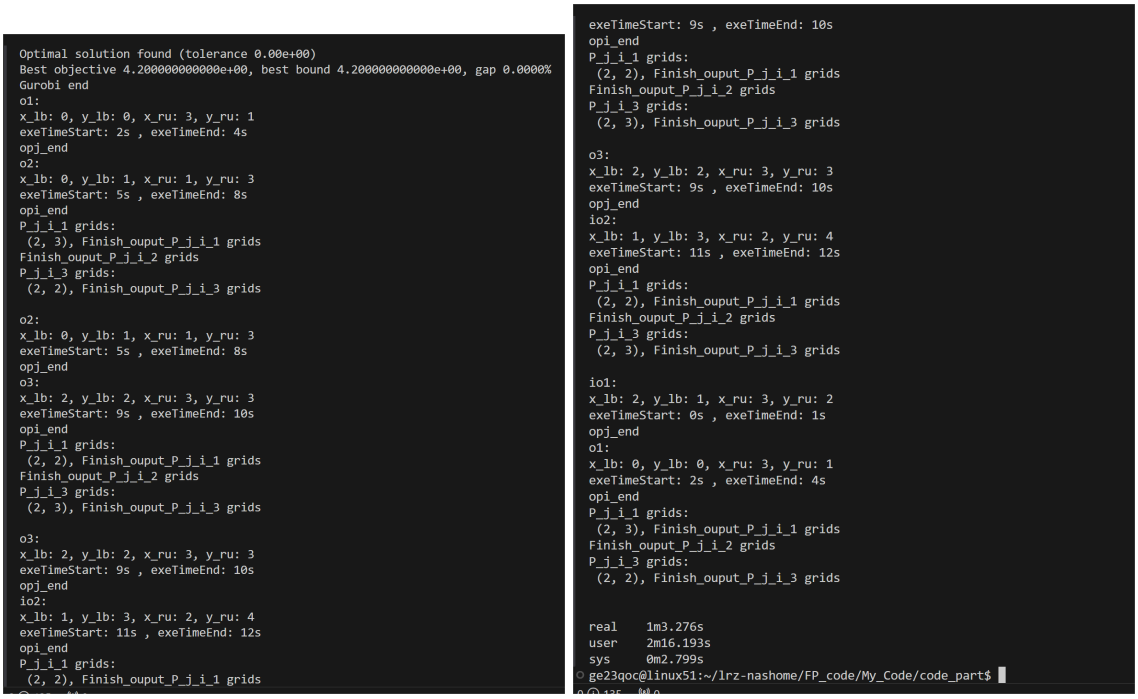


Figure 4.15.: Output format.

The chip architecture can be drawn based on the detailed design results, which are presented in section 5.

5. Experimental results

In this section, the study presents the results of the chip architecture generated by the improved mathematical model of BigIntegr for different input cases.

5.1. test case 1

In this example, the given assay begins with 1 fluid sample as input, which is processed through 3 operations (i.e., o_1 - o_3), and ultimately outputs the fluid at the designated output device. A device library consisting of 6 devices with varying dimensions is provided to execute the operations in the assay. Additionally, the chip layout is constrained to a 7-unit by 7-unit region, and the fluid flow velocity is set at 1 unit per second. The grid length is defined as 1 unit.

The input required for constructing the chip architecture includes several key components, such as the device library, which specifies the available devices and their respective dimensions; the sequential graph, which defines the sequence of operations that must be performed; and the design constraints, which impose limits on the chip layout, such as size and resource usage. These elements are crucial for accurately generating the chip architecture, and their details are illustrated as follows.

- The device library of test case 1 is shown in table 5.1.

available device	dimension($unit^2$)
d1: mixer1	3×1
d2: mixer2	2×4
d3: heater	1×2
d4: detector	1×1
dio1: input/output port	1×1
dio2: input/output port	1×1

Table 5.1.: Device library of test case 1

- The design constraints of test case 1 is shown in table 5.2.

Design constraints	Information
chip available area	7×7 $(unit^2)$
grid area	1×1 $(unit^2)$
storage area	2×1 $(unit^2)$
fluid flow velocity	1 unit per second

Table 5.2.: Design constraints of test case 1

- The sequential graph of test case 1 is shown in Figure 5.1.

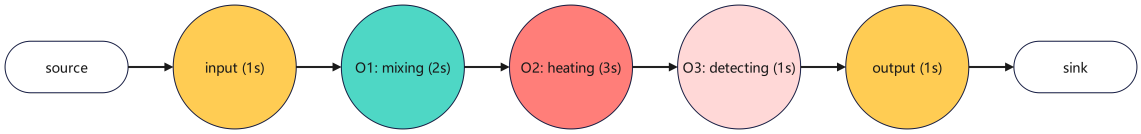


Figure 5.1.: Sequential graph of test case 1.

- Figure 5.2 illustrates the input format for this test case.

lrz-nashome > FP_code > My_Code > input_part > test1 > basicinfo3.txt

```
1   Chip:
2   chip1 7 7
3   end
4
5   Grid:
6   square 1
7   end
8
9   Storage:
10  storage1 2 1
11  end
12
13  Devices:
14  d1 M 3 1
15  d2 M 2 4
16  d3 H 1 2
17  d4 D 1 1
18  dio1 IO 1 1
19  dio2 IO 1 1
20  end
21
22  Operations:
23  o1 M 2 2 d1 d2
24  o2 H 3 1 d3
25  o3 D 1 1 d4
26  io1 IO 1 2 dio1 dio2
27  io2 IO 1 2 dio1 dio2
28  end
29
30  Graph:
31  io1 0 1 o1
32  o1 1 io1 1 o2
33  o2 1 o1 1 o3
34  o3 1 o2 1 io2
35  io2 1 o3 0
36  end
```

Figure 5.2.: Test case 1 input.

After 66.77 seconds of computation, the software produced the following optimal one-pass synthesis results:

- The results of the device binding task are shown in Table 5.3.

operation	device
input(input/output)	dio1
o1(mix)	d1
o2(heat)	d3
o3(detect)	d4
output(input/output)	dio2

Table 5.3.: Device binding result of test case 1.

- The results of the operation scheduling synthesis task are shown in Figure 5.3.

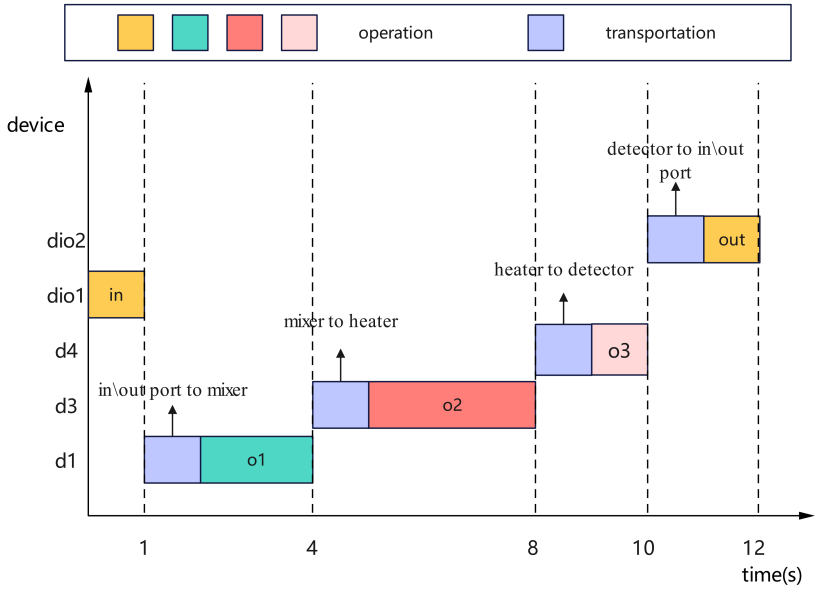


Figure 5.3.: Operation scheduling result of test case 1.

- The results of placement & routing synthesis task are shown in Figure 5.4.

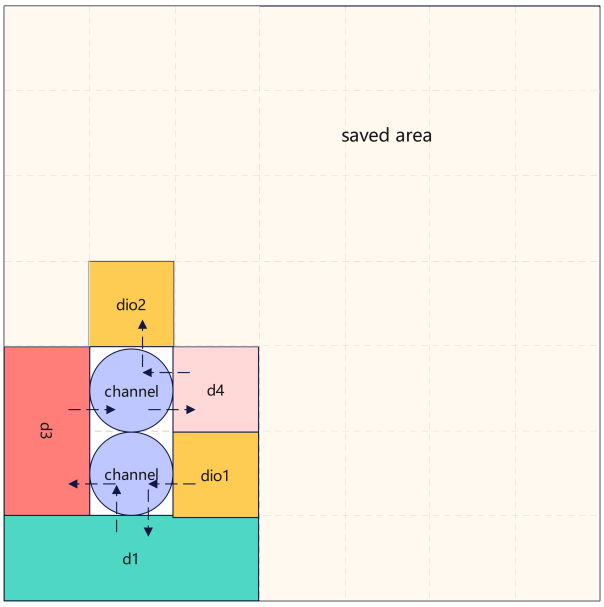


Figure 5.4.: Placement & routing result of test case 1.

5.2. test case 2

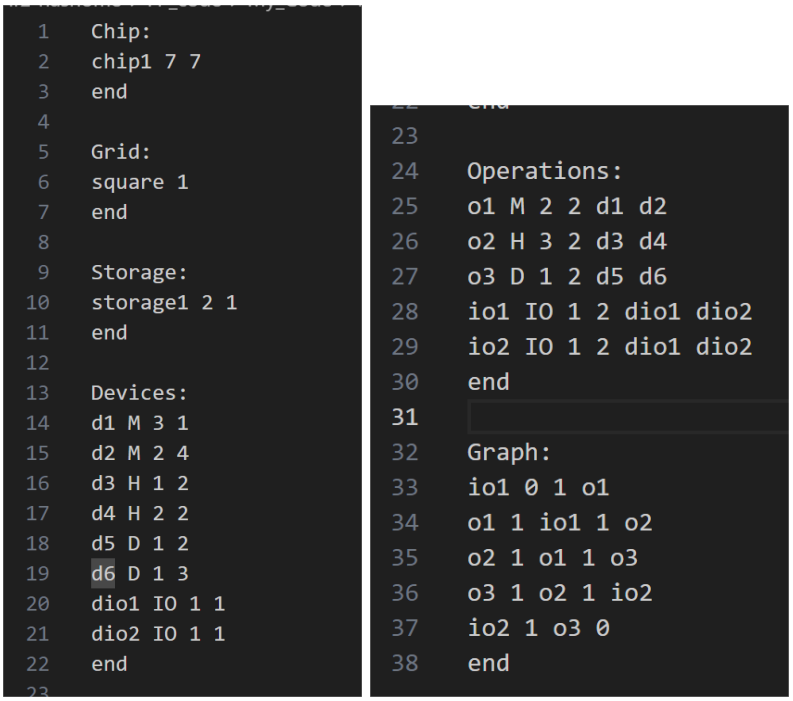
In this example, the device library is shown in Table 5.4. Figure 5.5 illustrates the input format

available device	dimension($unit^2$)
d1: mixer1	3 × 1
d2: mixer2	2 × 4
d3: heater1	1 × 2
d4: heater2	2 × 2
d5: detector1	1 × 2
d6: detector2	1 × 3
dio1: input/output port	1 × 1
dio2: input/output port	1 × 1

Table 5.4.: Device library of test case 2

for this test case.

After 98.30 seconds of computation, the software produced the following optimal one-pass synthesis results:



```
Chip:
chip1 7 7
end

Grid:
square 1
end

Storage:
storage1 2 1
end

Devices:
d1 M 3 1
d2 M 2 4
d3 H 1 2
d4 H 2 2
d5 D 1 2
d6 D 1 3
dio1 IO 1 1
dio2 IO 1 1
end
```

```

Operations:
o1 M 2 2 d1 d2
o2 H 3 2 d3 d4
o3 D 1 2 d5 d6
io1 IO 1 2 dio1 dio2
io2 IO 1 2 dio1 dio2
end

Graph:
io1 0 1 o1
o1 1 io1 1 o2
o2 1 o1 1 o3
o3 1 o2 1 io2
io2 1 o3 0
end
```

Figure 5.5.: Test case 2 input.

- The results of the device binding task are shown in Table 5.5.

operation	device
input(input/output)	dio1
o1(mix)	d2
o2(heat)	d3
o3(detect)	d5
output(input/output)	dio1

Table 5.5.: Device binding result of test case 2.

- The results of the operation scheduling synthesis task are shown in Figure 5.6.
- The results of placement & routing synthesis task for case 2 are shown in Figure 5.7.

The architecture design result information is summarized in Table 5.6.

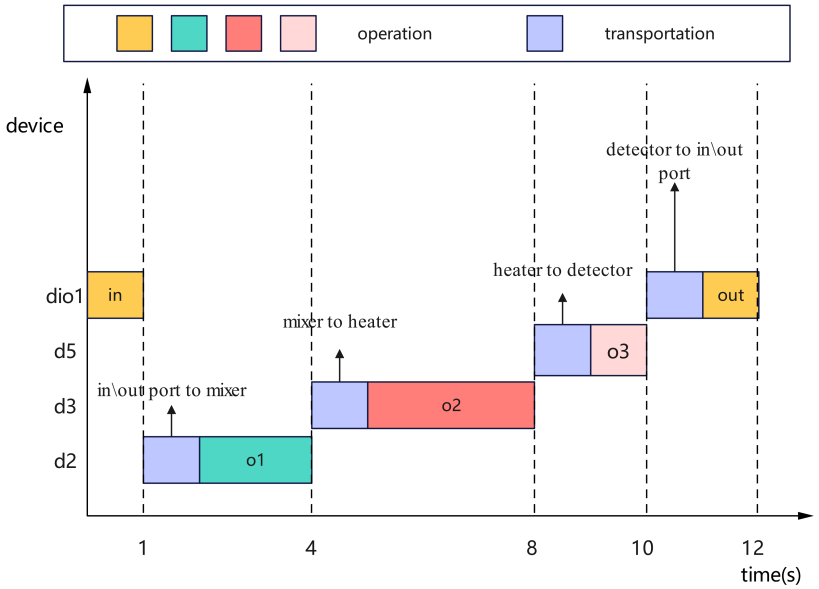


Figure 5.6.: Operation scheduling result of test case 2.

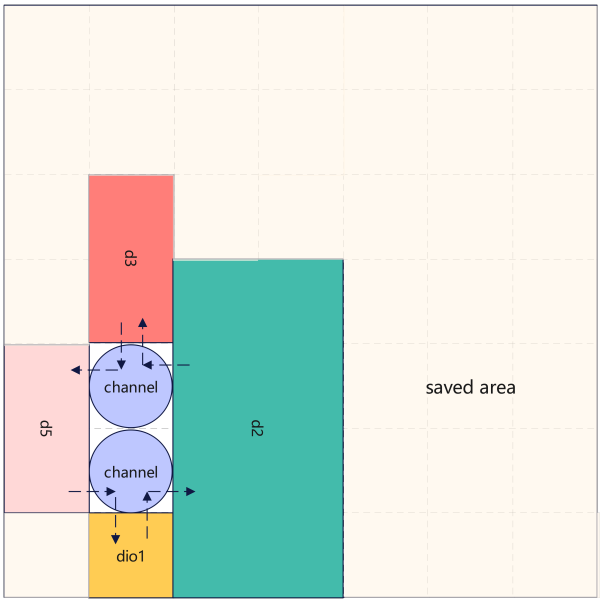


Figure 5.7.: Placement & routing result of test case 2.

Test case	runtime	chip area	execution of bioassays
test case 1	66.77s	10 ($unit^2$)	12s
test case 2	98.30s	15 ($unit^2$)	12s

Table 5.6.: Design result information.

6. Conclusion

6.1. Result conclusion

This study introduced an improved mathematical model of BigIntegr for chip design automation, addressing several shortcomings of the original approach. The enhancements focused on clarifying ambiguous constraints and incorporating corner case considerations, which were previously overlooked. Key improvements were made in areas such as device binding, operation scheduling, and routing, with particular attention given to storage and boundary conditions. By integrating these refinements, the proposed one-pass synthesis flow was able to generate optimized chip architectures for various input cases, as demonstrated through test case results. These enhancements not only improve the accuracy and reliability of automated chip design but also contribute to advancing the development of continuous-flow microfluidic biochip systems.

Through implementation and multiple tests, the effectiveness of the improved mathematical model of BigIntegr proposed in this study has been validated.

6.2. Future work

While the improved mathematical model of BigIntegr has demonstrated its effectiveness in chip design automation, there remain several areas for further exploration. One potential direction is to enhance the scalability of the model, optimizing its performance for larger and more complex bioassay protocols.

Current test results indicate that for highly complex chip architectures, the software's design time is excessively long. Future work will focus on further streamlining the model to enhance its speed, thereby broadening its applicability across a wider range of biochip technologies more complex bioassay protocols.

Bibliography

[1] Xing Huang, Youlin Pan, Zhen Chen, Wenzhong Guo, Robert Wille, Tsung-Yi Ho, and Ulf Schlichtmann. Bigintegr: One-pass architectural synthesis for continuous-flow microfluidic lab-on-a-chip systems. In 2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD), pages 1–8, 2021.

[2] X. Huang, T.-Y. Ho, W. Guo, B. Li, K. Chakrabarty, and U. Schlichtmann, "Computer-aided design techniques for flow-based microfluidic lab-on-a-chip systems," ACM Comput. Surv., vol. 54, no. 5, pp. 1–29, 2021.

[3] C. D. Chin, T. Laksanasopin, Y. K. Cheung, D. Steinmiller, V. Linder, H. Parsa, J. Wang, H. Moore, R. Rouse, G. Umviligihozo et al., "Microfluidics-based diagnostics of infectious diseases in the developing world," Nature medicine, vol. 17, no. 8, pp. 1015–1019, 2011.

[4] N. Convery and N. Gadegaard, "30 years of microfluidics," Micro and Nano Engineering, vol. 2, pp. 76–91, 2019.

[5] M. A. Unger, H.-P. Chou, T. Thorsen, A. Scherer, and S. R. Quake, "Monolithic microfabricated valves and pumps by multilayer soft lithography," Science, vol. 288, no. 5463, pp. 113–116, 2000.

[6] Y. Zhu, X. Huang, B. Li, T.-Y. Ho, Q. Wang, H. Yao, R. Wille, and U. Schlichtmann, "Multicontrol: Advanced control logic synthesis for flow-based microfluidic biochips," IEEE Trans. Comput.-Aided Design Integr. Circuits Syst., vol. 39, no. 10, pp. 2489–2502, 2020.

[7] T. Thorsen, S. J. Maerkl, and S. R. Quake, "Microfluidic large-scale integration," Science, vol. 298, no. 5593, pp. 580–584, 2002.

[8] I. E. Araci and S. R. Quake, "Microfluidic very large scale integration (mvlsi) with integrated micromechanical valves," Lab on a Chip, vol. 12, no. 16, pp. 2803–2806, 2012.

[9] K.-H. Tseng, S.-C. You, J.-Y. Liou, and T.-Y. Ho, "A top-down synthesis methodology for flow-based microfluidic biochips considering valve-switching minimization," in Proc. Int. Symp. Phys. Des., 2013, pp. 123–129.

[10] Gurobi Optimizer Reference Manual, Gurobi Optim., Inc., Beaverton, OR, USA, 2013. [Online]. Available: http://www.gurobi.com

[11] I. Griva, S. G. Nash, and A. Sofer, Linear and Nonlinear Optimization 2nd Edition. Philadelphia, PA: Society for Industrial, 2008. doi: 10.1137/1.9780898717730.