Compilation and Synthesis for Fault-Tolerant Digital Microfluidic Biochips

Alistar, Mirela

Publication date:
2014

Document Version
Publisher's PDF, also known as Version of record

Compilation and Synthesis for Fault-Tolerant Digital Microfluidic Biochips

Mirela Alistar

Kongens Lyngby 2014
IMM-PhD-2014-332

Technical University of Denmark
Department of Applied Mathematics and Computer Science
Matematiktorvet, building 303B,
2800 Kongens Lyngby, Denmark
Phone +45 4525 3351
compute@compute.dtu.dk
www.compute.dtu.dk IMM-PhD-2014-332

Summary

Microfluidic-based biochips are replacing the conventional biochemical analyzers, by integrating all the necessary functions for biochemical analysis using microfluidics. The digital microfluidic biochips (DMBs) manipulate discrete amounts of fluids of nanoliter volume, named droplets, on an array of electrodes to perform operations such as dispensing, transport, mixing, split, dilution and detection.

Researchers have proposed compilation approaches, which, starting from a biochemical application and a biochip architecture, determine the allocation, resource binding, scheduling, placement and routing of the operations in the application. During the execution of a bioassay, operations could experience transient faults, thus impacting negatively the correctness of the application. We have proposed both offline (design time) and online (runtime) recovery strategies. The online recovery strategy decides the introduction of the redundancy required for fault-tolerance. We consider both time redundancy, i.e., re-executing erroneous operations, and space redundancy, i.e., creating redundant droplets for fault-tolerance. Error recovery is performed such that the number of transient faults tolerated is maximized and the timing constraints of the biochemical application are satisfied.

Previous work has assumed that the biochip architecture is given, and most approaches consider a rectangular shape for the electrode array, where operations execute on rectangular "modules" formed of electrodes. However, non-regular application-specific architectures are common in practice. Hence, we have proposed an approach to the synthesis of application-specific architectures, such that the cost is minimized and the timing constraints of the application are satisfied.

We propose an algorithm to build a library of non-regular modules for a given application-specific architecture, so that the area of a non-regular application-specific biochip can be used effectively. During fabrication, DMBs can be affected by permanent faults, which may lead to the failure of the application. Our approach introduces redundant electrodes to synthesize fault-tolerant architectures aiming at increasing the yield of DMBs. We also propose a method to estimate, at design time, the application completion time in case of permanent faults in order to verify if an application can be successfully run on the architecture.

The proposed approaches were evaluated using several real-life case studies and synthetic benchmarks.

Resumé

Mikrofluidiske biochips erstatter i stigende grad konventionelle biokemiske analyser ved at integrere alle nødvendige operationer i den biokemiske analyse på en enkelt biochip. Digitale Mikrofluidiske Biochips (DMBs) manipulerer små diskrete mængder væske i størrelsesordnen nanoliter, kaldet dråber (eng., droplets), på et array af elektroder, for at udføre operationer så som: dosering, transport, blanding, opsplitning, fortynding og detektering.

Forskere har foreslået en samlet design proces, der startende fra en biokemisk applikation (bioassay) og en biochip arkitektur, fastsætter allokering, resurse binding, afviklings rækkefølge, placering og transport, af de i applikationen anvendte operationer. Under eksekveringen af bioassayet kan operationerne blive udsat for transiente fejl, hvilket kan påvirke nøjagtigheden af operationer og i værste fald betyde at applikationen leverer et forkert resultat. Vi har udviklet både offline (dvs. under design processen) og online (dvs. under eksekvering) fejlkorrektionsmetoder. Online fejlkorrigering beslutter den nødvendige redundans, der sikre tilstrækkelig fejltolerance. Vi håndterer både tids-redundans, dvs., genberegning af fejlende operationer, og fysisk-redundans, dvs., dublering af dråber. Fejlrettelse udføres således, at antallet af transiente fejl, der kan accepteres, maksimeres, samtidig med at tidskrav for den biokemiske applikation overholdes.

Tidligere arbejder har antaget, at biochip arkitekturen er givet på forhånd, og de fleste tilgange antager en rektangulær form af elektrodearrayet, hvor operationer afvikles inden for et rektangulært set af elektroder, kaldet et “modul” . Dog forekommer irregulære applikationsspecifikke arkitekturer ofte i praksis. Derfor har vi foreslået en udviklingsmetode til applikationsspecifike arkitekturer, der sikre at prisen minimeres og tidskravene for applikationen overholdes.

Vi foreslår en algoritme til at opbygge et bibliotek af ikke-rektangulære moduler til en given applikationsspecifik arkitektur, således, at arealet af irregulære applikationsspecifikke biochip kan udnyttes effektivt. Under fabrikation kan DMBs være udsat for permanente fejl, hvilket kan lede til at bioassayet fejler. Vi foreslår også en metode der, i tilfælde af permanente fejl, kan estimere tiden for færdiggørelsen af applikationen, under selve designprocessen. Vi kan således verificere, at applikationen kan afvikles på den fejlbehæftede arkitektur.

Den udviklingsmetode er blevet evalueret på adskillige virkelige case-studier og syntetiske benchmarks.

Preface

This thesis was prepared at the Department of Applied Mathematics and Computer Science, Technical University of Denmark in fulfillment of the requirements for acquiring the Ph.D. degree in computer engineering.

The thesis deals with optimization methods for the compilation and synthesis of fault-tolerant digital microfluidic biochips.

The work has been supervised by Associate Professor Paul Pop and co-supervised by Professor Jan Madsen.

Lyngby, 31 March 2014

Mirela Alistar

Papers included in the thesis

- Mirela Alistar, Elena Maftei, Paul Pop, and Jan Madsen. Synthesis of biochemical applications on digital microfluidic biochips with operation variability. *Symposium on Design Test Integration and Packaging of MEMS/MOEMS*, pages 350–357, 2010. Published.

- Mirela Alistar, Paul Pop, and Jan Madsen. Online synthesis for error recovery in digital microfluidic biochips with operation variability. *Symposium on Design Test Integration and Packaging of MEMS/MOEMS*, pages 53–58, 2012. Published.

- Mirela Alistar, Paul Pop, and Jan Madsen. Application-specific fault-tolerant architecture synthesis for digital microfluidic biochips. *Asia and South Pacific Design Automation Conference*, pages 794–800, 2013. Published.

- Mirela Alistar, Paul Pop, and Jan Madsen. Operation placement for application-specific digital microfluidic biochips. *Symposium on Design Test Integration and Packaging of MEMS/MOEMS*, pages 1–6, 2013. Published.

- Mirela Alistar, Paul Pop, and Jan Madsen. Biochemical application compilation and architecture synthesis for fault-tolerant digital microfluidic biochips. *Lab-on-a-Chip World Congress*, 2013. Poster presentation.

- Paul Pop, Jan Madsen, Kasper Understrup, Mirela Alistar, and Wajid Hassan Minhass. Programming language and tools for multipurpose lab-on-a-chip platforms. *Lab-on-a-Chip European Congress*, 2014. Poster presentation.

- Mirela Alistar, Paul Pop, and Jan Madsen. Compilation and synthesis for fault-tolerant digital microfluidic biochips. *Design Automation and Test in Europe Conference*, 2014. Poster presentation at PhD Forum. **Best poster award**.

- Mirela Alistar, Paul Pop, and Jan Madsen. Redundancy optimization for error recovery in digital microfluidic biochips. Under review in *Journal of Design Automation for Embedded Systems*, 2014.

Acknowledgements

I thank Paul Pop, my super supervisor, for his dedication and constant effort to make my PhD years both lucrative and a valuable life experience. I am extremely grateful for his guidance, 24/7 availability and unlimited patience. I also thank Jan Madsen, my co-supervisor, for helping my research with insightful feedback and creative ideas.

I thank my colleagues for having someone to share late working nights at university and great laughter during lunch breaks. My gratitude goes also to my family and friends that had to endure endless technical monologues. I thank them for supporting my enthusiasm, for encouraging me in troubled times and for being proud of me.

Lastly, I dedicate this thesis to Abi. *Little one, always learn to be smarter than life!*

Contents

List of Figures

List of Tables

Abbreviations

Abbreviation	Meaning
BFS	Breadth-First Search
CCD	Charged-Couple Device
CRM	Circular-Route Module
CPA	Colorimetric Protein Assay
DAG	Directed Acyclic Graph
DMB	Digital Microfluidic Biochip
EWOD	Electrowetting on Dielectric
FA-LSR	Fault-Aware List Scheduling and Routing
FTC	Fault-Tolerant Compilation
FTSG	Fault-Tolerant Sequencing Graph
GFTA	Generalized Fault-Tolerant Application
IDP	Interpolation Dilution of a Protein
ILB	Incremental Library Build
IVD	In-Vitro Diagnostics
LED	Light-Emitting Diode
LS	List Scheduling
LSPR	List Scheduling, Placement and Routing
OIA	Operation-Interdependency-Aware compilation
PCR	Polymerase Chain Reaction
PDNA	sample preparation for Plasmid DNA
ROS	Redundancy Optimization Strategy
RRT	Rectangular Route of varying Thickness
SA	Simulated Annealing
SB	Synthetic Benchmark
SFS	Straighforward Scheduling
TS	Tabu Search

Notations

Notation	Meaning
q	number of transient faults in all types of operations
s	number of transient faults in split operations
k	number of permanent faults
$\mathcal{G}^0$, $\mathcal{G}$	biochemical application graph without fault-tolerance
$\mathcal{G}^+$	biochemical application graph $\mathcal{G}^0$ with the detection operations
$\mathcal{G}^R$	biochemical application graph with redundancy
$\delta_{\mathcal{G}}$	application completion time without fault-tolerance
$\delta^s_{\mathcal{G}}$	application completion time in case of s faults in split operations
$\delta^k_{\mathcal{G}}$	application completion time in case of k permanent faults
$p^1_{0^\circ}$	percentage towards operation completion for a forward movement of one electrode
$p^2_{0^\circ}$	percentage towards operation completion for a forward movement of at least two consecutive electrodes
p_{180°	percentage towards operation completion for a backward movement
p_{90°	percentage towards operation completion for a 90° turn
p^0_{cycle}	percentage towards operation completion for a cycle when there are no faults
p^f_{cycle}	percentage towards operation completion for a cycle in case of f permanent faults

Notation	Meaning
C_i	operation execution time without fault-tolerance
C_i^k	operation execution time in case of k permanent faults
O_i	operation
$O_{i,x}$	the x^{th} copy of the operation O_i
O_i^R	redundant operation
D_i	detection operation
D_i^R	detection operation needed to detect an error during R_i
E_i	condition of a fault occurrence
$\overline{E}_i$	condition of a no fault occurrence
E_{Mix}	intrinsic error limit for mixing operation
E_{Ds}	intrinsic error limit for dispensing operation
E_{Slt}	intrinsic error limit for split operation
E_{Trans}	intrinsic error limit for transport operation
E_{Dlt}	intrinsic error limit for dilution operation
I_i	input operation error limit
E_{Thr}	error threshold
R_i	recovery subgraph for O_i
R_i^{Space}	recovery subgraph for space redundancy
R_i^{Time}	recovery subgraph for time redundancy
M_i	circular-route module
$\mathcal{L}$	module library
$\mathcal{A}$	biochip architecture
$\mathcal{F}$	fluidic library
$\mathcal{M}$	component library
$\mathcal{O}$	allocation
$\mathcal{B}$	binding
$\mathcal{P}$	placement
$\mathcal{U}$	routing
$\mathcal{S}$	schedule of operations
$In\ R$	dispensing reservoir for reagent
$In\ S$	dispensing reservoir for sample
$In\ B$	dispensing reservoir for buffer
n_{sns}	number of sensors
RF_i	reusability factor for operation O_i
$\mathcal{R}_i$	chain of electrodes
$\mathcal{E}$	set of electrodes
e_i	electrode
TI	initial temperature
TL	temperature length
ε	cooling ratio
W	penalty weight

CHAPTER 1

Introduction

Microfluidics, the study and handling of small volumes of fluids, is a well-established field, with over 10,000 papers published every year [56]. With the introduction at the beginning of 1990s of microfluidic components such as microvalves and micropumps, it was possible to realize "micro total analysis systems" (μTAS), also called "lab-on-a-chip" and "biochips", for the automation, miniaturization and integration of complex biochemical protocols [55].

The trend today is towards *microfluidic platforms*, which according to [55], provide "a set of fluidic unit operations, which are designed for easy combination within a well-defined fabrication technology", and offer a "generic and consistent way for miniaturization, integration, customization and parallelization of (bio-)chemical processes". Microfluidic platforms are used in many application areas, such as, *in vitro* diagnostics (point-of-care, self-testing), drug discovery (high-throughput screening, hit characterization), biotech (process monitoring, process development), ecology (agriculture, environment, homeland security) [14, 72].

Microfluidic platforms can be classified according to the liquid propulsion principle used for operation, e.g., capillary, pressure driven, centrifugal, electrokinetic or acoustic. In this thesis, we are interested in microfluidic platforms which manipulate the liquids as droplets, using electrokinetics, i.e., electrowetting-on-dielectric (EWOD) [60]. We call such platforms *digital microfluidic biochips* (DMBs).

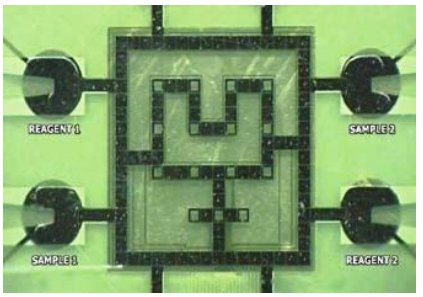


Figure 1.1: Digital microfluidic biochip example [82]

Figure 1.2: Example controller platform [7]

DMBs integrate on-chip all the functions needed to complete biochemical applications such as bioassays, which measure the concentration of a specific constituent in a mixture (e.g., measuring the concentration of glucose in plasma, serum, urine and saliva [72]).

DMBs are typically modeled as an array of electrodes, where each electrode can hold a droplet, see Fig. 1.1 and 1.3 for example DMBs. Hence, DMBs are able to perform fluidic operations such as dispensing, transport, mixing, split, dilution and detection using droplets (discrete amount of fluid of nanoliters volume) [19, 22].

Immediate advantages of DMBs are *automation*—reducing the likelihood of human error, and *integration*—eliminating additional equipment for intermediate steps. Moreover, due to *miniaturization*, the reagent and sample consumption is lower and the bioassay time-to-result is shortened. By using smaller volumes of expensive reagents and hard-to-obtain samples, the costs are significantly reduced thus addressing an important concern of clinical laboratories. Moreover, faster reaction times are observed when using volumes at the microliter scale, making DMBs suitable for flash chemistry applications [89, 44]. The reduced size of DMBs contribute to their portability, making digital microfluidic platforms ideal candidates for near-patient and point-of-care testing [68].

For example, digital microfluidic platforms have been proposed for newborn screening, a procedure that tests newborns for genetical diseases that can result in irreversible organ damage if not treated soon after birth. In order to screen for Pompe and Fabry diseases, a DMB needs a fraction of sample and reagents volumes required by standard methods [69]. The incubation time was reduced to less than 2 hours, resulting in a 10 times faster time-to-result than when using standard methods. Recently, Advanced Liquid Logic, Inc., proposed an advanced DMB that is able to screen 40 newborns at the same time [7].

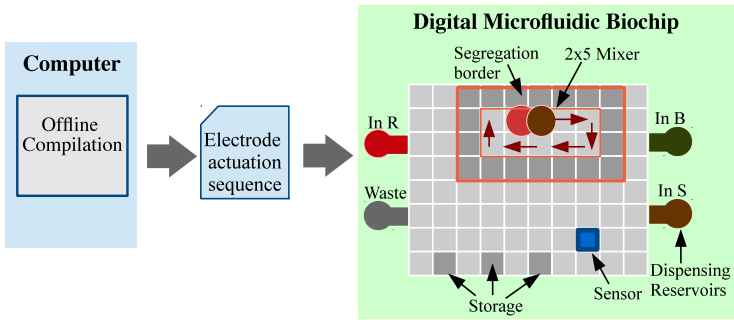


Figure 1.3: Setup for running an application on a DMB

In order to run a biochemical application, a DMB is typically used with a controller platform that has additional components such as sensing systems, charged-couple device camera-based detectors, magnetic bars, heaters, etc. [68]. The DMB is loaded with the input fluids (samples, reagents, buffers), then placed in the controller platform, which is connected to the computer.

The flow of a biochemical application execution is schematically represented in Fig. 1.3. A control software that compiles the biochemical application is executed on the computer. The output of the control software is the "electrode actuation sequence", which controls the movement of droplets in order to run the biochemical application. The controller platform actuates the droplets and activates the additional components according to the electrode actuation sequence.

Fig. 1.2 shows the controller platform LSD100 developed by Advanced Liquid Logic, Inc., for Lysosomal Storage Enzyme Analysis [7]. Alternatively, the results of the control software can be stored on a flash drive which is plugged in the controller platform, thus increasing the portability of the system.

1.1 Related work

Initially, bottom-up approaches have been used for the design of DMBs. However, due to the increase in the complexity of DMBs, top-down methodologies are more suitable, as they can easily scale for new designs [14]. There is a lot of research on methods for the design of biochips, see [13, 9] for surveys about this area. This section is not intended to give an overview of the research in this field. Instead, we mention only the research that is closely related to the methods proposed in the thesis.

Researchers have used the term "synthesis" to denote the tasks that determine the "electrode actuation sequence", which controls the movement of droplets to run the biochemical application. We will call these synthesis tasks *compilation*, to distinguish it from the architecture *synthesis*, which determines the biochip architecture for a specific biochemical application.

Researchers have initially assumed regular rectangular architectures and have focused on the compilation tasks. The compilation process is a NP-complete problem [14], which, consists of the following tasks: allocation, resource binding, scheduling, placement and routing, which are explained in the following. Biochemical applications can be *modeled* using programming languages such as BioCoder [8, 28]. However, most researchers have used sequencing graph models, where each node is an operation, and each edge represents a dependency [14]. Most of previous work has assumed that operations execute on predetermined areas of electrodes, named "modules", which are *allocated* from a given module library. Recently, [46] have proposed the use of additional equipment (e.g., sensing systems) to eliminate the need for characterizing a module library. As soon as the *binding* of operations to the allocated modules is decided, the *scheduling* algorithm determines the time duration for each bioassay operation, subject to resource constraints and precedence constraints imposed by the application. Next, the *placement* [90] of operations on the microfluidic array and the *routing* [15, 38] of droplets from one module to another have to be determined. In Chapter 3 we present examples of compilation.

Three of the compilation tasks are NP-complete problems: scheduling [79], placement [77] and routing [91]. Hence, in order to reduce the complexity of the compilation problem, researchers have initially separated the compilation tasks into "architectural-level" compilation (i.e., modeling, allocation, binding and scheduling) and "physical-level" compilation (i.e., placement and routing) [31].

At architectural-level, compilation approaches were proposed using Integer Linear Programing [73], modified List Scheduling algorithm [73] and metaheuristics such as Genetic Algorithms [73, 63]. At physical-level, [77, 74] proposed placement algorithms based on Simulated Annealing.

Next, researchers have considered unified approaches for the compilation of DMBs, which combine the architectural and physical levels. The integer linear programming formulation proposed in [54] derives optimal solutions for small applications. Near-optimal results in terms of application completion time were obtained by compilation implementations based on search metaheuristics such as Tabu Search [50] and Parallel Recombinative Simulated Annealing [14], which combines genetic algorithms and simulated annealing algorithms. List Scheduling-based compilations were proposed in [4, 45, 29, 27]. These compilations are faster and thus can be used to take decisions at runtime and to quickly evaluate an alternative architecture during architecture synthesis.

In related literature, several placement strategies have been proposed for rectangular architectures. In [77], a Simulated Annealing-based method is used to determine the placement of the operations on the biochip. A unified compilation and module placement, based on Parallel Recombinative Simulated Annealing, was proposed in [75]. Better results were obtained by using a T-Tree algorithm for placement [90] or by using a fast-template placement [10] integrated in a Tabu Search-based compilation [50]. A placement approach that minimizes droplet routing, when deciding the locations of the modules, is considered in [87]. Placement strategies based on virtual topology [27] were proposed for fast compilation approaches.

A lot of work has been done concerning the routing problem. [11] proposed a routing algorithm based on A* search, which finds the optimal-length route, but works only when routing is possible. Hence, [80] proposed redoing the placement when routing fails and used modified Lee algorithm for routing. In [15], a "bypassibility" metric is proposed to assign droplet priorities such that deadlocks are avoided. However, in case of deadlock, "concession zones" are used to store the droplets and thus eliminate the deadlock.

A network-flow based routing algorithm was proposed in [91], which has two stages: (i) global routing, where a set of independent nets is determined and a rough routing path is generated for each droplet, and (ii) detailed routing, which routes each droplet using a negotiation-based algorithm. The routing strategy proposed in [36] determines a global routing track on which droplets prefer to move ordered by an entropy-based metrics. The routes were compacted in [36] using dynamic programming.

Cross-contamination is a frequent problem for the biochemical assays that use proteins [93]. To avoid cross-contamination, wash droplets are transported over the contaminated areas to clean the residues. Researchers have proposed methods to optimize the washing [92, 93] and routing algorithms to minimize the intersection of operations with contamination conflicts [37].

Several mixing and dilution techniques have been proposed for sample preparation [62, 65, 33], focusing on obtaining a desired concentration by using the byproduct droplets of operations already executed and thus reduce the waste. Researchers have also proposed techniques for fabrication [60, 25] and testing of DMBs [88, 81]. However, these topics (i.e., contamination, mixing/dilution algorithms, fabrication and testing) are orthogonal to the research work presented in this thesis.

Fault-tolerance to transient faults. During the execution of the bioassay, the volume of droplets can vary erroneously due to transient faults. The errors propagate throughout the entire application, affecting eventually the result of the bioassay. Biochemical applications have high accuracy requirements, determined by the acceptance range for the concentration of droplets. Example applications with accuracy requirements of less than $\pm 10\%$ are drug discovery applications [64] and plasmid DNA preparation [41].

Past research has addressed the erroneous volume variation due to transient faults using re-execution and checkpointing as recovery techniques [1, 95]. The work in [95] addresses the volume variations in operations, by duplicating intermediate droplets of correct volumes and storing them at checkpoints. When an error is detected, the stored droplets are used in the recovery subroutine. The locations of the checkpoints and the recovery subroutines are determined offline and stored in a microcontroller memory. If an error is detected during runtime at a checkpoint, the microcontroller interrupts the bioassay, and transports the intermediate product droplets to the storage units; then the corresponding recovery subroutine is executed using a statically predetermined allocation and placement, which do not consider the current context. Consequently, the delays introduced by the recovery subroutines can lead to the application missing the deadline.

Hence, in [44] the authors propose a method to precompute and store a dictionary that contains recovery solutions for all combinations of errors. When an error is detected, the system looks in the dictionary for the corresponding recovery actuation sequence. Since the recovery solutions consider the current context, the delays due to recovery are minimized. However, taking into account all possible scenarios for any combinations of errors comes with high storage requirements. Hence, compression algorithms are needed to reduce the size of the dictionary in order to store it on the flash memory of the microcontroller.

In all mentioned approaches, the error recovery actions are determined *offline*, and are applied online when a fault is detected. Researchers have also proposed *online* approaches that determine the necessary recovery actions *during the execution of the biochemical application*, at the moment when an error is detected. Such online recovery approaches, some of which also perform online re-compilation to reconfigure the electrode actuation sequence, are possible because the biochemical application execution times are much slower compared to the control software execution.

The work in [32] addresses sample preparation and proposes dynamic error recovery to recreate online the desired target concentrations, using the stored intermediate droplets. A general approach, that compiles a new implementation containing the appropriate error recovery actions whenever errors are detected, is proposed in [45]. The online compilation re-computes the placement of operations and the droplets routes using a List-Scheduling based implementation.

In this thesis we propose both offline (design time) and online (runtime) recovery strategies. The online recovery strategy decides the introduction of the redundancy required for fault-tolerance. We consider both time redundancy, i.e., re-executing erroneous operations, and space redundancy, i.e., creating redundant droplets for fault-tolerance. Error recovery is performed such that the number of transient faults tolerated is maximized and the timing constraints of the biochemical application are satisfied.

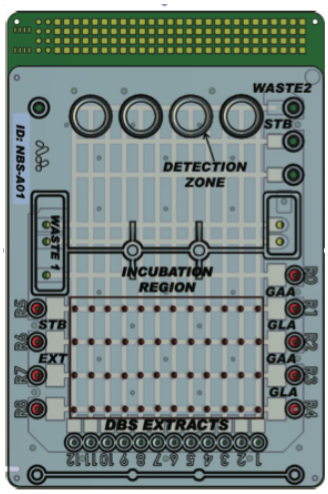


Figure 1.4: DMB for newborn screening [69]

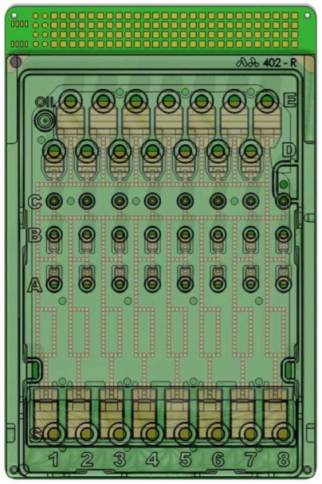


Figure 1.5: DMB for sample preparation [57]

Architecture synthesis. The physical architecture of a DMB consists of *physical* components, such as electrodes, sensors, detectors, heaters, actuators, reservoirs for dispensing and waste. Previous work assumes that the physical architecture of a DMB is given and focuses on the automation of the application execution.

Moreover, most researchers use general-purpose architectures, which have a rectangular shape (Fig. 1.3). However, in practice, application-specific architectures which are non-regular (Fig. 1.1) are more common because they can significantly reduce the costs by including only the components that are necessary for the execution of the application.

Application-specific architectures are designed manually, which is an expensive time-consuming process. Fig. 1.4 and 1.5 show two examples of application-specific biochips designed for newborn screening [69] and sample preparation [57]. Most work done so far, only considered varying the dimensions of purely rectangular general-purpose architectures or addressed aspects such as minimizing the number of pins used to control the electrodes [94]. Researchers have proposed approaches to optimize the biochip architecture for targeted applications, such as the polymerase chain reaction [43] and sample preparation [32].

In the context of application-specific architectures, the placement problem becomes more challenging. All approaches previously mentioned consider placement of rectangular modules, which do not take advantage of the non-regular area of an application-specific biochip.

A placement strategy for modules of non-rectangular shapes is proposed in [50]. However, [50] uses a "black-box" approach, i.e., the whole module area is considered occupied during the execution of the operations, blocking the corresponding electrodes from being used for other operations. An alternative is the routing-based approach from [52], which allows the droplets to move freely on the biochip until the operation is completed. However, in case of contamination, the routing-based strategy requires a lot of washing, which slows considerably the execution of the bioassay and can lead to routing deadlocks.

DMBs can be affected by permanent faults, which may lead to the failure of the biochemical application. In addition, yield is a big concern for biochips—researchers have proposed fabrication methodologies to increase the yield of DMBs, e.g., from a very low 30% to 90% [78]. After fabrication, the biochips are tested and if permanent faults are detected, the biochip is discarded, unless the applications can be reconfigured to avoid them [51]. In order to increase the yield, which is very important for the market success of DMBs, the design of DMBs has to take into account possible defects that can be introduced during the fabrication process. Because of their optimized layout, application-specific architectures are critically affected by permanent faults.

The issue of fault-tolerance has only been tackled in the context of rectangular architectures, by introducing a regular pattern of redundant electrodes [51, 78]. Hence, there is an imperative need for design methodologies for application-specific DMBs that are fault-tolerant to permanent faults.

In this thesis, we propose an approach to synthesize a fault-tolerant application-specific architecture, such that the cost is minimized and the timing constraints of the application are satisfied. We address the placement problem by proposing an algorithm to build a library of non-regular modules for a given application-specific architecture, so that the area of a non-regular application-specific biochip can be used effectively. Our approach introduces redundant electrodes to synthesize fault-tolerant architectures aiming at increasing the yield of DMBs. We also propose a method to estimate, at design time, the application completion time in case of permanent faults in order to verify if an application can be successfully run on the architecture.

1.2 Contributions

First, we have addressed the problem of transient faults during the execution of the biochemical application. The erroneous variation of the droplets volumes due to transient faults, propagates throughout the execution of the bioassay. Eventually, the errors may lead to an incorrect result of the biochemical application.

- Initially, we have focused on the erroneous volume variation after an unbalanced split operation [1]. In Section 4.1 we present our compilation solution, which derives *offline* the schedules needed to recover from all combinations of faulty split operations. *Online*, the scheduler will switch to the backup schedules that tolerate the observed error occurrences.

- In Section 4.2 we have considered errors in all types of operations. Hence, we have proposed an online recovery strategy, which decides *online* and *dynamically* the optimization of redundancy required for fault-tolerance [2]. Our online optimization approach decides between time redundancy (re-execution of operations) and space redundancy (creating redundant droplets) depending on the current error scenario.

- As part of our recovery strategy we have proposed a biochemical application model, which captures the extra operations needed for redundancy (Section 2.4.2), together with an algorithm for generating the required redundant operations needed for fault-tolerance (Section 4.2).

Prior work has mainly considered general-purpose architectures for DMBs, such as the one in Fig. 1.3, which due to their rectangular shape are highly reconfigurable, making them suitable for multiple applications. In addition, in case of permanent faults, general-purpose architectures can be easily reconfigured to avoid the faulty electrodes [53]. However, in practice, there is an increased interest for development of application-specific biochips, see Fig. 1.4 and 1.5, because of their reduced costs. Hence, the second contribution of this thesis is an architecture synthesis of application-specific biochips [5]. We used two different metaheuristics to implement our architecture synthesis: Simulated Annealing [5] and Tabu Search [3].

- A search-optimization metaheuristic explores the solution space by starting from a given initial solution and generating new alternative solutions. To evaluate architecture alternatives, we have proposed in Section 5.3 a List Scheduling-based compilation that determines the worst-case completion time of an application running on an architecture that has maximum k permanent faults. In order to reduce the risk of eliminating good quality architectures by evaluating them using a worst-case approach, we have also proposed an estimation method which is less pessimistic (Section 5.4).

- We consider that the operations execute on circular-route modules (CRMs), which are routes of one-electrode thickness that start and end in the same electrode, and do not intersect themselves. Due to their non-regular shape, CRMs are suitable for a placement strategy that uses effectively the layout of application-specific biochips. In Section 3.2 we have proposed a method to determine, for a given application-specific architecture, the corresponding library of CRMs [6].

1.3 Thesis overview

The remainder of the thesis is organized in seven chapters as follows:

Chapter 2 introduces the biochip architecture and biochemical application models. We present in details the physical components of a biochip and we evaluate their impact on the architecture cost. We discuss different alternatives for operation execution and we introduce a fault-tolerant biochemical application model that captures the operations needed for error recovery. The existing fault models are presented and several fault-tolerant techniques are discussed.

Chapter 3 contains an overview of the DMBs design process. We discuss the methodology and we show detailed examples of both the application compilation and the architecture synthesis. We discuss the online and offline approaches for fault-tolerant compilation.

In **Chapter 4** we address the problem of transient faults during the execution of a biochemical application. First, in Section 4.1, we propose a fault-tolerant compilation approach that focuses on errors during split operations. We show that, by taking into account fault-occurrence information, we can derive better quality implementations, which leads to shorter application completion times, even in the case of faults.

In Section 4.2, we propose an online recovery strategy for all types of operations. Our approach decides during the execution of the biochemical application the introduction of the redundancy required for fault-tolerance. Error recovery is performed such that the number of tolerated faults is maximized and the timing constraints of the biochemical application are satisfied. The proposed redundancy optimization approach has been evaluated using several benchmarks.

The work presented so far in Chapter 4 considered general-purpose architectures, which have a rectangular shape for the electrode array. However, non-regular application-specific architectures are common in practice.

In **Chapter 5**, we address the application-specific architecture synthesis problem. We propose two solutions, based on metaheuristics, that determine, for a given application, the minimum cost architecture that can execute the application within the deadline even in the case of permanent faults.

Our first solution is a Simulated Annealing-based approach (Section 5.3), which uses rectangular modules for operation execution and exhaustive search to determine the worst-case execution time in case of faults. Our second solution is based on the Tabu Search metaheuristics, see Section 5.4. We present our algorithm that builds a library $\mathcal{L}$ of circular-route modules for operation execution and we propose a faster estima-

tion method for application completion time. We have run a series of experiments to evaluate and compare the two proposed architecture syntheses.

In **Chapter 6** we present our conclusions and we discuss the future directions of this work.

CHAPTER 2

System Models

This chapter presents the models used in the thesis. Section 2.1 presents the biochip architecture model. We address in this thesis both transient and permanent faults. The fault models and the associated assumptions are also outlined in Section 2.1.1.

We consider that operations are executed on "circular routes" and we present the models used for the operation execution in Section 2.2. The biochemical application model is presented in Section 2.3, and we propose in Section 2.4 extensions to this model to capture the fault-tolerance techniques required for recovery from transient faults.

2.1 Biochip architecture model

In a digital microfluidic biochip (DMB), a droplet is sandwiched between a top ground-electrode and bottom control-electrodes, see Fig. 2.2. The droplet is separated from electrodes by insulating layers and it can be surrounded by a filler fluid (such as silicone oil) or by air. Two glass plates, a top and a bottom one, protect the DMB from external factors. The electrodes are coated with a dielectric layer.

The droplets are manipulated using the electrowetting-on-dielectric (EWOD) principle [60]. When voltage is applied at the side of the droplet, an imbalance is created

in the interfacial forces between the electrode, droplet and the filler fluid. As a result, the droplet moves towards the side where the voltage was applied. For example, in Fig. 2.2, if the control-electrode on which the droplet is resting is turned off, and the left control-electrode is activated by applying voltage, the droplet will move to the left. In order to be actuated, the droplet has to be large enough to overlap the gap between the neighboring electrodes. Considering the biochip in Fig. 1.3, represented as an array of electrodes, a droplet can only move up, down, left or right with EWOD, and cannot move diagonally. A biochip is typically connected to a computer (or microcontroller) as shown in Fig. 1.3 and it is controlled based on an "electrode actuation sequence" that specifies for each time step which electrodes have to be turned on and off, in order to run a biochemical application.

The biochip contains devices such as input (dispensing) and waste reservoirs, sensors and actuators, on which the non-reconfigurable operations are performed. Sensors can be used to determine the result of the bioassay or for error detection. For example, a Light-Emitting Diode (LED) and a photodiode combination is used as detector for glucose concentration in a droplet [55, 71]. The location of these devices is fixed on the biochip array. In this thesis we assume that the locations of the reservoirs are on the boundaries of the array of electrodes. Fig. 1.3 shows the location of reservoirs and a sensor, which are placed on a biochip architecture of 10×8 electrodes. Example actuators are heaters [56] and filters [55].

An architecture of a DMB, denoted with $\mathcal{A}$ is modeled as a two-dimensional array of identical control-electrodes (see Fig. 1.3 and 2.1), where each electrode can hold a droplet. We distinguish between two types of architectures: general-purpose architecture and application-specific architecture. Most general-purpose DMBs have architectures of rectangular shapes, such as the one in Fig. 1.3, while an application-specific DMB has an non-regular layout, see Fig. 2.1.

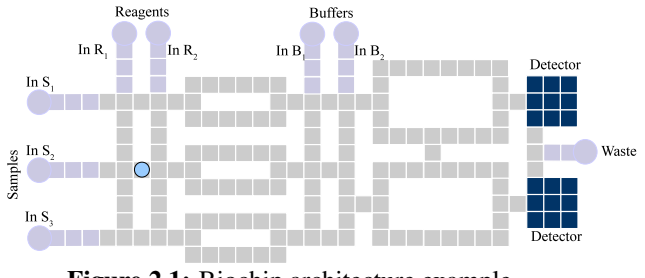


Figure 2.1: Biochip architecture example

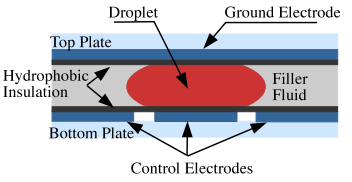


Figure 2.2: EWOD example

2.1.1 Fault models

Many biochemical applications, such as drug development and clinical diagnostics, have high accuracy requirements. DMBs can be affected by faults, resulting in failure to complete the application or in an incorrect result of the bioassay. Hence, researchers have addressed faults by proposing fault models [88], testing and detection methods [82, 81] and error recovery strategies [34, 47]. Faults can be classified in two main categories: (1) permanent faults and (2) transient faults. In Table 2.1 we present the most common types of faults.

Permanent faults. Also known as "catastrophic faults", permanent faults are caused by defects introduced usually during the fabrication of the DMBs. Permanent faults prevent the operation from executing. Some of the typical causes leading to permanent faults are given as follows [34, 81]:

- *Dielectric breakdown*—is caused when applying high voltage levels. A short between the droplet and the electrode is created, preventing further actuation of the droplet, i.e., the droplet is stuck on the electrode. Fig. 2.3 shows a DMB with dielectric breakdown defect.
- *Short between adjacent electrodes*—merges the two electrodes into a large one. In that case, the droplet becomes too small to overlap the adjacent electrodes, thus it cannot be actuated.
- *Degradation of the insulator* (see Fig. 2.4)—is caused by aging and repeated use of the same electrode. This defect is unpredictable and it results in fragmented droplets because of erroneous variation of surface tension forces.
- *Open in the metal connection between the electrode and the control source*—results in failure to charge the electrode, and thus prevents actuation of droplets.

Researchers have proposed several methods of testing for permanent faults in DMBs. Most of the test methods use test droplets that move on the biochip array according

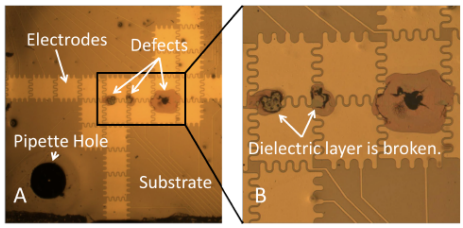


Figure 2.3: Dielectric breakdown [34]

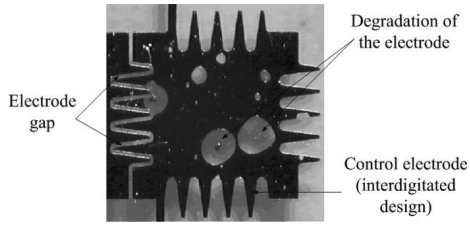


Figure 2.4: Insulator degradation [83]

Table 2.1: Types of faults in DMBs [31]

Fault name	Fault model	Cause of fault
Dielectric breakdown	Droplet-electrode short	Excessive actuation voltage applied to an electrode
Irreversible charge concentration on an electrode	Electrode stuck-on	Electrode actuation for excessive duration
Misalignment of parallel plates	Pressure gradient	Excessive mechanical force applied to the DMB
Non-uniform dielectric layer	Dielectric islands	Coating failure
Metal connection between two adjacent electrodes	Electrode short	Abnormal metal layer deposition and etch variation during fabrication
Broken wire to control source	Electrode open	
Grounding failure	Floating droplets	
Particle contamination	Contamination	Adsorption of proteins at electrode surface

to a testing scheme. For example, a straightforward testing scheme is the parallel-scan [86], which sends droplets first on the columns and then on the rows of the biochip array. A capacitive sensor is used to detect the presence of the droplets at the expected position. In case an electrode on the droplet's route is affected by permanent faults, the droplet will get stuck at an intermediate position and not reach the destination electrode. Researchers have focused on testing schemes that optimize the testing time and the number of dispensing reservoirs for test droplets [83, 81].

Transient faults. Also known as "parametric faults", transient faults occur unpredictably during the execution of an operation. Although transient faults do not prevent the operation from executing, the result of the operation does not correspond to its specified behavior. Some of the typical causes leading to transient faults are given as follows [34, 81]:

- *Misalignament between the droplet and the control-electrode*—is the most frequent cause of unbalanced split operation. Erroneous variation in droplet volume can have a significant negative impact on the outcome of the biochemical application. Estimates show that erroneous variation in droplet volume can count to up 80% of the total error in a bioassay [24]. An example erroneous split operation, i.e., resulting in unbalanced droplets, is presented in Fig. 2.5.

- *Change in viscosity of droplets and filler fluid*—caused by temperature variation during operation execution induced by an abnormal environment. As a result,

the concentration of droplets after a mixing operation can be erroneous.

- *Particle contamination*—occurs when the droplet or the filler fluid gets contaminated by a particle, which gets attached to the insulator surface of an electrode. In case of contamination, the result of the bioassay cannot be trusted.

DMBs can have integrated sensors that operate at a speed comparable to the execution time of a fluidic operation. Such sensors facilitate real-time error detection and recovery. A LED and photodiode sensor can be used for determining the concentration of a specific compound in a droplet (e.g., glucose [72, 71]), whereas a capacitive sensor [61] can be used to determine the volume of a droplet. The capacitive-detection circuit used to measure the volume operates at high frequency (15 KHz [60]), while the LED-photodiode sensor needs 5 seconds to measure the absorbance of the product droplet in order to determine its concentration. For the photodiode detector, a transparent droplet has to be mixed with a reagent to generate a colored analyte. In this case, the initial droplet is not suitable for other operations. The capacitive sensor does not alter the initial droplet, which can be used for subsequent operations.

Erroneous droplet volumes can also be detected by using a Charged-Coupled Device (CCD) camera-based detection system (see Section 4.2.5), which analyzes the images captured during the bioassay execution [44]. The CCD camera-based detection system adds to the complexity of the system by requiring external instruments and specialized software, but has the advantage of detecting the errors when they occur, eliminating the need for specialized detection operations, which have to transport a droplet to a sensor on the biochip.

In this thesis we consider permanent faults in the context of architecture synthesis (Section 5.1). Our solutions to the architecture synthesis problem introduce redundant electrodes in order to determine an application-specific architecture that is fault-tolerant to permanent faults, i.e., the application can be completed within the deadline even in the presence of k permanent faults. We assume that our architecture synthesis is part of a methodology, presented in Chapter 3, and thus, the architecture is tested after fabrication in order to determine the actual locations of faults.

Figure 2.5: Unbalanced split [20]

We consider transient faults during operation execution. First, we focus on the transient faults during split operations. We propose an fault-tolerant application that assumes maximum s faults during split operations (Section 2.4.1). Our proposed fault-tolerant model is used in an offline compilation approach (Section 4.1) to recover from the faulty split operation.

Then, we consider faults in all operations, and we propose a fault-tolerant application model (Section 2.4.2) and an online compilation strategy to decide at runtime the appropriate actions for error recovery (Section 4.2). In this thesis, for transient faults, which may result in erroneous droplet volumes, we use both capacitive sensors and a CCD camera-detection system for determining the volume of a droplet, which is then compared to its expected volume in order to perform error detection.

2.2 Operation execution

There are two types of operations: *reconfigurable operations* (mixing, split, dilution, merge, transport), which can be executed on any electrode on the biochip, and *non-reconfigurable operations* (dispensing, detection), which are bound to a specific device such as a reservoir, a detector or a sensor. Based on experiments, researchers characterize a module library $\mathcal{L}$, such as the one in Table 2.2, which provides the area and corresponding execution time that are needed for each operation. As shown in Table 2.2, the time needed for two droplets to mix on a 2×5 module is 2 s.

Researchers have used several approaches to group the electrodes in the architecture array and thus form "virtual devices" on which the operations execute [50]. The straightforward approach used by most researchers is to consider a rectangular area of electrodes, called a "module". Fig. 1.3 shows an example of a 2×5 module. In case two droplets are on neighboring electrodes, they merge instantly. Hence, in order to avoid accidental merging, each module is surrounded by a "segregation border" of one-electrode thickness, as shown in Fig. 1.3.

A mixing operation is executed when two droplets are moved to the same location and then transported together according to a specific pattern (see Fig. 1.3). Mixing of two droplets happens due to diffusion. In order to study the process, a fluorescent droplet and a non-fluorescent droplet are brought together and left to diffuse over a period of time, while the levels of gray are measured for the product droplet [17]. However, it has been observed that mixing can be enhanced if diffusion is helped by movement [58].

When droplets are moved back and forth on a linear mixer, a regression is noticed during the backward moves, due to flow reversibility. A 2×2 mixer eliminates the flow reversibility issue, but does not provide better results (see Table 2.2), because the

Table 2.2: Example module library $\mathcal{L}$ [76]

Operation	Area	Time (s)
Mix	2×5	2
Mix	2×4	3
Mix	2×3	6
Mix	1×3	5
Mix	2×2	10
Dilution	2×5	4
Dilution	2×4	5
Dilution	3×3	10
Dilution	1×3	7
Dilution	2×2	12
Store	1×1	N/A
Transport	1×1	0.01

droplets are pivoting around a single point. Having more pivoting points, like in a 2×3 mixer, enhances the mixing. Increasing the movement on one direction is also beneficial for mixing, so, as seen in Table 2.2, even better results are obtained for a 2×4 mixer (3 s) in comparison to a 2×3 mixer (6 s). The two droplets from Fig. 1.3 are mixing on a 2×5 module, by moving according to the indicated pattern.

A split operation is done by keeping the electrode on which the droplet is resting turned off, while applying concurrently the same voltage on two opposite neighboring electrodes. For example, in Fig. 2.2, to split the droplet, we have to turn off the control-electrode in the middle and turn on simultaneously the left and right control-electrodes. Dilution is a mixing operation followed by a split operation.

To dispense a droplet from the reservoir, several electrodes are activated to form a "finger" droplet, which is afterwards split to obtain the final droplet [61].

For droplet transportation, we use the data from [60], thus we assume that routing a droplet between two adjacent electrodes takes 0.01 s (see the "Transport " operation in Table 2.2).

2.2.1 Circular-route module

As mentioned, each reconfigurable operation is executed in a determined biochip area, called a "module". Similar to the "black-box" approach, all the electrodes forming such a rectangular module are considered occupied during the operation execution, and cannot be used by other operations. However, an operation is not necessarily confined

Table 2.3: CRM library $\mathcal{L}$ for the architecture in Fig. 2.6a

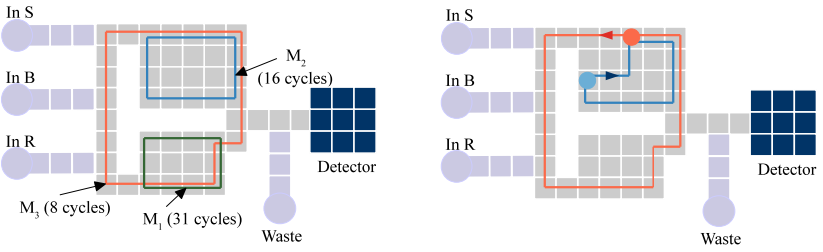
Operation	CRM	Time (s)
Mix	M_1	2.7
	M_2	2.1
	M_3	2.08
Dilution	M_1	5
	M_2	3.92
	M_3	3.9

to a rectangular area and it can execute anywhere on the microfluidic array as is the case with the routing-based compilation [52], which allows operations to execute on any route.

The advantage of routing-based operation execution is that it utilizes better the available biochip area. However, some biochemical applications use protein-based compounds that can leave residues behind [93]. The disadvantage of routing-based operation execution is that it makes it difficult to avoid contamination. Therefore, [52] later advocated a routing-based operation execution constrained to a given area [53].

We use a similar approach in this thesis, and we constrain the routing-based operation execution to a given "circular route". We define a *Circular-Route Module* (CRM) as a route of one-electrode thickness which starts and ends in the same electrode, and does not intersect itself. Given a CRM, a droplet will move repeatedly on the route until the operation is complete. We denote such a CRM with M_i. Fig. 2.6a shows three examples of CRMs, M_1, M_2 and M_3.

A "droplet-aware" operation execution is proposed in [53], based on the assumption that we know the position of the droplets during the execution of the operation. Thus, only the electrode holding the droplet and the adjacent electrodes are considered occupied (to avoid accidental merging). The rest of the electrodes assigned to the CRM are



a) Application-specific architecture

b) Adjusted route to avoid droplet merging

Figure 2.6: Example of circular-route modules

not considered occupied, and can be used for other operations. As a consequence, the routes for different operations may overlap over several electrodes. In order to avoid undesired droplet merging for intersecting routes during runtime, we instruct one of the droplets to take a detour from its predetermined route as shown in Fig. 2.6b or to wait until the other droplet passed by. In addition, to avoid contamination, we can capture restrictions for the operations that have specified contamination conflicts.

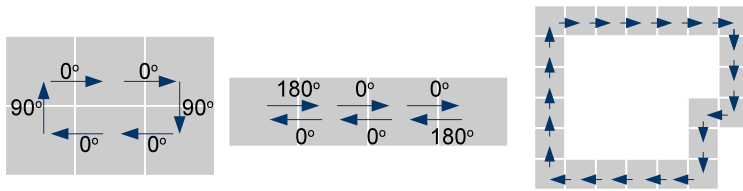
We use the module decomposition approach proposed in [52] to estimate the operation completion time for each CRM. In [52], the droplet movement during an operation is decomposed into basic movements and the impact of each basic movement on the operation execution is calculated.

As seen in Fig. 2.7a, on a 2×3 mixer, a cycle is completed by forward movements (0°), followed by turns (90°). On a 1×4 mixer (see Fig. 2.7b), the droplets complete one cycle in 3 movements: one backward movement (180°) followed by two forward movements (0°). Using an experimentally determined library, that contains information about the execution times of the operations, the method proposed in [52] estimates, for each movement, the percentage completion towards operation execution.

Thus, we can determine p^0_{cycle}—the percentage towards operation completion for a cycle when there are no faults, using the following equation:

$$p^0_{cycle} = n^1_{0^\circ} \times p^1_{0^\circ} + n^2_{0^\circ} \times p^2_{0^\circ} + n_{180^\circ} \times p_{180^\circ} + n_{90^\circ} \times p_{90^\circ}, \tag{2.1}$$

where $p^1_{0^\circ}$, $p^2_{0^\circ}$, p_{180°, p_{90° are the percentages towards operation completion for a forward movement for one electrode, a forward movement for at least two consecutive electrodes, a backward movement and a turn, respectively, and $n^1_{0^\circ}$, $n^2_{0^\circ}$, n_{180°, n_{90° are the number of forward movements for one electrode, forward movements for at least two consecutive electrodes, backward movements and turns, respectively. Then, we determine n_i—the minimum number of times the droplets have to rotate on a given circular route to achieve at least 100% operation completion. Fig. 2.6a shows n_i for each of the three CRMs, 31 for M_1, 16 for M_2 and 8 for M_3. The total execution time is obtained by multiplying n_i with the time needed to complete one rotation.



a) 2x3 module, t=6.1 s b) 1x4 module, t=4.6 s c) circular-route module, t=2.2 s

Figure 2.7: Module decomposition approach for operation execution

For example, for the route depicted in Fig. 2.7c, the droplets need to cycle 10 times in order to complete the mixing operation, resulting in an execution time of $t = 2.2$ s. We have used the following values for the percentages towards operation completion: $p^1_{0^\circ} = 0.29\%$, $p^2_{0^\circ} = 0.58\%$, $p_{180^\circ} = -0.5\%$, $p_{90^\circ} = 0.1\%$ [52].

In this thesis, we propose an algorithm (see Section 3.2) that determines, for a given application-specific architecture, a module library $\mathcal{L}$, which provides the shape of each CRM M_i and the corresponding execution time needed for each operation. For example, for the architecture in Fig. 2.6a, we have determined the CRM library shown in Table 2.3. The library $\mathcal{L}$ is used during compilation to determine the application completion time.

2.2.2 Worst-case operation execution overhead in case of permanent faults

Permanent faults, which are introduced during fabrication, can affect the execution of the biochemical application, see Section 2.1.1. We want the biochemical application to run within its deadline even in the case of maximum k permanent faults. In case a module contains faulty electrodes, the droplets need to be re-routed in order to avoid the permanent faults. Let us consider the module from Fig. 2.8b, where the faulty electrodes are marked with "x". In order to avoid the faults, the droplets are instructed to take a detour (i.e., are re-routed), as shown in Fig. 2.8b.

In order to determine if an application $\mathcal{G}$ finishes within its deadline even in case of faults, we run a compilation that determines the application completion time $\delta^k_{\mathcal{G}}$ in case of k faults. We denote with C_i the operation execution time without fault tolerance and with C^k_i the operation execution time in case of k faults. The compilation uses the execution time C^k_i of the operation O_i to determine $\delta^k_{\mathcal{G}}$. The value of $\delta^k_{\mathcal{G}}$ depends on the location of the k permanent faults.

In this thesis, we use the compilation in two situations, see the methodology in Chapter 3, as follows.

(i) Once a biochip has been fabricated, we use compilation to determine the electrode actuation sequence needed to run the application. In this case, the actual locations of the permanent faults are known, and are used to update the worst-case operation execution time.

(ii) Before a biochip has been fabricated, inside a design space exploration which searches for an application-specific biochip architecture. In this case, we do not yet know the locations of the permanent faults. Next, we present how we estimate C^k_i in

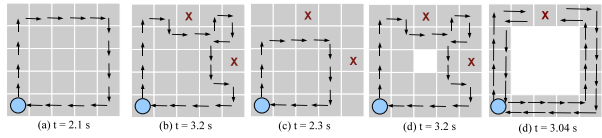


Figure 2.8: Worst-case operation execution time in case of permanent faults

this case. In this section we present a pessimistic but safe exhaustive approach, and in Section 2.2.3, we present a fast, but potentially unsafe approach (safe means that for all the possible patterns of permanent faults, the actual execution time of O_i is not larger than our determined C_i^k). The advantages and disadvantages of each approach to operation execution estimation in case of permanent faults are discussed in Section 5.1.

In the first approach, we consider that *each module* placed on the biochip suffers from k faulty electrodes (note that the k faults are for the entire biochip), and we propose a technique to determine the overhead of the k permanent faults on an operation execution. Our approach is to determine the worst-case execution time C_i^k, i.e., the largest operation execution time among all possible combinations of k faults placed on the electrodes of the module M_i.

Because the modules have a small area, we use at design time an exhaustive search to determine, for each possible combination of k faults on the module's electrodes, the best new route which avoids the faults, and leads to the fastest operation execution. The largest time among these is C_i^k. Our proposed approach is general and it can handle both rectangular modules and CRMs.

Let us consider, for example, the module M_1 in Fig. 2.8a and determine C_1^2, which is the worst-case execution time for $k = 2$ permanent faults. Fig. 2.8b, c present two different routes that avoid the same combination of $k = 2$ faulty electrodes, marked with a red "x". The best route out of the two is the one shown in Fig. 2.8c, which completes the operation in 2.3 s.

By evaluating all possible routes the avoid a specific pattern of faulty electrodes, we can determine an optimal execution time. In our example, for the pattern of faulty electrodes in Fig. 2.8b, the optimal execution time is 2.3 s. Next, we determine the optimal execution times for all possible combinations of $k = 2$ faulty electrodes. The largest value among these is the worst-case execution time C_1^2, which for our example is 2.71 s.

2.2.3 Estimation of operation execution in case of permanent faults

Determining the worst-case execution time through exhaustive search is very time consuming and cannot be used inside a search metaheuristic such as the one we propose for architecture synthesis, see Chapter 5.

Hence, we propose a faster method to determine an estimate of the operation execution time, which is less pessimistic than the worst-case value. Our estimation method considers CRMs and determines for a CRM M_i the corresponding operation execution time C_i^f, which is the time needed by a reconfigurable operation (e.g., mix, dilution) to complete on M_i considering f permanent faults, $f = 1$ to k, where k is the maximum number of permanent faults.

Let us denote with p_{cycle}^0 the percentage towards operation completion for a cycle when there are no faults. The value of p_{cycle}^0 is obtained using Equation (2.1) as explained in Section 2.2.1. We need to estimate p_{cycle}^f—the percentage towards operation completion for a cycle when there are f permanent faults. Once we know p_{cycle}^f, we calculate n_i^f—the number of cycles needed to achieve at least 100% operation execution. The value of C_i^f is obtained by multiplying n_i^f with the time needed for one cycle p_{cycle}^f.

In case a CRM contains faulty electrodes, the droplets need to be re-routed in order to avoid the permanent faults. Let us consider that the CRM in Fig. 2.9a has the faulty electrodes marked marked with "x" in Fig. 2.9b. In order to avoid the faults, the droplets are instructed to take a detour, as shown in Fig. 2.9b. Since the position of the faults is not known, in our previous approach we have used an exhaustive search to determine the execution time for the worst-case fault pattern occurring in a CRM.

As an alternative, here we propose a faster estimation heuristic, which, instead of performing an exhaustive search, makes some simplifying assumptions about the impact of the worst-case fault patterns on the operation execution.

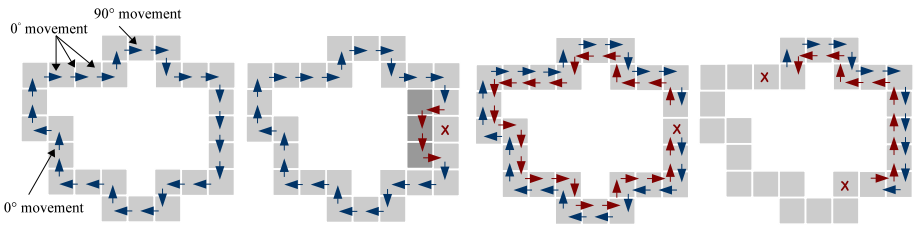


Figure 2.9: Estimation of operation execution time in case of permanent faults

One assumption is that the permanent faults form a pattern which can only be avoided by using backward movements (180° turns). Backward movements lengthen the operation execution time C_i, because they induce flow reversibility.

Fig. 2.9c and d show the routes taken by the droplets to avoid $f = 1$ and $f = 2$ permanent faults, respectively. However, if one of the faults is positioned so that it has non-faulty neighboring electrodes, the fault can be avoided by taking a detour (see Fig. 2.9b).

Backward movements have a negative impact on the operation execution (i.e., the operation execution is regressed). Hence, in most of the cases, using a backward movement can lead to a larger increase in execution time than taking a detour. Since we do not know the exact location of the faults, and consequently whether such detour-routes are possible, it is our assumption, as mentioned, that faults are avoided by using backward movements. Hence, instead of avoiding the faulty electrodes by finding a path around them, the droplets will be routed back and forth between two of the f faults, as shown in Fig. 2.9d. Consequently, the cycle of droplets on the CRM is reduced to the distance between two of the f faults. In case $f = 1$, the droplets will be routed as shown in Fig. 2.9c.

Another assumption is that, if there are more faults ($f > 1$), they are located such that they lead to the "most damage", i.e., the largest increase in C_i. We assume that this happens when the faults are located at equal distance on the CRM, as shown in Fig. 2.9d. Our assumption is based on the fact that a route with a higher frequency of backward movements will need more time to complete the operation.

Considering the assumptions mentioned above, we estimate p^f_{cycle} using the following equation:

$$p^f_{cycle} = p^0_{cycle}/f - 2 \times p^2_{0^\circ} + p_{180^\circ}, \tag{2.2}$$

where p^0_{cycle}, p_{180° and $p^2_{0^\circ}$ are the percentages towards operation completion for a cycle with no faults, a backward movement and a forward movement for at least two consecutive electrodes, respectively. Since we consider that the f faults are located at equal distance, we obtain in the first term in Equation (2.2), a rough estimation of p^f_{cycle} by dividing p^0_{cycle} to f. However, to be more precise, we take into account that 2 electrodes are occupied by faults (second term in Equation (2.2)) and that a 180° turn is needed (third term in Equation (2.2)). For the second term in Equation (2.2), we assumed we are loosing electrodes which contribute most towards operation completion (i.e., the completion percentage is $p^2_{0^\circ}$).

The value of p^0_{cycle} (determined using Equation (2.1), see Section 2.2.1), $p^2_{0^\circ}$ and p_{180° depend on the operation type and on the fluids used the operation. Hence, Equation (2.2) determines for each CRM a *parametric* estimation of the execution time, where the parameters are the percentages $p^1_{0^\circ}$, $p^2_{0^\circ}$, p_{180° and p_{90°. Once the binding of

the operations is decided, i.e., we know which operations are assigned to each CRM, then we introduce in Equation (2.2) the corresponding values for p^0_{cycle}, $p^2_{0^\circ}$ and p_{180°.

For our example we use the following values: $p^2_{0^\circ} = 0.58\%$ and $p_{180^\circ} = -0.5\%$ [52]. Considering $k = 2$ permanent faults for the CRM M_1 in Fig. 2.9a, we estimate, using Equation (2.2), the following execution times: $C^1_1 = 4.8\ s$ and $C^2_1 = 5.28\ s$ (see Fig. 2.9c, d). These operation execution values C^f_i are used inside the compilation to determine the application completion time $\delta^k_{\mathcal{G}}$.

2.3 Biochemical application model

A biochemical application is modeled using a Directed Acyclic Graph (DAG) model [14], where the nodes represent the operations, and the edges represent the dependencies between them. We have extended the model proposed in [14] to model error detection and error recovery and to capture the operations needed for recovery.

We denote with $\mathcal{G}^0$ the biochemical application model without fault-tolerance features. Fig. 2.10 presents such an application graph $\mathcal{G}^0$ with 15 operations. A node in $\mathcal{G}^0$ represents an operation O_i. In Fig. 2.10 we have operations O_1 to O_{15}. A directed edge e_{ij} between operations O_i and O_j models a dependency: O_j can start to execute only when it has received the input droplet from O_i. An operation is ready to execute only after it has received all its input droplets. For example, in Fig. 2.10, the mixing operation O_6 is ready to execute only after operations O_3 and O_4 have finished executing and the droplets have been transported to the biochip area where O_6 will perform the mixing. If the produced droplet cannot be used immediately (e.g., has to wait for another operation to finish), it has to be stored in a storage unit (see Table 2.2) to avoid accidental merging. In our model, we do not capture explicitly the routing operations required to transport the droplets, but we take routing into account during the compilation.

Biochemical applications can have strict timing constraints. For example, in the case of sample preparation, the reagents degenerate fast, affecting the efficiency of the entire bioassay [32]. In addition, operations can have local deadlines. For example, once two droplets are mixed, they should not wait more than a certain time before they are subsequently used (e.g., the reactions of aryllithium compounds bearing alkoxycarbonyl groups [89]). We can easily model such local deadlines by introducing dummy nodes in the application graph, and by having a global application deadline.

In this thesis we assume a hard deadline $d_{\mathcal{G}}$, but our approach can be extended to handle soft deadlines, with the aim of maximizing the utility. The deadline $d_{\mathcal{G}}$ is hard, i.e., the application is considered faulty if it does not complete within $d_{\mathcal{G}}$, even in the case of faults. However, many biochemical application can have soft deadlines, where there is

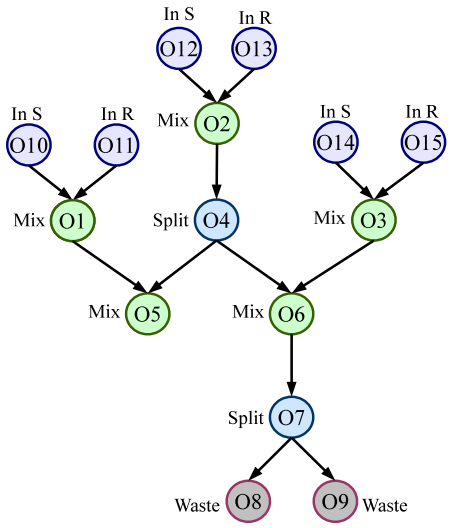


Figure 2.10: Biochemical application graph $\mathcal{G}^0$

still some utility in continuing to execute the application after the deadline. We discuss this issue in Section 6.2 on future work.

2.4 Transient faults and fault-tolerance models

DMBs can experience permanent and transient faults, as discussed in Section 2.1.1. In this section we present our transient fault model and we show how the application model from the previous section can be extended to capture the fault-tolerance required to recover from the transient faults.

During the execution of the biochemical application, the droplets will naturally undergo changes in volume during mixing, dilution and split operations. For example, when two droplets merge for a mixing operation, the resulting droplet has a volume equal with the sum of the input droplets volumes. After a split operation, the resulting droplets have volumes equal to half of the initial droplet volume. However, the volume of a droplet can also vary erroneously due to transient faults, such as an electrode coating fault or unequal actuation voltages during split [88]. An example of a faulty split operation is presented in Fig. 2.5. The erroneous droplet volume propagates throughout the execution of the bioassay, thus impacting negatively the correctness of the application.

Section 2.4.1 presents an application model that captures the fault-tolerance required to recover from faulty split operations. This model will be used in Section 4.1 to compile offline a fault-tolerant implementation. Section 2.4.2 presents a more general application model, which can capture the fault-tolerance required for transient faults in all types of operations, not only the split operations. This model will be used in Section 4.2 to compile online, during the application execution, the required recovery actions.

2.4.1 Fault-Tolerant Sequencing Graph

Let us now discuss our application model for transient faults in split operations. In this context, our assumption is that there can be at most s faults in the split operations of an application.

In order to determine if a split operation is faulty, we bring one of the resulted droplets to a capacitive sensor which measures the droplets volume. Two outcomes are possible after a detection operation. The first one corresponds to a correct droplet volume, and the second one to an erroneous droplet volume. In case the measured volume is the expected one, i.e. no error has occurred, the corresponding droplet is transported from the sensor to the location where the subsequent operations will execute. Otherwise, in case the split operation is erroneous, the resulted droplets are merged back and the split is re-executed. In the worst-case, a split will have to be performed $s + 1$ times, to tolerate the maximum s faults that can happen in the application. The last split does not have to be followed by a detection operation, since we know it will not experience an error: all faults have already happened. Note, however, that these s faults can happen in any of the split operations of the application.

We propose a Fault-Tolerant Sequencing Graph (FTSG) $\mathcal{G}^S$ to capture all fault scenarios considering maximum s faulty split operations. In $\mathcal{G}^S$, each split operation is followed by a detection operation which detects if a fault has occurred. Each split operation O_i is transformed into a structure which models all possible fault occurrence scenarios. Let us consider the initial application graph $\mathcal{G}^0$ in Fig. 2.10 and the corresponding FTSG in Fig. 2.11. O_4 is transformed into the structure that starts with node $O_{4.1}$ in Fig. 2.11. We use the notation convention $O_{i.x}$ to denote the x^{th} copy of the split operation O_i inserted in $\mathcal{G}^S$. Each such split operation is followed by a detection operation. As shown in Fig. 2.11, the detection operation D_{16} was introduced in $\mathcal{G}^S$ after the split operation $O_{4.1}$. Note that operations O_8–O_{15} from Fig. 2.10 are depicted in Fig. 2.11 as "...". During a detection operation, one of the droplets resulted from the previous split operation is routed to the sensor for error detection. The number n_{sns} of the sensors and their placement on the biochip are decided during the compilation phase.

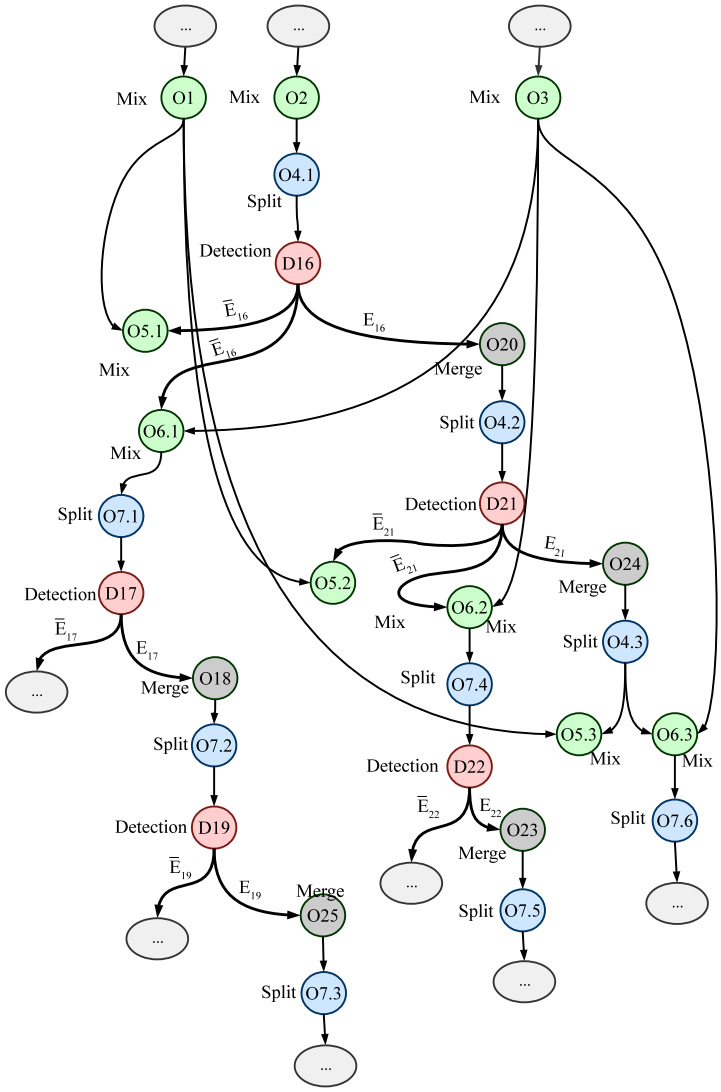


Figure 2.11: Fault-Tolerant Sequencing Graph

Each detection operation is followed by two conditional edges corresponding to the faulty and non-faulty split scenarios, respectively. A *conditional edge* is a dependency between two operations, which is activated only when the associated condition is true. Conditional edges are used to model the outcome of a detection operation D_i. Let us assume that D_i will produce an error condition E_i, which is true if an error has been detected and false if an error has not been detected. Thus, D_i will have two outgoing

conditional edges, labeled with E_i and $\overline{E}_i$. We call such an operation with outgoing conditional edges a *disjunction node*.

For example, for the detection operation D_{16}, we insert the following conditional edges: $D_{16} \rightarrow O_{20}$ under the condition of a fault occurrence E_{16}, and edges $D_{16} \rightarrow O_{5.1}$ and $D_{16} \rightarrow O_{6.1}$, under the condition of no fault occurrence $\overline{E}_{16}$, respectively. On the faulty branch, we have to add a merge operation (O_{20}) and a recovery split operation ($O_{4.2}$). For both scenarios, we have to copy from $\mathcal{G}^0$ the subgraphs originating from the split operation. Hence, in case a fault is detected by the detection operation D_{16}, the condition on edge $D_{16} \rightarrow O_{20}$ is satisfied and node O_{20} is activated. In this case, the two resulting droplets are merged back into the initial one, and the split operation is repeated. However, if the detection operation does not detect a fault, nodes $O_{5.1}$ and $O_{6.1}$ are activated instead.

We continue the transformation with the next split operations, including those introduced in $\mathcal{G}^S$ by the previous transformations. The process continues until all possible alternative scenarios are built. A scenario represents the fault pattern of maximum s transient faults that can happen during the split operations from $\mathcal{G}^0$.

The graph in Fig. 2.11 assumes a maximum number of 2 faults which can occur on the split operations O_4 and O_7. The split operation $O_{4.2}$ is placed on the faulty branch originating from the detection operation D_{16}, which means that a fault has already occurred (in $O_{4.1}$). Since $s = 2$, another fault can occur, which means that $O_{4.2}$ has to be followed by a detection operation, D_{21}. Our construction procedure keeps track of the fault occurrence to build the structure of $\mathcal{G}^S$. On the faulty branch from O_{21} we introduce the recovery split operation $O_{4.3}$. However, $O_{4.3}$ is not followed by a detection operation, since we are currently in the scenario when both faults have already occurred (first in $O_{4.1}$ and second in $O_{4.2}$).

There are 6 possible scenarios in this particular case: $\emptyset$—no faults at all; $\{O_4\}$—one fault during O_4; $\{O_7\}$—one fault during O_7; $\{O_4, O_7\}$—two faults, one during O_4, and one during O_7; $\{O_4, O_4\}$—two faults during O_4; $\{O_7, O_7\}$—two faults during O_7. These six alternative scenarios are captured in the FTSG in Fig. 2.11.

2.4.2 Generalized Fault-Tolerant Application model

The previous FTSG model captures transient faults only in the split operations, and assumes a single recovery technique: merging the incorrect droplets and splitting them again, which is similar to re-execution, a form of time redundancy. In this section we propose a Generalized Fault-Tolerant Application (GFTA) model that addresses transient faults in all operations, and we consider several recovery techniques. In this context, our transient fault models does not make any assumptions on the number of

maximum transient faults that can happen, i.e., we can capture any number q of transient faults.

2.4.2.1 Error propagation and error detection

Errors due to transient faults can propagate from one operation to another operation, eventually impacting negatively the correctness of the bioassay's results. In [95], the authors use error analysis [84] to derive the error limit at the output of an operation from its intrinsic error limit and the limits of the input operations. The assumption is that each fluidic operation has a specific error range associated with it, called "intrinsic error limit", which captures the worst-case volume variations.

For example, if the intrinsic error limit E_{Mix} for mixing is 10%, after a mix operation the output droplet can have a volume between 90% and 110% of the nominal value. We use the following notation: E_{Mix} is the intrinsic error limit for mixing operation, E_{Dlt} for dilution, E_{Trans} for transport, E_{Ds} for dispensing, E_{Slt} for split. Experimentally, the following values were determined for the intrinsic error limits: $E_{Ds} = E_{Dlt} = E_{Slt} = 8\%$, $E_{Mix} = 10\%$, $E_{Trans} = 12\%$ [95].

The equations in Fig. 2.12c [95] calculate the error limit ε_{Mix} at the output of a mixing operation, ε_{Ds} for dispensing, ε_{Dlt} for dilution, ε_{Trans} for transporting and ε_{Slt} for split operations as a function of intrinsic error limits E_{Mix}, E_{Ds}, E_{Dlt}, E_{Trans} and E_{Slt} respectively, and input error limits I_1 and I_2. The error limit at the output of an operation is propagated and becomes the error limit for its successor operation. In Fig. 2.12b, for the dilution operation O_4 we have the intrinsic error $E_{Dlt} = 8\%$ and the input operation error limits $I_1 = 11.4\%$ (for O_3) and $I_2 = 8\%$ (for O_9). Using Eq.(v) from Fig. 2.12c, we estimate the error limit at the output of O_4 to be 17.4%.

We continue to calculate the error limits for all fluidic operations in the biochemical application. For every bioassay, according to its specific accuracy requirements, the designer decides on a specific volume variation boundary E_{Thr}, named threshold error, which is the maximum permitted variation from the nominal volume. When the error after an operation O_i, calculated according to the presented error analysis, exceeds the error threshold E_{Thr}, a detection operation D_i is inserted into $\mathcal{G}^0$ to detect at runtime if an error has actually occurred or not.

For the graph in Fig. 2.12a, the E_{Thr} was set to 12%; as a result, the detection operations D_4 and D_7 were inserted into $\mathcal{G}^0$ after O_4 and O_7, respectively, obtaining $\mathcal{G}^+$, as depicted in Fig. 2.12b. In case O_i is an operation with two output droplets (e.g. the dilution operation O_4 in Fig. 2.12b), the detection operation will have two inputs, as in the case with operation D_4 in Fig. 2.12b. However, it is sufficient to measure the volume of only one droplet in order to determine if an error has occurred.

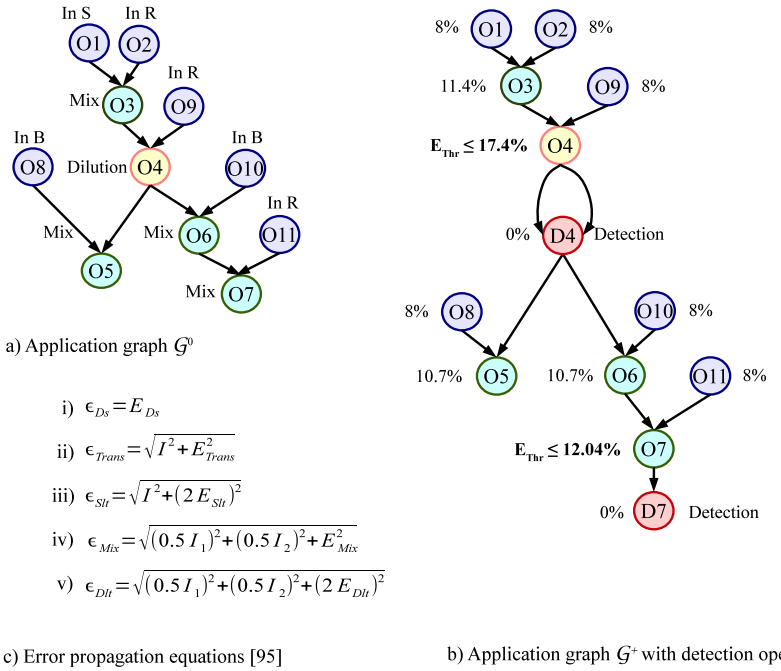


Figure 2.12: Example application model, with error propagation and detection

After each detection, we reset the error limit to 0%, since it is assumed that in case an error is detected, the necessary actions to recover from the error are taken. The assumption is that a volume error occurring in an earlier operation can also be detected later, after it has propagated. For operations where this is not the case, the designer will statically assign a corresponding detection operation at a pre-determined place in the graph. Researchers have so far assumed that all the detection operations are statically assigned. However, in Section 4.2.4.1, where we propose an online redundancy optimization and recover strategy, we discuss how to assign dynamically the detection operations by adjusting at runtime the error threshold E_{Thr} based on the current fault occurrences.

If the volume of a droplet is detected as erroneous, we have to create a new similar droplet with the correct volume (i.e., we recover from the detected error). This can be done in several ways. The simplest solution is to discard all the operations executed so far and re-execute the entire application from the beginning. However, this is very time-consuming, especially for the cases when errors occur at later stages. For most applications, a complete re-execution results in exceeding the deadline and wasting expensive reagents and hard-to-obtain samples. For example, in Fig. 2.12b, if an error

is detected in D_7, we have to re-generate the droplets needed for O_7. In this case, we do not need to re-execute operations O_5 and O_8.

In our approach, we use three strategies to create droplets with the correct volume:

1. We re-execute the operations needed to re-generate the droplet, *after* an error has been detected. We call such an approach *time redundancy*. The advantage of time redundancy is that it re-executes operations only when needed (when an error has been detected); the disadvantage is that it leads to delays in the application execution.

2. We execute operations which will produce a correct droplet *before* we know if an error has occurred, in parallel to the application execution. We call this approach *space redundancy*. The advantage of space redundancy is that, if an error is detected, we can use the redundant correct droplet directly, without waiting to be re-generated. The goal is to use the extra biochip area, if available, to speculatively produce correct droplets, without a negative impact on the application execution. The disadvantage is that if not enough area is available, space redundancy will introduce delays during the application execution, since it competes for the same resources with the regular operations.

3. We use the redundant droplets available as a by-product of the regular application execution or after using space and time redundancy for other operations. For example, if we use only one droplet after a dilution operation, we can use the second droplet for fault tolerance, if it has the correct volume. Let us assume that we need to re-generate the droplets for an operation O_j. If we predicted a fault in a predecessor operation O_i of O_j, and we used space redundancy for O_i but an error has not been detected after O_i, we may be able to use in O_j some of the redundant droplets produced by space redundancy for O_i.

2.4.2.2 Redundancy models

In all three strategies outlined earlier, we generate the correct droplets by using redundant operations in the application graph, corresponding to a detection operation D_i. These redundant operations are grouped into a subgraph R_i, which is connected to the graph $\mathcal{G}^+$, i.e., the application graph $\mathcal{G}^0$ with the detection operations. These subgraphs are responsible for producing correctly-sized droplets, and are inserted into $\mathcal{G}^+$ such that output droplets produced by R_i become the input droplets for the successors of operation D_i.

Fig. 2.13b shows the recovery subgraph R_{11} for detection D_{11} in the graph in Fig. 2.13a. A recovery subgraph R_i can be obtained at design time by performing a breadth-first search on the graph $\mathcal{G}^+$, starting from O_i and going backwards towards the inputs. Note that, as shown in Fig. 2.13c, not all the operations in R_i will be needed at runtime

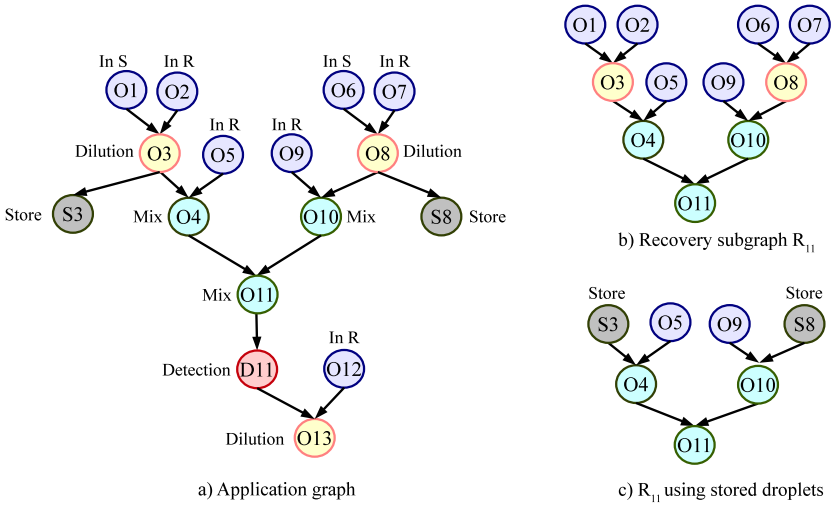


Figure 2.13: Example of recovery subgraph

because redundant droplets may be already available, as discussed at point 3 earlier. Our online compilation strategy from Section 4.2.4.1, will carefully manage these redundant droplets and will eliminate from R_i, at runtime, the superfluous operations for which such droplets are available.

Let us now discuss the difference between time and space redundancy in terms of how the subgraph R_i is connected to the application graph $\mathcal{G}^+$ and how it is executed in case of time and space redundancy. In the following models, we use the same concept of *conditional edge* as introduced in Section 2.4.1. In addition, we define an *execution guard* as a condition which has to be true in order to activate the operations of a redundant subgraph R_i.

Time redundancy. Fig. 2.14a presents how the subgraph R_i is connected to the graph $\mathcal{G}^+$ in case of time redundancy for an operation O_i followed by a detection D_i. The subgraph R_i is depicted using a rectangular node. Such a node is hierarchical, since it contains all the operations of R_i. Note that an error can occur also during the execution of the subgraph R_i used for recovery. We denote with D_i^R the detection operation needed to detect such an error, which occurs during the recovery. We denote with E_i and E_i^R the error conditions produced after the detection operations D_i and D_i^R, respectively.

With time redundancy, the subgraph R_i is activated if an error is detected by D_i or by D_i^R, i.e., if $E_i \vee E_i^R$ is true. This is depicted in Fig. 2.14a with an arrow on top of the rectangular node R_i, labeled with the execution guard $E_i \vee E_i^R$. Let us denote with O_B the successor operation of O_i (corresponding to the detection D_i). O_B will be activated

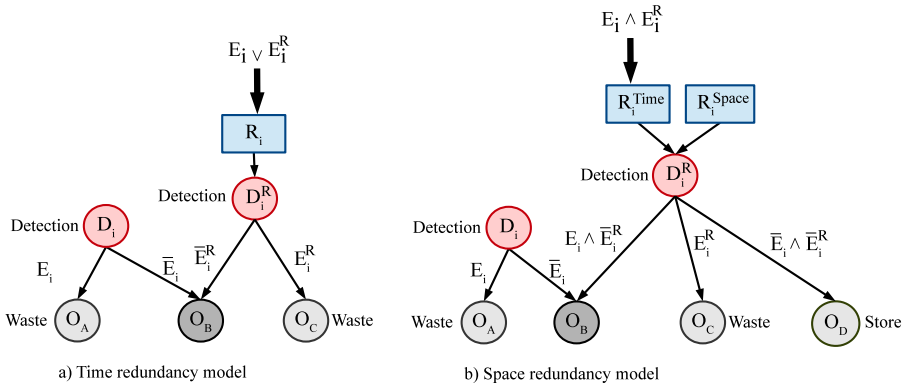


Figure 2.14: Recovery using time vs. space redundancy

only if no error is detected by D_i or no error is detected by D_i^R after the recovery subgraph R_i. This is captured in our model by connecting O_B with the conditional edges $\overline{E}_i$ and $\overline{E}_i^R$ to D_i and D_i^R, respectively.

If an error is detected by D_i or D_i^R, the corresponding incorrectly sized droplets will have to be discarded. This is achieved by inserting the operations O_A and O_C in the graph and connecting them to D_i and D_i^R using the conditional edges E_i and E_i^R, respectively. The operations O_A and O_C are responsible to transport the incorrect droplets to the waste reservoirs. In these cases, i.e., E_i or E_i^R are true, R_i is activated, as discussed.

Section 4.2.4 presents how R_i is compiled, including how its operations are scheduled, in order to be executed. We also compile the operations O_A and O_C which transport the incorrect droplets to the waste.

Because D_i^R detects an error during R_i and thus activates it again for execution, our time redundancy model tolerates several transient faults, constrained only by the deadline $d_{\mathcal{G}}$.

Space redundancy. Fig. 2.14b presents our space redundancy model. We use space redundancy to tolerate a *single* transient fault detected during a detection operation D_i. If a second transient fault is detected in the same place, we revert to time redundancy.

We denote with R_i^{Space} the subgraph R_i used for space redundancy in Fig. 2.14b and with R_i^{Time} the one used for time redundancy. For a detection operations D_i, we do not introduce more than one subgraph for space redundancy because they consume biochip area and, if a fault does not occur, too much space will be wasted.

Similar to time redundancy, we denote with D_i^R the operation needed to detect a fault in R_i^{Space} or R_i^{Time}, with O_B the successor operation of O_i and with O_A and O_C we denote waste operations. The main difference to the time redundancy model from Fig. 2.14a is that the subgraph R_i^{Space} used for space redundancy does not have an execution guard, i.e., it is executed regardless if an error is detected by D_i or not.

The advantage of space redundancy is that if an error is detected by D_i, we do not have to wait for the re-execution of R_i to get the correct droplets, as it is the case with time redundancy. Instead, O_B is ready to execute using the redundant droplet produced by R_i^{Space}. This is captured in the model in Fig. 2.14b by the conditional edge $E_i \wedge \overline{E}_i^R$ from D_i^R to O_B, which is activated only if an error has occurred in D_i and no error has occurred during the execution of R_i^{Space}. That is, we only use the redundant droplet from R_i^{Space} if it is of correct volume, condition checked by D_i^R, to which R_i^{Space} is connected, and captured by $\overline{E}_i^R$. Note that O_B may have to wait for R_i^{Space} to finish executing, if not all operations in it have completed.

In case an error has been detected by D_i and R_i^{Space} has also experienced an error, which was detected by D_i^R, we will use time redundancy (R_i^{Time}) to recover from these two errors. Hence, R_i^{Time} is only activated if both E_i and E_i^R are true. Any errors in R_i^{Time} will be handled as discussed for time redundancy. Finally, if there are no errors at all in Fig. 2.14b (i.e., $\overline{E}_i \wedge \overline{E}_i^R$), we are left with redundant droplets produced by R_i^{Space}. Our online compilation for recovery (discussed in Section 4.2, where the models presented here are used) will decide what to do with these droplets. For example, they can be stored to be used later during other recoveries. This is depicted in Fig. 2.14b with the "store" operation O_D, connected with the conditional edge $\overline{E}_i \wedge \overline{E}_i^R$ to D_i^R.

Fig. 2.15 presents the GFTA model $\mathcal{G}^R$ for the graph $\mathcal{G}^0$ in Fig. 4.11a. $\mathcal{G}^R$ was obtained after deciding to use time redundancy for and D_{13} and space redundancy for D_9 and D_{11}. As seen in Fig. 2.15, the corresponding recovery graphs R_{13}^{Time}, R_9^{Space} and R_{11}^{Space} were inserted according to the models presented above and depicted in Fig. 2.14.

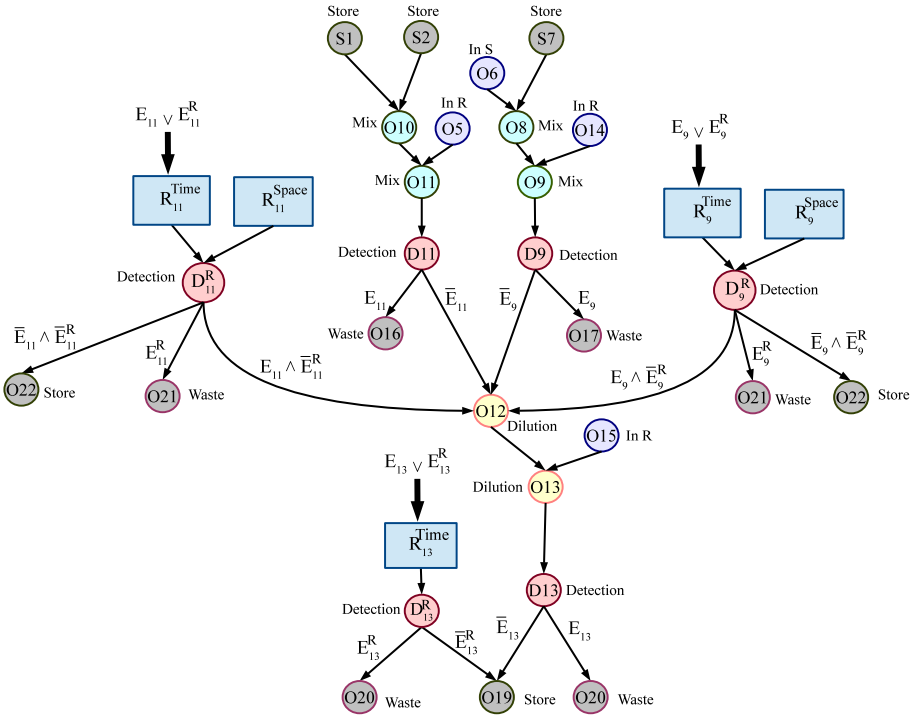


Figure 2.15: Example GFTA model for $\mathcal{G}^0$ in Fig. 4.11a

CHAPTER 3

Design Methodology for DMBs

This chapter presents the typical phases of a methodology for the design of DMBs. The purpose is to explain how the methods presented in this thesis are used within a design methodology and to define the main design tasks. Note that the methods proposed in this thesis can work with any methodology.

Before we can run a biochemical application on a DMB, we need to design its architecture and then we have to fabricate it. Once the biochip is available, we need to compile the biochemical application to produce the "electrode actuation sequence". Then, the application is run by actuating the biochip components using this sequence, see Fig. 1.3. Thus, a design methodology for DMBs consists typically of the following phases, see Fig. 3.1:

1. *Architecture design.* During this phase, the architecture of the biochip is decided. The architecture can be rectangular (i.e., general-purpose) or non-regular (i.e., application-specific). In Chapter 5, we have proposed a method for the synthesis of an application-specific architecture $\mathcal{A}$ for a biochemical application $\mathcal{G}$ with a deadline $d_{\mathcal{G}}$, considering a maximum number of permanent faults k that have to be tolerated. Since the architecture synthesis is performed before the fabrication (step 2) and testing (step 3), the locations of permanent faults are *not known* during the architecture synthesis.

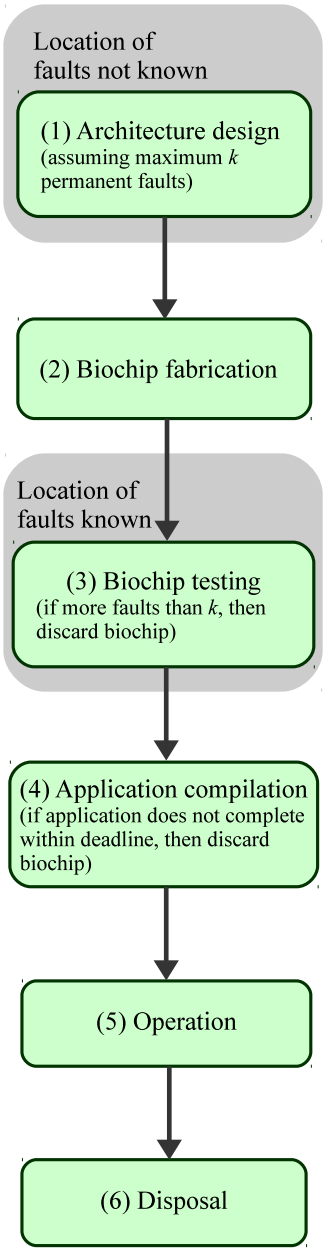


Figure 3.1: Typical phases of a design methodology for DMBs

2. *Fabrication.* During the fabrication phase, the biochip is fabricated based on the architecture design produced in the previous step. A fabrication process may introduce permanent faults in the biochip. We say that an architecture is fault-tolerant to k permanent faults if it can still successfully run the biochemical application within the required deadline.

3. *Testing.* During this phase, all the fabricated biochips are tested to determine if they have permanent faults using testing techniques such as the ones proposed in [88]. We assume that the architecture design phase has synthesized an architecture which is fault-tolerant to maximum k permanent faults. Hence, if, after testing, there are more than k faults, the biochip is discarded. The exact locations of permanent faults, will be known after this phase for each fabricated biochip. Each biochip may have different permanent fault patterns.

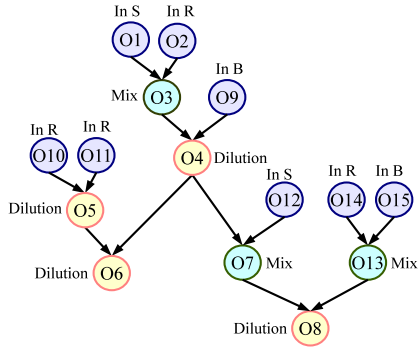
4. *Compilation*. During this phase, we perform a compilation of the application $\mathcal{G}$ on the architecture $\mathcal{A}$ to obtain the electrode actuation sequence. Since the locations of permanent faults are known, we can use permanent fault-aware compilation method, such as the one proposed by [51], to determine the actual completion time $\delta^k_{\mathcal{G}}$ [51]. As introduced earlier, $\delta_{\mathcal{G}}$ represents the application completion time in case no faults are present. We use $\delta^k_{\mathcal{G}}$ to denote the application completion time in case k faults are present and may introduce an overhead on top of $\delta_{\mathcal{G}}$. In case $\delta^k_{\mathcal{G}}$ exceeds the application deadline $d_{\mathcal{G}}$, the biochip is discarded.

5. *Operation*. During the operation phase, the bioassay is run on the biochip. In Chapter 4 we present methods for the operation phase which are able to tolerate transient faults during the application execution.

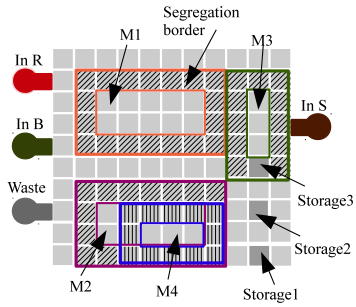
6. *Disposal.* After the biochip has been used, it is properly discarded or stored (in case the results have to archived, e.g., as part of drug screening).

3.1 Compilation of biochemical applications

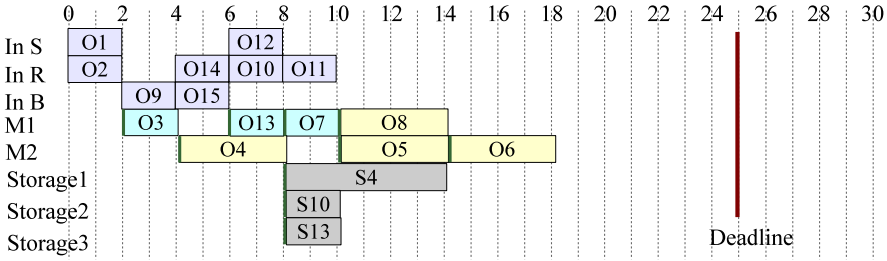
Considering an application $\mathcal{G}$, a biochip architecture $\mathcal{A}$ and a module library $\mathcal{L}$, given as input, the application completion time $\delta_{\mathcal{G}}$ is obtained through *compilation.* Compilation is an optimization problem that can have multiple objectives. Researchers so far have typically performed compilation such that the application completion time is minimized. The compilation task has to determine the following: (i) the *allocation* O, which selects the modules to be used from library $\mathcal{L}$; (ii) the *binding* $\mathcal{B}$ of the selected modules to the operations in the application $\mathcal{G}$; (iii) the *placement* $\mathcal{P}$, which decides the positions of modules on the architecture $\mathcal{A}$; (iv) the schedule $\mathcal{S}$ of the operations; and (v) the *routing* $\mathcal{U}$ of the droplets to the needed locations on the biochip.



a) Application graph $\mathcal{G}^0$



b) Placement of modules on biochip



c) Schedule of operations

Figure 3.2: Example compilation task

Let us illustrate each of these tasks, using the application graph $\mathcal{G}^0$ from Fig. 3.2a, which has a deadline $d_{\mathcal{G}} = 25$ s and has to be executed on the 11×10 biochip from Fig. 3.2b.

3.1.1 Allocation

During the allocation step we decide which modules to use for the execution of the operations. To do that, we need a module library $\mathcal{L}$, which provides for each module the area and the time needed to execute an operation. Most research assumes a given module library that has been previously characterized by designers. The characterization of a module library takes time and it can have a high reagent-cost, since the application has to be executed several times to confirm the results. In addition, in the context of an architecture synthesis, such a characterized module library may not be available

because of the non-regular layout of the application-specific architecture. Hence, in Section 3.2 we propose a method to build a module library for a given architecture.

For our example, we use the characterized module library from Table 2.2. During the allocation phase, the following modules are selected: two 2×5 modules and two 1×3 modules.

3.1.2 Placement of operations

Due to the dynamic reconfiguration feature of the biochip, each of these modules can be placed anywhere on the chip. Modules can physically overlap on-chip, provided that they do not overlap in time, i.e., they are used during different time intervals. If two droplets get too close to each other (e.g., they are situated on adjacent electrodes), then they tend to merge into a single droplet. That is the reason why, when a module is placed on the chip, a segregation border is needed.

Fig. 3.2b shows the placement of the four modules M_{1-4} allocated for our example.

3.1.3 Binding and scheduling

Once the modules have been allocated and placed on the biochip, we have to decide where to execute the operations (binding) and in which order (scheduling). All the compilation methods presented in this thesis extend a List Scheduling heuristic [67] to perform scheduling. List Scheduling has the advantage of producing good quality results in a very short time, hence it is suitable during online compilation and during the architecture evaluation, part of the architecture synthesis. Hence, we briefly present here the main features of List Scheduling, see Fig. 3.3.

ListScheduling takes as input the application graph $\mathcal{G}$, the biochip architecture $\mathcal{A}$ and the module library $\mathcal{L}$ and outputs schedule $\mathcal{S}$ of operations and the application completion time $\delta_{\mathcal{G}}$. Every node from $\mathcal{G}$ is assigned a specific priority according to the *critical path priority function* (line 1 in Fig. 3.3) [67]. The critical path is defined as the longest path in the graph [67], between the root and the leaf nodes. Next, we sort the library (line 2) in ascending order of operation execution time, i.e., the fastest modules are ordered first in the library. *List* contains all operations that are ready to run, sorted by priority (line 4). An operation is ready to be executed when all input droplets have been produced, i.e. all predecessor operations from the application graph $\mathcal{G}$ finished executing. The intermediate droplets that have to wait for the other operations to finish, are stored on the biochip.

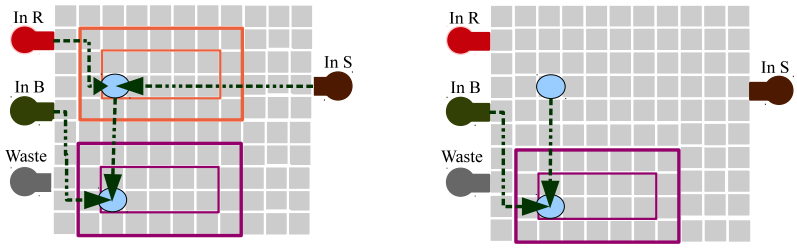
ListScheduling($\mathcal{G}$, $\mathcal{A}$, $\mathcal{L}$)
1: CriticalPath($\mathcal{G}$)
2: SortLibrary($\mathcal{L}$)
3: **repeat**
4: $List$ = GetReadyOperations($\mathcal{G}$)
5: O_i = RemoveOperation($List$)
6: M_j = Place($\mathcal{L}$, $\mathcal{A}$)
7: Bind(M_j, O_i)
8: $route$ = DetermineRoute(O_i, M_j, $\mathcal{A}$)
9: t_i^{start} = Schedule(O_i, $\mathcal{S}$, $route$)
10: t = the earliest time when an operation finishes
11: UpdateReadyList($\mathcal{G}$, t, $List$)
12: **until** $List = \emptyset$
13: **return** $\delta_{\mathcal{G}}$

Figure 3.3: List Scheduling compilation

The algorithm takes each ready operation O_i (line 5) and performs placement, binding, routing and scheduling. Hence, the function **Place** (line 6) returns the first available module $M_j \in \mathcal{L}$ that can be placed on the biochip $\mathcal{A}$. Since the library is ordered by operation execution time, we know M_j is the available module that can execute O_i the fastest. Next, O_i is bound to M_j (line 7), the routing is determined (line 8) and O_i is scheduled (line 9). When a scheduled operation finished executing, *List* is updated with the operations that have become ready (line 11). The repeat loop terminates when the *List* is empty (line 12). The finish time of the last operation in the schedule $\mathcal{S}$ is the application completion time $\delta_{\mathcal{G}}$.

Considering the graph in Fig. 3.2a, the obtained schedule without fault-tolerance is presented in Fig. 3.2c. The schedule is depicted as a Gantt chart, where for each module, we represent the operations as rectangles with their length corresponding to the duration of that operation on the module. The allocation and binding of operations to devices are shown in the Gantt chart as labels at the beginning of each row of operations. The thick vertical lines in Fig. 3.2c represent the routing times. For example, operation O_3 is bound to module M_1 and starts immediately after operation O_2 (t_3^{start} = 2.09) and takes 2 s, finishing at time t_3^{finish} = 4.09.

Depending on the application, dispensing operations can take up to 7 s [76], so it is important to schedule them intelligently. Our heuristics is the following: A dispensing operation, such as O_1 in Fig. 3.2a, has no predecessor operations, therefore, if the corresponding reservoir is available, it can be scheduled at time $t = 0$. However, until they can be used, the dispensed droplets have to be stored on the biochip, occupying areas that can be used for other operations. To avoid this situation, we schedule the dispensing operations only when the dispensed droplets are needed.



a) Routing at t = 2 b) Routing at t = 4

Figure 3.4: Routing example

At time $t = 2$ s mixing operation O_3 has the highest priority among all the ready operations (an operation is ready if all its input droplets have arrived). Module M_1 is the fastest available (i.e., not occupied by other operations) module, hence O_3 is bound to M_1. After that, we determine the routes to bring the input droplets of operation O_3 to module M_1. The routing time is much faster in comparison to the execution times of the operations, hence we represent it with thick green lines in Fig. 3.2a. The next subsection presents examples of routing. However, for simplicity reasons, we have ignored routing in all the other examples in this thesis, but we take routing into account in our algorithms. At time $t = 4$ s, operation O_3 finishes executing, and *List* is updated with its successor, operation O_4, which becomes ready to execute. The total schedule length is 18.32 s.

3.1.4 Routing

In order to start executing an operation, we need to route the droplets, i.e., to bring the input droplets to the location of the operation. As mentioned in the previous section, we perform routing inside the **ListScheduling** algorithm. For our example, the droplet routes determined at $t = 2$ and $t = 4$ are shown in Fig. 3.4a, b. For this example, we consider that the operations start and finish executing on the bottom left electrode of the module. As shown in Fig. 3.2c, at time $t = 2$ operation O_3 is scheduled to start on module M_1. The input droplets are brought from dispensing reservoirs *In R* and *In S*, as shown in Fig. 3.4a. At $t = 4.09$, operation O_4 is scheduled to execute on module M_2 and thus, the output droplet of O_3 and the dispensed droplet from *In B* are routed as shown in Fig. 3.4b. As mentioned in Section 2.2, we assume that routing a droplet between two adjacent electrodes takes 0.01 s [60].

3.2 Building a library of circular-route modules

As mentioned, the compilation needs to use a module library $\mathcal{L}$ to decide what modules to allocate and bind to the operations. For regular architectures, researchers have characterized and used libraries such as the one in Table 2.2. However, such a library, which contains rectangular modules, cannot be used for non-regular application-specific architectures (see Fig. 2.1 for an example application-specific architecture). For such architectures, we assume that we first build a library $\mathcal{L}$ of Circular-Route Modules (CRMs) (see Section 2.2.1).

We want to determine CRMs that will use effectively the area on $\mathcal{A}$, so that the application completion time is minimized. As discussed in Section 2.2, mixing of two droplets is achieved faster when the forward movement of the droplets is increased and the backward movement is avoided [58]. An application-specific architecture can have a non-regular shape, so we need to find those locations on the biochip where the operations can be executed faster.

In Fig. 3.5 we present our proposed algorithm **BuildLibrary**, which returns a library of CRMs for a given application-specific architecture. **BuildLibrary** starts by identifying restricted rectangles (RRs) (line 1), which are areas of rectangular shape bordered by the margins of the architecture. We use the RRs as guiding areas for obtaining CRMs. Then, for each RR found, we call **DetermineCRM** to determine a list of circular route modules $\mathcal{L}_{CRM}$ (line 3), which is stored in the library $\mathcal{L}$.

We use the cutting algorithm from [85], developed for paper cutting problems, where the material needs to be optimally cut so that it minimizes waste. The list of restricted rectangles L_{RR} (line 1) is obtained by using "guillotine" cuts, done parallel with the edges of the architecture. We start the cuts from each corner-electrodes of the architecture, using horizontal and vertical cuts. A corner-electrode is an electrode that at least two edges which are not bordered by any other electrode. To obtain R_{XY} we cut horizontally and vertically, and then, changing the order, we use first vertical and then horizontal cuts to obtain R_{YX}, like the RRs depicted in Fig. 3.6 obtained for the bottom

BuildLibrary($\mathcal{A}, MinR, MaxR, MinW, MaxW$)
1: L_{RR} = DetermineRestrictedRectangles($\mathcal{A}$)
2: **for each** RR_i in L_{RR} **do**
3: L_{CRM} = DetermineCRM($\mathcal{A}, RR_i, MinR, MaxR, MinW, MaxW$)
4: InsertInLibrary(L_{CRM}, $\mathcal{L}$)
5: **end for**
6: **return** $\mathcal{L}$

Figure 3.5: Algorithm for building a CRM library

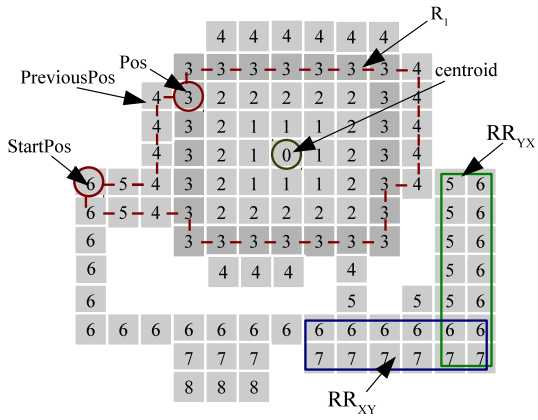


Figure 3.6: Determining circular-route modules

right corner of the architecture. In some of the cases, R_{XY} is the same rectangle as R_{YX}. Also, we consider those unused areas containing inactive electrodes and we extend L_{RR} with restricted rectangles of such inactive electrodes. In this case, the restricted rectangles will be bordered by active electrodes.

As an input to the **DetermineCRM** function (presented in Fig. 3.7 and discussed in Section 3.2.1), we use the control parameters *MinR, MaxR, MinW, MaxW*, which are experimentally determined for a given application-specific architecture. The CRMs are stored in the library $\mathcal{L}$ and used during the compilation.

3.2.1 Determining a circular-route module

For each restricted rectangle (RR), we determine a list of CRMs L_{CRM} using **DetermineCRM**, illustrated in Fig. 3.7. We start from the centroid (geometric center) of the RR and "graphic fill" the architecture, considering each electrode a pixel (line 2). The centroid of a rectangle is situated at the intersection of its diagonals. Fig. 3.6 shows a filled architecture starting from the centroid of the restricted rectangle R_1. We use a greedy approach to find CRMs that fulfill the distance constraints set by control parameters *MinR, MaxR, MinW, MaxW* (line 8). *MinR* and *MaxR* bound the Radius, which sets the distance from the centroid and it is used to determine the start position of the CRM. *MinW* and *MaxW* set the boundaries for the next electrodes of the CRM, which can be situated at a distance from the center that variates between [$Radius - Window$, $Radius$], where *Window* can have any value in the range [*MinW*, *MaxW*].

After the architecture is filled, the list of start-electrodes L_{SP} for the CRM is determined (line 3), by selecting all start-electrodes located at a distance equal with *Radius* from the centroid of the considered RR. For example, for the restricted rectangle R_1 from Fig. 3.6, and a $Radius = 6$, the list of starting start-electrodes L_{SP} contains all electrodes marked with 6. From each of the start-electrodes we construct a route (line 6–11), by adding new electrodes until the route completes in a circle, i.e., it reaches the start-electrodes.

DetermineCRM($\mathcal{A}, RR, MinR, MaxR, MinW, MaxW$)

```
1:  L_CRM = List of circular route modules
2:  FillArch(A, RR)
3:  L_SP = GetStartPosition(A, RR, Radius)
4:  for each StartPos in L_SP do
5:    for Radius from MinR to MaxR do
6:      for Window from MinW to MaxW do
7:        InsertInRoute(CRM, StartPos)
8:        repeat
9:          NextPos = GetBestNeighbor(CRM, Radius, Window)
10:         InsertInRoute(CRM, NextPos)
11:       until NextPos is StartPos
12:       UpdateList(L_CRM, CRM)
13:     end for
14:   end for
15: end for
16: return L_CRM
```

Figure 3.7: Determine CRM algorithm

GetBestNeighbor (line 8) uses a greedy randomized [21] approach to select the next electrode of the route. Out of the possible next electrodes, which are those that can be reached from the current position, **GetBestNeighbor** selects from the neighbors that are located within the boundaries imposed by control parameters, the one that leads to the largest operation completion percentage (see Section 2.2.1). In case there are two equally good candidates for the next position, **GetBestNeighbor** randomly selects one of them. Backward moves are also permitted in case there are no other options.

Let us consider the example in Fig. 3.6, where the start-electrode is labeled *StartPos* and the current position is labeled *Pos*. We also consider the given control parameters $Radius = 6$, $MinW = 1$ and $MaxW = 3$. For such a window of size 3, out of the four neighboring electrodes that can be reached from *Pos*, only three (the left, up and down ones) fulfill the distance requirement, which is to be at a distance between 6 and 3 from the centroid of R_1. In our example, selecting the top or bottom electrode improves the mixing with 0.1%, while the left electrode with -0.5%, due to flow reversibility. Since the top and the bottom-electrodes are equally good candidates, we randomly select the

top electrode to be the next position. The search continues, adding at each step a new electrode to the CRM, until the start-electrode, labeled *StartPos* in Fig. 3.6, is reached. We obtain the CRM marked with a red interrupted line in Fig. 3.6.

It is difficult to predict at the pre-compilation stage which CRM should be selected, to reduce the completion time of the application, as it depends not only on the architecture, but also on application's particularities such as dependencies between operations and contamination constraints. Hence, each of the determined CRMs is evaluated, and only three are stored in L_{CRM} (line 12): the one that minimizes the use of area, the one that minimizes operation completion time; a third CRM, represented by the corresponding RR is also stored in L_{CRM}. **DetermineCRM** returns the list of CRMs L_{CRM} (line 16).

Note that, if a general-purpose architecture is given as input instead of a non-regular architecture, our **BuildLibrary** algorithm will determine only modules of rectangular shape. Hence, our algorithm is general and can be used for both general-purpose architectures and application-specific architectures.

CHAPTER 4

Compilation for Error Recovery

In this chapter we are interested in fault-tolerance against transient faults, which occur during the execution of the application, see Section 2.1.1 for a discussion on transient faults and how they can be detected. Biochemical applications are executed based on the electrode actuation sequence, which is produced in the compilation task, see Section 3.1. When a transient error is detected during the execution, the original electrode actuation sequence has to be interrupted, and recovery actions have to be initiated to remedy the error. These recovery actions are a sequence of operations that have to be executed, and will have to be compiled also into an electrode actuation sequence.

There are two main approaches for recovery, depending on when the compilation of the recovery actions is performed: *offline*, at design time, or *online*, at runtime. In the offline approach, all possible fault scenarios are identified, and a compilation is performed for each corresponding sequence of recovery operations. These will form alternative schedules, which are stored into a database. During the execution of the application, when an error is detected, the corresponding schedule is selected from the database and applied to recover from the error. This approach has the disadvantage of a state explosion in case there are too many fault scenarios.

Because biochemical application execution is several orders of magnitude slower than instructions executing on a microprocessor (see Table 4.1 for typical operation exe-

cution times), we have the opportunity to decide the appropriate recovery actions and compile them into an electrode actuation sequence at runtime. This has the advantage of taking into account the actual fault-occurrences detected at runtime, which allows for a more appropriate response, i.e., minimizing the recovery time and thus potentially tolerating more transient faults.

Section 4.1 presents an offline approach to tolerating s transient faults that affect split operations, whereas Section 4.2 presents an online recovery technique which can tolerate transient faults in all types of operations. The focus of these two sections is on the compilation techniques required for producing the recovery actions.

4.1 Offline compilation for error recovery

In this section, we focus on the erroneous volume variation after an unbalanced split operation. We propose a Fault-Tolerant Compilation (**FTC**) method that derives all the backup static schedules needed to recover from all combinations of faulty split operations.

A split operation is performed by turning on simultaneously the control electrodes to the right and left of the droplet. However, due to the misalignment between the droplet and the control electrode or because of the breakdown of electrode dielectric [70], the resulting droplet volumes after a split operation might be unbalanced, see Fig. 2.5. Recovery from faulty split operations is done by merging the droplets back and re-executing the split operation. The error recovery actions are determined *offline*, and are applied online when an error is detected. Hence, at runtime, the microcontroller will switch to the backup schedules corresponding to the observed error occurrences.

Our approach for the proposed **FTC** is based on the following assumptions:

- The biochemical application is executed on a general-purpose biochip with a rectangular architecture, such as the one in Fig. 4.1c.
- The operations execute on electrodes grouped in rectangular areas, called "modules". We assume that the designers have characterized a module library $\mathcal{L}$, which contains the execution time and area needed for an operation to complete, similar to the one in Table 4.1.
- We consider that a split operation is faulty if it results in droplets with volume variation from the expected volume, below a given threshold. The threshold is given by the designer and depends on the application. If an error is detected (the volume variation is below or above the given threshold), the resulted droplets

are *merged* back. They have to be routed to the same place on the chip, and the merging is instantaneous. The split operation will have to be performed again.

- We model the biochemical application using the proposed Fault-Tolerant Sequencing Graph (FTSG), presented in Section 2.4.1.

4.1.1 Problem formulation

In this section we address the following problem. As input we have a biochemical application modeled as a graph $\mathcal{G}$ with a deadline $d_{\mathcal{G}}$ and a rectangular biochip architecture $\mathcal{A}$. The fault model is given by the parameter s which denotes the maximum number of transient faults that can occur during split operations. The designer provides a characterized module library $\mathcal{L}$ and specifies the maximum number n_{sns} of volume sensors that can be used. We are interested in compiling a fault-tolerant implementation Ψ such that the worst-case application completion time $\delta^s_{\mathcal{G}}$ is minimized and the deadline $d_{\mathcal{G}}$ is satisfied. The worst-case application completion time $\delta^s_{\mathcal{G}}$ is defined as the longest execution time of $\mathcal{G}$ over all possible faults scenarios.

Hence, we have to decide on: the allocation $\mathcal{O}$, which determines what modules from library $\mathcal{L}$ are to be used; the binding $\mathcal{B}$ of each operation to a module $M_i \in \mathcal{L}$; the placement $\mathcal{P}$ of the modules and of the sensors on the architecture $\mathcal{A}$; the fault tolerant schedule $\mathcal{S}$ of the application, which contains the start time of each operation on its corresponding module and the routing $\mathcal{U}$ of the droplets to the needed locations on the biochip.

Let us consider the application graph $\mathcal{G}$ from Fig. 4.1a which is performed on an 8×8 biochip with three sample-reservoirs, two buffer-reservoirs and one reagent-reservoir, using the module library $\mathcal{L}$ provided in Table 4.1. The deadline of the application is $d_{\mathcal{G}} = 25s$. The input operations are already assigned to the corresponding input reservoirs. During the allocation task, specific modules are selected from $\mathcal{L}$ and placed on the 8×8 chip, such that the application completion time is minimized. For this example, the following modules are used: one 1×3 mixer, two 2×5 mixers and one 2×4 mixer, see Fig. 4.1c–f. The obtained schedule without fault-tolerance is presented in Fig. 4.1b. As shown in the schedule, the biochemical application completes in 8 s, satisfying the timing constraints. Note that we have ignored the dispensing operations in this example.

However, the presented schedule does not take in account the possibility of fault occurrence during a split operation. Let us consider a maximum number s of faults that can occur during the application execution. The faults are detected using sensors, which have a fixed placement. For the application in Fig. 4.1a, we use one sensor, placed as

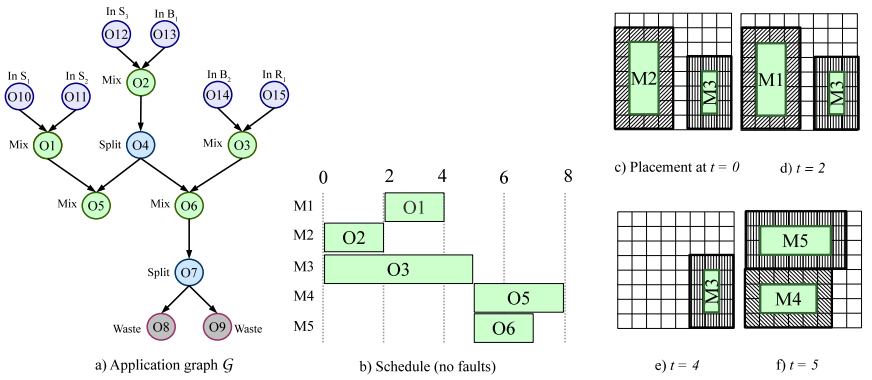


Figure 4.1: Compilation results (no faults)

in Fig. 4.2a–d, where it occupies 1 cell (3×3 with protection borders) at the top right corner of the chip.

The straightforward way to adapt the schedule from Fig. 4.1b is to introduce after each split operation enough *slack* (idle time) that allows the application to fully recover in case of faults. The fault-tolerance is achieved through error detection (detection) and recovery (merging back the droplets, followed again by a split). Considering the worst-case, in which all s faults happen in the same split operation, the required slack time is calculated as:

$$t_{slack} = s \times (t_{detection} + t_{merge} + t_{split}). \tag{4.1}$$

We assume that merge and split operations are instantaneous and we use a detection time of 5 s, see Table. 4.1. Thus, for $s = 2$, the slack required for recovering the split operation O_4 is 2×5 = 10 s, as depicted in Fig. 4.2e, with a rectangle labeled "O4 slack". A similar slack is introduced for O_7, thus obtaining the fault-tolerant schedule from Fig. 4.2e, with a worst-case application completion time of 24 s. We call such a fault tolerant strategy StraightForward Scheduling (**SFS**). Although the timing constraints of the application are satisfied, the schedule obtained by using **SFS** wastes a lot of unnecessary time for recovery. For example, for the schedule in Fig. 4.2e, if both faults happened during the split operation O_4, then the maximum number of faults ($s = 2$) is reached, and hence there is not need in allocating slack time after split operation O_7.

Our proposed **FTC** uses an improved fault-tolerant scheduling technique, which can take into account the actual fault-occurrence pattern during the execution. By taking into account fault-occurrence information, **FTC** produces shorter schedules, leading to a reduced worst-case application completion time δ^s_G. **FTC** relies on the FTSG, proposed in Section 2.4.1, which captures all the possible fault-scenarios. The FTSG from Fig. 4.6 is build starting from the application graph from Fig. 4.1a and captures

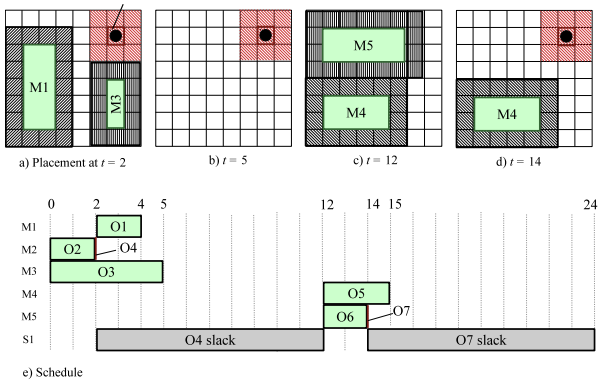


Figure 4.2: SFS schedule

all alternative scenarios for $s = 2$. Starting from the FTSG $\mathcal{G}^S$ our **FTC** algorithm generates a table $\mathcal{S}$ where, for each operation, we have the activation condition (the particular combination of faults) and the corresponding start time. For example, the merge operation O_{20} will be activated at time $t = 7$ if a fault has occurred in the split operation $O_{4.1}$ (see Fig. 4.3e).

During runtime, depending on the detected fault occurrences, a microcontroller will activate the corresponding operations. For example, for the fault scenario captured by the shaded subgraph in Fig. 4.6 (first fault in O_4 and the second in O_7), the operations in Fig. 4.3e will be activated at the depicted start times. For the case when two faults happen in O_7 we have the start times depicted in Fig. 4.4e. The worst-case application completion time is 19 s for **FTC**, compared to 24 s for **SFS**. The difference between **FTC** and **SFS** results from the detection operation time: unnecessary detection operations are avoided by **FTC**. We have considered that a detection operation takes 5 s. However, there are capacitance sensor implementations that can detect a droplet volume in shorter time [59]. In this case, **SFS** is preferable over **FTC** due to its simplicity.

Table 4.1: Module library $\mathcal{L}$ for FTC

Operation	Module area	Operation time (s)
Mix	2×5	2
Mix	2×4	3
Mix	1×3	5
Mix	3×3	7
Mix	2×2	10
Sensing	1×1	5

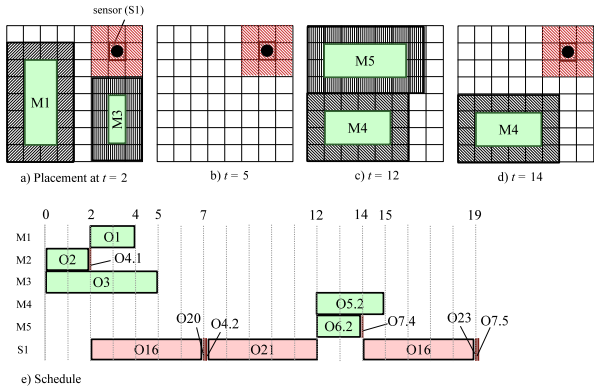


Figure 4.3: FTC schedule for faults in O_4 and O_7

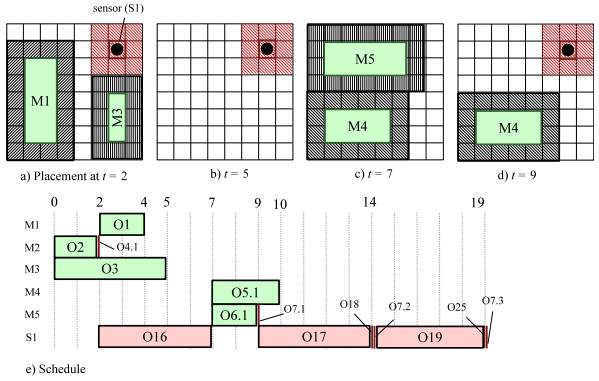


Figure 4.4: FTC schedule for faults in O_7

4.1.2 Fault-tolerant compilation

Our proposed **FTC**, outlined in Fig. 4.5 and has three steps:

1. In the first step, we use the compilation algorithm from [49], called by **DMB-Compilation** function (line 1 in Fig. 4.5), to obtain the allocation $\mathcal{O}^0$, binding $\mathcal{B}^0$ and placement $\mathcal{P}^0$ that minimizes the application completion time without considering faults. We have extended the compilation from [49] to decide the number of sensors and their placement given the maximum number of sensors n_{sns} that are available.

2. In the second step, we build a FTSG $\mathcal{G}^S$ starting from the application graph $\mathcal{G}$ (line 2 in Fig. 4.5) that captures all fault scenarios for a given s maximum number of faults. The FTSG graph $\mathcal{G}$ is generated by the function **GenerateFTSG** which takes as parameters the application graph $\mathcal{G}$ and the maximum number of faults s. For the application graph in Fig. 4.1a, considering $s = 2$, we obtain the FTSG from Fig. 4.6.

3. In the third step (line 3 in Fig. 4.5), we obtain a fault-tolerant schedule table $\mathcal{S}$ using the **FTScheduling** algorithm from Fig. 4.7.

As mentioned, the fault-tolerant schedule table $\mathcal{S}$ is obtained by the **FTScheduling** algorithm from Fig. 4.7, which takes as input the FTSG graph $\mathcal{G}^S$ generated in step 2, the biochip architecture $\mathcal{A}$, the binding $\mathcal{B}^0$ and placement $\mathcal{P}^0$ obtained in step 1, and the module library $\mathcal{L}$. We start by generating all the fault scenarios *FaultScenList* (line 1 in Fig. 4.7) considering s maximum faults. Let us consider as example the graph $\mathcal{G}$ from Fig. 4.1a, which has two split operations: O_4 and O_7.

Then, we traverse the FTSG and extract all subgraphs corresponding to each possible scenario $F_i \in FaultScenList$. We use the Breadth-First Search (BFS) algorithm to traverse $\mathcal{G}$ (line 10) and for each split operation encountered we remove the branch that does not correspond to the current scenario F_i. In Fig. 4.6, the scenario $\{O_4, O_7\}$ corresponds to the case when the first fault happens during O_4, so when we evaluate the split operation $O_{4.1}$, we remove the non-faulty branch, starting with the edges $D_{16} \rightarrow O_{5.1}$ and $D_{16} \rightarrow O_{6.1}$. The process continues until all split operations are evaluated. Eventually, for $\{O_4, O_7\}$ we obtain the shaded subgraph in Fig. 4.6.

After extracting the scenario subgraphs, we schedule each of them (Fig. 4.7, line 13) by using the List Scheduling (LS) algorithm from Fig. 3.3. We have adapted the **ListScheduling** algorithm, explained in detail in Section 3.1.3, to use the binding $\mathcal{B}^0$ and the placement $\mathcal{P}^0$ determined at step 1, see Fig. 4.5. For the considered fault scenario O_4, O_7, **ListScheduling** outputs the schedule table from Fig. 4.3e.

FTC($\mathcal{G}$, $\mathcal{A}$, $\mathcal{L}$, s)
1: Ψ^0 = DMBCompilation($\mathcal{G}$, $\mathcal{A}$, $\mathcal{L}$)
2: $\mathcal{G}^S$ = GenerateFTSG($\mathcal{G}$, s)
3: $\mathcal{S}$ = FTScheduling($\mathcal{G}^S$, $\mathcal{A}$, $\mathcal{B}^0$, $\mathcal{P}^0$, s)
4: **return** $\Psi = < O^0, \mathcal{B}^0, \mathcal{S}, \mathcal{P}^0 >$

Figure 4.5: Fault-tolerant compilation example

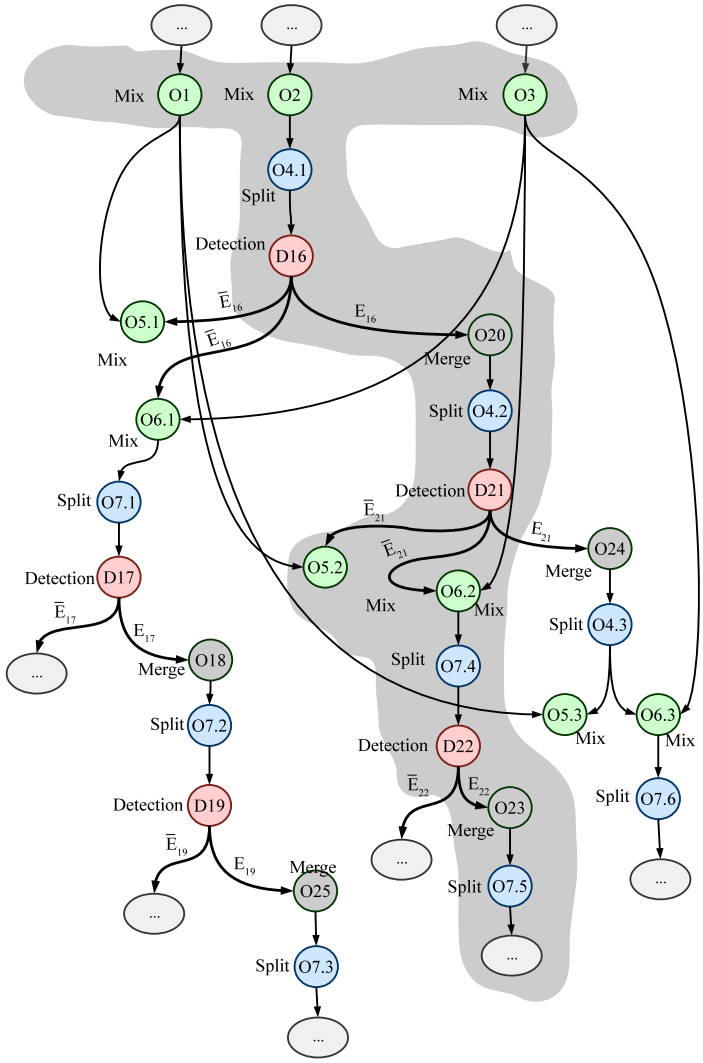


Figure 4.6: FTSG $\mathcal{G}^S$ for application in Fig. 4.1a

FTScheduling($\mathcal{G}$, $\mathcal{A}$, $\mathcal{B}$, $\mathcal{P}$, $\mathcal{L}$)

```
1: FaultScenList = GenerateFaultScenarios(G)
2: S = ∅
3: for each F_i ∈ F do do
4:    G' = G
5:    O_i = source
6:    while O_i ≠ ∅ do
7:       if O_i is split operation then then
8:          RemoveBranch(G', O_i, F_i)
9:       end if
10:      O_i = BFS(G', O_i)
11:   end while
12:   Graph = G'
13:   S = ListScheduling(Graph, A, L)
14: end for
15: return S
```

Figure 4.7: Fault-tolerant scheduling algorithm

4.2 Online compilation for error recovery

We propose and online error recovery approach which uses for error recovery a combination of time and space redundancy techniques, which are presented in Section 2.4.2.2. In order to decide for the appropriate recovery technique we propose a redundancy optimization strategy (**ROS**), presented in Section 4.2.2. Our online error recovery addresses the volume variations in all types of operations, not only split operation, and it is based on the following assumptions:

- The biochemical application is executed on a general-purpose biochip with a rectangular architecture, such as the one in Fig. 4.8c.
- The operations execute on electrodes grouped in rectangular areas, called "modules". We assume that the designers have characterized a module library $\mathcal{L}$, which contains the execution time and area needed for an operation to complete.
- We model the biochemical application using the proposed redundancy recovery graph, presented in Section 2.4.2.2.
- We consider both capacitive sensors and a Charged-Coupled Device (CCD) camera-detection system for determining the volume of a droplet, which is then compared to its expected volume in order to perform error detection.

4.2.1 Problem formulation

In this section we address the following problem. As input we have a biochemical application modeled as a graph $\mathcal{G}^0$ with a deadline $d_{\mathcal{G}}$, which is executed the architecture $\mathcal{A}$. A characterized library $\mathcal{L}$, containing the area and execution time for each operation (similar to Table 2.2), is also given as input. We are interested to determine online the necessary recovery actions, so that the number of transient faults tolerated is maximized and the application deadline $d_{\mathcal{G}}$ is satisfied.

As mentioned, we consider both time redundancy and space redundancy when deciding what fault-tolerant policy to use for each detection operation. We decide online where to introduce detection operations and which redundancy technique to use.

The advantage of using space redundancy is faster recovery time in case of error, at the cost of extra overhead in completion time, in case of no error. When time redundancy is used, the recovery actions are executed only after an error is detected, so no extra time overhead is added in case of no error. However, in case of error, the recovery is slower for time redundancy than space redundancy. Since the error scenarios are not known in advance, an *online* redundancy optimization strategy can better exploit the current configuration, leading to improved results.

Let us illustrate this by using the application graph $\mathcal{G}^0$ from Fig. 4.8a, which has a deadline $d_{\mathcal{G}} = 25$ s and has to be executed on the 10×8 biochip from Fig. 4.8c. We used for this example the module library from Table 2.2. In Fig. 4.9a we show the schedule of the application for the case when we do not consider the issue of fault-tolerance (and there are no errors). The schedule of operations is presented as a Gantt chart, where the start time of an operation is captured by the left edge of the respective rectangle, and the length of the rectangle represents the duration. As shown in Fig. 4.9a, operation O_1 starts executing at $t = 0$ *s* and finishes at $t = 2$ *s*. The completion time of $\mathcal{G}^0$ is $\delta_{\mathcal{G}^0} = 18$ s. Such a schedule has a one-to-one correspondence to the electrode actuation sequence, used by the control software on the computer the run the biochemical application on the biochip. As mentioned in Chapter 3, an implementation consists of allocation, binding, placement, scheduling and routing. The allocation and binding of operations to devices are shown in the Gantt chart as labels at the beginning of each row of operations. For example, the non-reconfigurable operation O_1 is bound to the dispensing reservoir *In S*, while the mixing operation O_3 is bound to $Mixer_1$, for which we have allocated a 2×5 module. The placement of modules, for all the examples in this section, is presented in Fig. 4.8c.

In this example, we are interested to tolerate two transient errors, affecting the volume of droplet. Two detection operations D_6 and D_8 are inserted in $\mathcal{G}^0$, obtaining the graph $\mathcal{G}^+$ from Fig. 4.8b. For the considered example, we have four possible error scenarios in the case of maximum two transient faults: (1) when no error is detected, (2) when

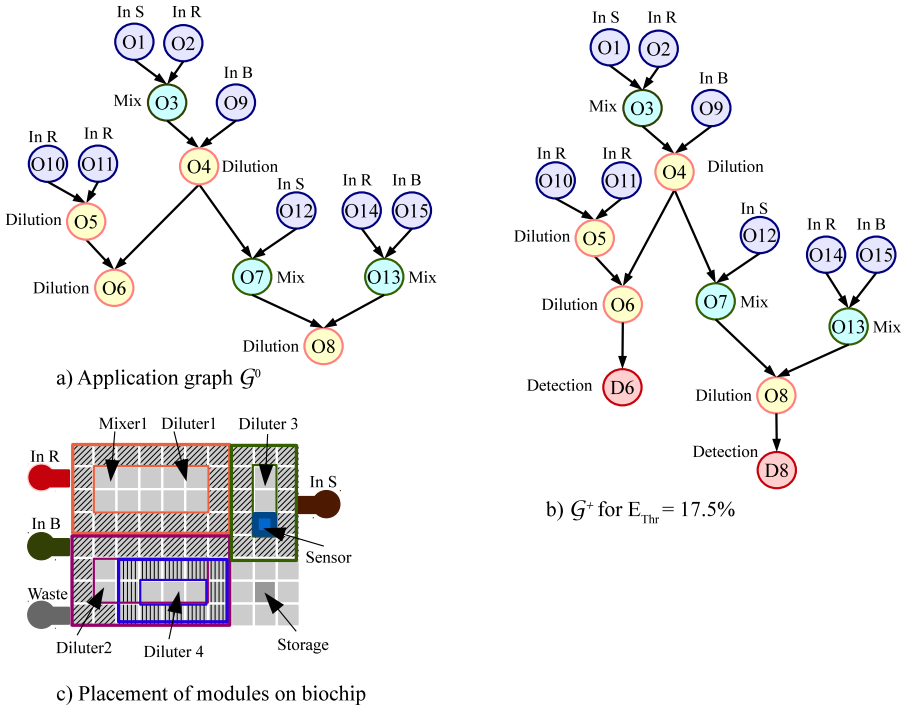


Figure 4.8: Motivational example

a single transient error is detected by D_6, (3) when a single transient error is detected by D_8 and (4) when two transient errors are detected by both D_6 and D_8. These error scenarios are presented in the rows of Table 4.2. For this example we assume no errors during recovery. However, our approach also takes into account errors during the recovery operations.

There are several possible redundancy solutions to tolerate the transient faults in each scenario. We are interested to decide on an assignment of redundancy to the application, such that the deadline of 25 s is satisfied in every fault scenario (the application is fault-tolerant only if it completes within its deadline in all the possible fault scenarios). There are four possible solutions for our example: (a) using only time redundancy, (b) using only space redundancy, (c) using time redundancy for tolerating the error detected by D_6 and space redundancy for tolerating the error detected by D_8 and (d) using time redundancy for D_8 and space redundancy for D_6.

The time and space redundancy subgraphs are added to $\mathcal{G}^+$ in Fig. 4.8b as discussed in Section 2.4.2.2. Columns 3 to 6 in Table 4.2 present the best results in terms of the application completion time $\delta_{\mathcal{G}}$ obtained using each redundancy scheme (a)–(d) for

Table 4.2: Application completion times (for combinations of error scenarios and redundancy scenarios)

Scenario	Detected by	Fault-tolerance solutions (a) Only Time redundancy	(b) Only Space redundancy	(c) Time in D_6 Space in D_8	(d) Time in D_8 Space in D_6
1	—	18	18	18	18
2	$\{D_6\}$	(36)	(26)	(30)	22
3	$\{D_8\}$	(28)	24	23	23
4	$\{D_6, D_8\}$	(46)	(36)	(30)	25

each error scenario (1)–(4). The completion times δ_G that miss the deadline of 25 s are showed in parenthesis. In these situations, we consider that the application was not able to tolerate the transient faults.

As we see from Table 4.2, the only situation when the application is able to recover in all error scenarios and complete before the deadline is solution (d) when time redundancy is used in D_8 and space redundancy is used in D_6. The schedule length is 25 s, satisfying the deadline. In Fig. 4.9c we show the schedule for two errors, one in D_6 and one in D_8, error scenario (4), in case (d). If we use only time redundancy, as in case (a), we miss the deadline in error scenarios (2)–(4). The schedule for the case (a) for error scenario (3) is presented in Fig. 4.9b, and has a length of 28 s, which means that the deadline is missed when only time redundancy is used and the error is detected by D_8. The schedule depicts the detection operations as thick lines labeled with the operation name. For this example we consider that detections happen in zero time. However, in our implementation, the time needed for the detection operation is calculated as the routing time to bring the droplet to the sensor plus the detection time. We also consider the waiting time in case the sensor is busy with another detection operation.

If we use only space redundancy, as in case (b), we miss the deadline in error scenarios (2) and (4). The biochip used in this example has an area of 10×8 electrodes. However, if we use an area of 10×11 electrodes, and also add one extra reservoirs for reagent to the biochip architecture, to parallelize the dispensing, we obtain an application completion time within the required deadline for all error scenarios by using space redundancy only. Our online recovery approach takes into account the available resources when optimizing the redundancy. For a 10×8 architecture, using only space redundancy is not a good option.

In solution (c), using time redundancy for D_6 and space redundancy for D_8 also turned out to be a bad decision, since we miss the deadline in the error scenarios (2) and (4).

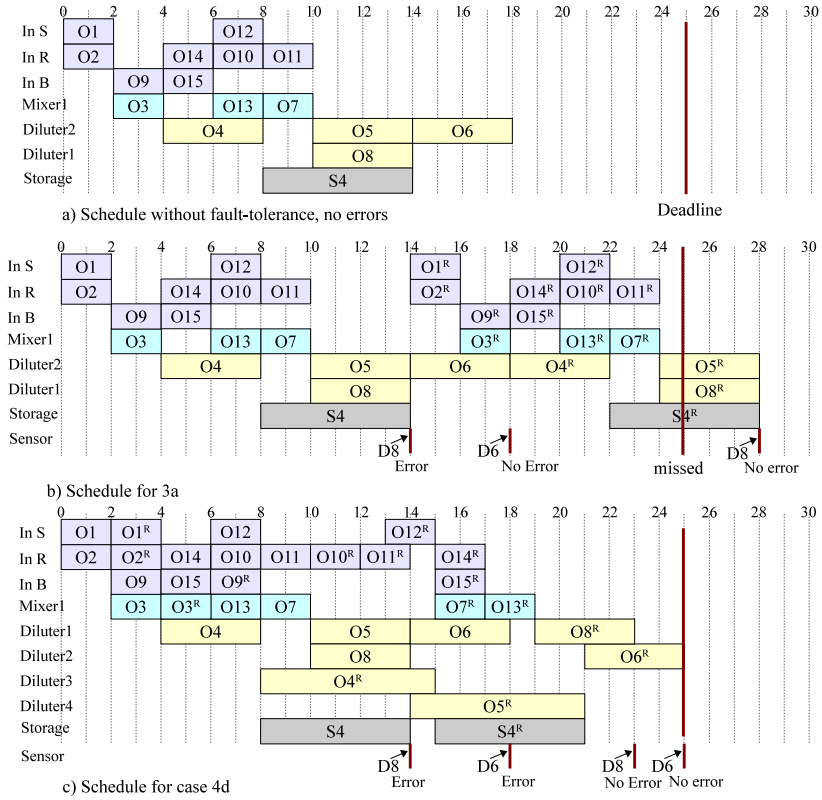


Figure 4.9: Schedules for various error scenarios

This motivational example shows that (i) using a single fault-tolerance technique is not a good decision and that (ii) we need to find the right combination of time and space redundancy to tolerate the faults in all possible error scenarios, and that (iii) the right decisions depend also on the application and architecture. Our redundancy optimization approach will decide online the introduction of the right combination of fault-tolerance, such that the number of transient faults tolerated is maximized and the application deadline is satisfied.

4.2.2 Online error recovery strategy

Fig. 4.10a presents the general strategy of our online recovery approach for the case when a sensor-based detection is used. We discuss the case when a CCD camera-based detection is used in Section 4.2.5.

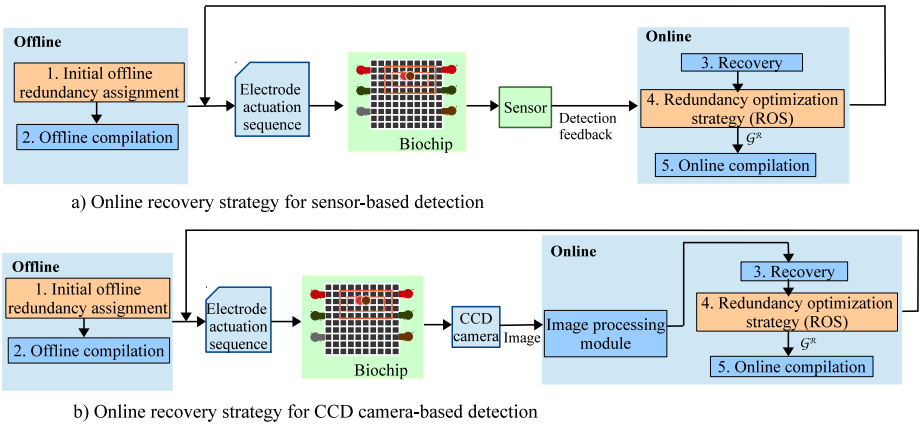


Figure 4.10: Online error recovery strategy

Our strategy has two components: an offline component consisting of steps 1 and 2, performed at design time, and an online component, steps 3–5, invoked during the execution of the biochemical application. The steps presented in light blue rectangles (see Fig. 4.10a) are executed on a computer or a microcontroller, whereas the ones in green rectangles are performed on the biochip. Steps 1 and 2 produce offline a fault-tolerant implementation without performing redundancy optimizations that are possible once the error scenarios are known at runtime. Step 1 decides an initial redundancy assignment and the produced fault-tolerant graph is then compiled during step 2. The initial offline redundancy assignment from step 1 can be decided manually by the designer (as we do in [4]) or by any other method. The same way, step 2 can be implemented using any available compilation such as the Tabu-Search [50] or Simulated Annealing-based [14] implementations. In the experimental results we have used our **ROS**, from Section 4.2.4 considering a no-faults scenario, to produce the initial redundancy assignment for step 1, and the compilation from [4] for step 2.

The offline compilation results are executed on the biochip until a detection operation finishes, when the bioassay execution is stopped and the online component is invoked. If an error is detected by the detection operation D_i, we use step 3 to recover. As described in Section 2.4.2.2, if time redundancy has been previously assigned to D_i, we run the corresponding subgraph R_i to recover from the error detected by D_i. If space redundancy has been assigned to D_i, we use for recovery the redundant droplets produced by R_i^{Space}. Next, at step 4, we run our **ROS**, which optimizes the introduction of detection points and associated redundancy (see Section 4.2.4).

ROS uses the available information about the current error scenario to optimize the assignment of time and space redundancy for fault-tolerance. Hence, **ROS** is invoked only when new information about the occurrences of errors is available, that is, after

the detection operations (see the arrow labeled "Detection feedback" in Fig. 4.10a). The fault-tolerant graph $\mathcal{G}^R$, outputted by **ROS**, is compiled during step 5, determining a new electrode actuation sequence to be executed on the biochip. The compilation implementation for step 5 has to be fast, as it is run online and will add an overhead to the execution of the bioassay. Hence, for step 5 we use a LS-based online compilation (see Section 3.1.3) as it is able to obtain good quality results in a short time.

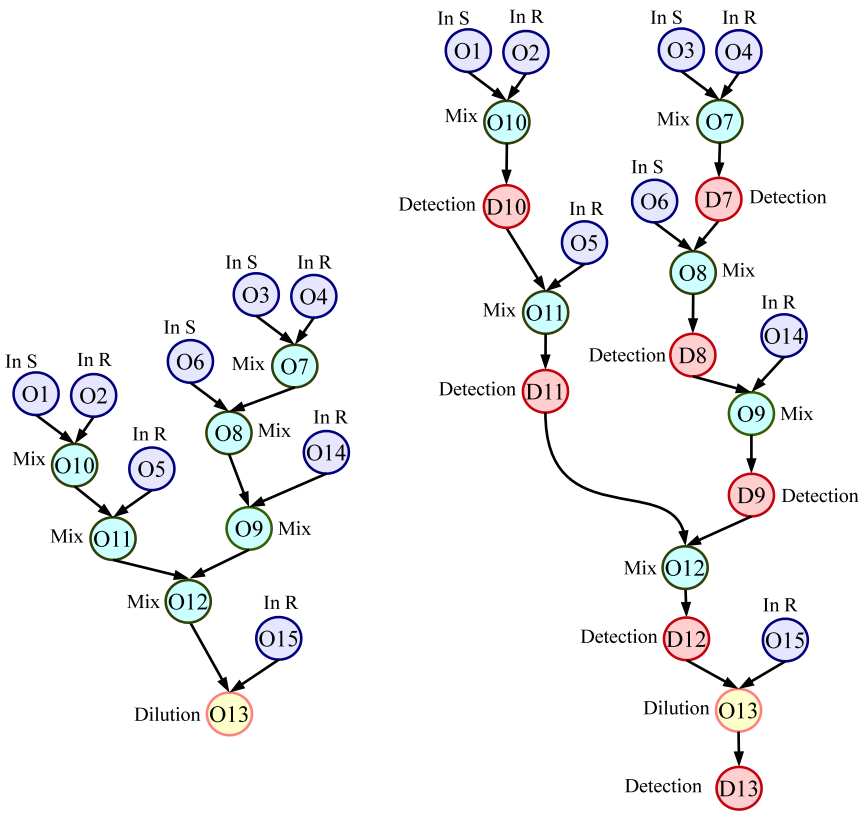
4.2.3 Recovery strategy example

Let us consider the application graph $\mathcal{G}^0$ from Fig. 4.11a, which is executed on a biochip of 8×7 electrodes, with one dispensing reservoir for the sample *In S*, and one for the reagent *In R*, using the module library from Table 2.2. Detection is performed using a capacitive sensor. During the offline step 1 (see Fig. 4.10a), the detection operations D_{7-13} are inserted in $\mathcal{G}^0$, after operations O_{7-13}, respectively, as depicted in Fig. 4.11b. To determine the locations of the detection operations, we use the error propagation model from [95], as described in Section 2.4.2, considering a threshold error $E_{Thr} = 10\%$. For this example, we assume that, during step 1, time redundancy was assigned for all detection operations. The resulted graph with detection operations and corresponding time-redundant subgraphs is given as input to the offline compilation, which derives the results from Fig. 4.12a.

Let us assume that only one transient fault occurs during the execution of the application, and it affects operation O_{12}. Hence, at time $t = 4\ s$ when the detection operation D_7 finishes executing, no error will be detected. We now have the information that D_7 has not detected an error, so we invoke online steps 4 and 5 from our online recovery strategy depicted in Fig. 4.10a. During step 4, we decide when to introduce detection operations and which redundancy techniques to use. In step 5 we compile this new implementation online, updating thus the "electrode actuation sequence".

Thus, **ROS** will be called in step 4 and will decide to reduce the number of detection operations to only three (D_9, D_{11} and D_{13} from Fig. 4.14a), and insert the redundant subgraphs R_9^{Space} for D_9, R_{11}^{Space} for D_{11} and R_{13}^{Time} for D_{13}. The details of how **ROS** works are presented in the next section. In this example we show that **ROS** has decided to remove the detection operation D_{12} (see Section 4.2.4.1 for a discussion on the advantages of such a decision). When finishes, **ROS** will output the graph $\mathcal{G}^R$ with the new detection operations and redundant subgraphs from Fig. 2.15.

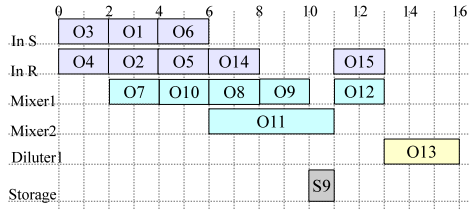
The compilation in step 5 takes as input $\mathcal{G}^R$ and derives a new implementation. For step 5, we use a LS-based compilation, see Section 3.1.3, to perform binding, placement, routing and scheduling. Part of the resulted schedule is presented in Fig. 4.12b, between $t = 4$ and $t = 10$ s. In Fig. 4.12b, which depicts the execution of the application at runtime, the overhead due to the execution of the online steps is represented as



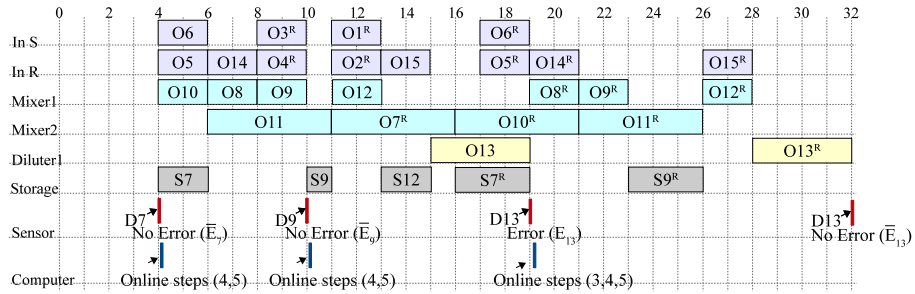
a) Application graph $\mathcal{G}^0$

b) $\mathcal{G}^+$ at t = 0 for E_{Thr} = 10%

Figure 4.11: Initial offline redundancy assignment



a) Schedule determined offline



b) The execution of the application from Fig. 4.11a when an error occurs in operation O_{12}

Figure 4.12: Schedules for execution of application from Fig. 4.11a

a blue line under the row labeled ”Computer”. The redundant operations, part of the inserted redundant subgraphs, are marked O_i^R in the schedule (e.g., O_1^R).

The new implementation continues to execute until the next detection operation finishes. As depicted in Fig. 4.12b, the online steps 4 and 5 are invoked again at $t = 10$ s, after detection D_9 finishes executing. Part of the new resulted schedule is depicted in Fig. 4.12b, between $t = 10$ and $t = 19$ s. As mentioned, we have assumed that a transient error will affect O_{12}. (Note that the errors are unpredictable.) The error in O_{12} is detected by D_{13} (since the error will propagate) at $t = 19$. This will trigger the online recovery step 3 of our strategy, followed by steps 4 and 5. The application completes in 32 s and has tolerated the transient fault in O_{12}.

4.2.4 Assignment of redundancy for error recovery

Our **ROS** is presented in Fig. 4.13. It takes as input the detection operation D which triggered it, the graph $\mathcal{G}'$, the biochip architecture $\mathcal{A}$, the estimated number of faults q^0, the number q of faults occurred so far and the current time t. $\mathcal{G}'$ is the currently executing application graph, from which we have removed the operations which have finished executing, the previously decided detection operations and their associated

ROS($D, \mathcal{G}', \mathcal{A}, q^0, q, t$)

1: E_{Thr} = AdjustErrorThreshold(q^0, q, t)
2: Q = DetermineDetectionOperations($\mathcal{G}', E_{Thr}$)
3: $\mathcal{G}^R$ = InsertDetections($\mathcal{G}'$,Q)
4: Prioritize(Q)
5: $AreaFunction$ = GetAreaFunction($\mathcal{G}', \mathcal{A}, t$)
6: **repeat**
7: D_i = Head(Q)
8: R_i^{Space} = SpaceRedundantSubgraph($D_i, \mathcal{G}'$)
9: r_{area} = RequiredArea(R_i^{Space})
10: a_{area} = AvailableArea($AreaFunction, R_i^{Space}$)
11: **if** $a_{area} \geq r_{area}$ **then**
12: Insert(R_i^{Space}, $\mathcal{G}^R$)
13: UpdateArea($AreaFunction$, R_i^{Space})
14: **end if**
15: **until** $Q = \emptyset$ or $a_{area} < r_{area}$
16: **for each** remaining D_i in Q **do**
17: R_i^{Time} = TimeRedundantSubgraph($D_i, \mathcal{G}^R$)
18: Insert(R_i^{Time}, $\mathcal{G}^R$)
19: **end for**
20: **return** $\mathcal{G}^R$

Figure 4.13: Redundancy Optimization Strategy

recovery subgraphs.

ROS has three components. First, it decides where to insert detection operations, lines 1–3 in Fig. 4.13 and discussed in Section 4.2.4.1. Second, for each inserted detection operation, **ROS** decides between time and space redundancy, see Section 4.2.4.2. **ROS** prefers space redundancy for important operations as long as there is enough area for the corresponding redundant subgraph (lines 4–15), and uses time redundancy for the rest (lines 16–19). Third, **ROS** has to determine, for each redundancy scheme introduced, the redundant subgraph R_i. This is done in lines 8 and 17 in Fig. 4.13, as discussed in Section 4.2.4.3.

Based on the error information after the detection operation and on the current configuration (redundant droplets available), the goal is to minimize the resources used by redundancy (slack time and area) such that the number of tolerated transient faults is maximized. **ROS** produces a new application graph $\mathcal{G}^R$, with updated detection points and fault-tolerance, which is passed to the online compilation in step 5, Fig. 4.10a.

4.2.4.1 Deciding the detection operations

The detection operations and the associated redundancy are required for fault-tolerance. However, redundancy introduces delays in the application execution in case there are no faults. In case of faults, it is important to detect and recover from them as soon as possible, so that no time is wasted. Researchers have used the error analysis from Section 2.4.2, based on a designer-specified error threshold E_{Thr}. To decide where to introduce the detection operations, a given E_{Thr} assumes a number of faults q^0 that may happen during a given time (this is similar to the fault rate of VLSI circuits). **ROS** decides on the detection operations by adjusting E_{Thr}. Then, it uses the error analysis from Section 2.4.2 with the new E_{Thr} to decide the detections. This is especially important for biochips used in applications that require monitoring over a long time, such as bioterrorism, environment and water monitoring.

The threshold E_{Thr} is adjusted in the **AdjustErrorThreshold** function in line 1, Fig. 4.13. The function receives the number of faults q^0 expected over a given time period, the number of faults q that have happened so far, and the time t. The time period is specified as a multiple of the application deadline (which is also its period, for monitoring applications) and the time t is relative to the current invocation of the application. We assume that the faults are uniformly distributed in time. This assumption is used only to adjust E_{Thr} and does not affect our ability to provide fault-tolerance. In case q is larger than expected, E_{Thr} is decreased proportionally, allowing more detection operations to be inserted. Otherwise, E_{Thr} is increased, resulting in less detection operations. Considering the example from Fig. 4.11a, and that no fault happened so far, at time $t = 4$ s we adjust E_{Thr} from 10% to 12%.

We then call the function **DetermineDetectionOperations** using the new E_{Thr} values, line 2. The function uses the error analysis from Section 2.4.2 to calculate the error limits for each operation in $\mathcal{G}'$. For the example in Fig. 4.14a, we conclude that operations O_9, O_{11} and O_{13} exceed the threshold error $E_{Thr} = 12\%$. It follows that for operations O_9, O_{11} and O_{13} we will need the detection operations D_9, D_{11} and D_{13} which are returned as a queue Q. Finally, the function **InsertDetections** from line 3 inserts the detection operations from Q into the graph $\mathcal{G}'$ (see the graph from Fig. 4.14a).

4.2.4.2 Redundancy Optimization Strategy

For each operation D_i in Q, **ROS** has to decide the associated redundant subgraph R_i, and insert it into the current graph $\mathcal{G}^R$. Section 2.4.2.2 has discussed the trade-offs between time and space redundancy. Our heuristic strategy in **ROS** is to introduce space redundancy (because it saves time at the expense of area) only if the extra area used does not lead to greater delays (because regular operations do not have space to

execute on the biochip). For the cases when **ROS** decides that space redundancy is not appropriate, it introduces time redundancy instead.

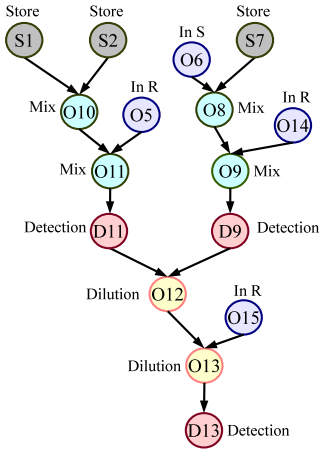
Thus, in the repeat loop (lines 6–15 from Fig. 4.13), we decide where to introduce space redundancy. We consider every operation D_i in the queue Q. At line 4, we prioritize the order in which we visit the detections according to a priority function $Priority(D_i)$. The critical path is defined as the longest path in the graph [67], between the root and the leaf nodes. Hence, we want to prioritize those detection operations (1) for which we predict that an error is more likely to occur and (2) whose redundant droplets produced by space redundancy can be reused by operations on the critical path of the application graph, in case the predicted error does not occur. These two cases are captured by the two terms of the following equation, where a and b are weights given by the designer:

$$Priority(D_i) = a \times E_i + b \times RF_i, \tag{4.2}$$

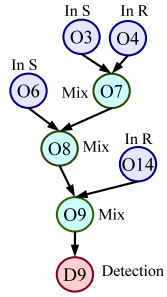
(1) Regarding the first term, we assume that an error is more likely to occur if the error limit E_i (first term) of the operation D_i is higher.
(2) The second term is calculated in the following way: in case an error does not occur we would like to reuse the correctly sized redundant droplets produced by the subgraph R_i^{Space}. The completion time $\delta_{\mathcal{G}^R}$ of the current graph $\mathcal{G}^R$ is determined by the critical path of $\mathcal{G}^R$. To reduce $\delta_{\mathcal{G}^R}$, we prefer that the droplets from R_i^{Space} are reused by operations on the critical path. This is captured by the reusability factor RF_i in the second term. The reusability factor RF_i is given by the cumulative execution time T_i over all the operations that can use the droplet produced by R_i^{Space}. For a fair comparison to E_i, which is a percentage, we obtain RF_i by dividing T_i to the execution time of the critical path. For example, let us consider the detection operation D_9. The droplet produced by R_9^{Space} can be reused by operation O_9, in case an error is detected by D_9, or by operations O_{12} and O_{13}, otherwise (Fig. 4.14a). The total execution time T_9, calculated for operations O_9, O_{12} and O_{13} is of 8 s. The critical path execution time is 12 s, so we obtain the reusability factor $RF_9 = 0.66$, as shown in Fig. 4.14b.

Considering $a = 0.4$ and $b = 0.6$, we obtained for detections D_9, D_{11} and D_{13} from Fig. 4.14a, the values from Fig. 4.14b. Detection D_{13} has the lowest priority.

Next, we decide for each detection operation, prioritized as explained previously, whether we introduce space or time redundancy. We use a LS-based online compilation, as discussed in Section 3.1.3. LS uses a priority function to select among operations which are ready for execution (this is different from the priority function we use for the detection operations, $Priority(D_i)$). Our approach with **ROS** is to use a lower LS-priority function for the operations O_i^R from the space redundant subgraph R_i^{Space} compared to regular operations. However, even if executed with lower priority, the redundant operations O_i^R produce intermediate droplets that need to be stored on the biochip for later use. Storing the intermediate droplets can take away area from the other operations, causing delays in the application completion time. We can determine exactly these de-



a) Graph at t = 4 for E_{Thr} = 12



c) Recovery subgraph R_9^{Space} for D_9

D_i	E_i	RF_i	Priority	Redundancy
D_9	0.12	0.66	0.44	space
D_{11}	0.122	0.66	0.44	space
D_{13}	0.172	0.33	0.26	time

b) Priority calculation for detection operations

Figure 4.14: Online redundancy assignment at $t = 4$ for the application in Fig. 4.11a

lays by running an online compilation, such as the one in [4], which determines if the introduction of space redundancy delays the application completion time. However, the compilation takes time and has to be run for each detection operation.

Instead, our heuristic with **ROS** is to quickly estimate these delays without performing a compilation, as follows. The repeat loop (lines 6–15 in Fig. 4.13) removes each detection operation D_i from the head of the priority-sorted queue Q. For each such D_i, our approach calculates the required area r_{area} to store the redundant droplets produced by R_i^{Space} (line 9). The required area r_{area} is calculated by traversing R_i^{Space} and determining the maximum number of operations that can execute simultaneously (also known as the maximum width of a tree). For example, the r_{area} for R_9^{Space} in Fig. 4.14c is of $2 \times 9 = 18$ electrodes, since maximum two operations can run in parallel and nine electrodes are needed to store each droplet (see the "Store " operation in Table 2.2).

Next, our heuristic determines the available area a_{area} on the biochip and if a_{area} can accommodate r_{area}, then it introduces space redundancy for detection D_i. We estimate the maximum time interval $[t_i^{start}, t_i^{stop}]$ during which R_i^{Space} will be executed. The start time t_i^{start} is given by the earliest time t when R_i^{Space} can start executing. For example, R_9^{Space} in Fig. 4.14c cannot start executing before $t_9^{start} = 6\ s$, when the reservoir $In\ S$ is free to be used. The stop time t_i^{stop} is calculated starting from the time moment when the detection D_i is executed and adding the critical path execution time for R_i^{Space}. For R_9^{Space} (Fig. 4.14c) $t_9^{stop} = 18\ s$, obtained by adding the critical path execution time of R_9^{Space}, which is 8 s, to the time moment $t = 10\ s$, when detection D_9 finished. The

critical paths are determined offline for every relevant operation and are adjusted online. We use the **AreaFunction** to calculate the available area a_{area} for the determined time interval $[t_i^{start}, t_i^{stop}]$ (line 10 in Fig. 4.13). If there is enough available area, condition checked in line 11, **ROS** decides to introduce space redundancy for D_i.

The repeat loop (lines 6–15 in Fig. 4.13) terminates when there is not enough available area, or if the priority-sorted queue Q is empty. Next, if Q is not empty, **ROS** assigns time redundancy for all the remaining detection operations (lines 16–19). In our example, there is enough storage area for R_9^{Space} and R_{11}^{Space}, so space redundancy is assigned for D_9 and D_{11}. The remaining available area is not large enough to accommodate R_{13}^{Space}, therefore time redundancy is assigned to D_{13}. The space and time-redundant subgraphs are inserted in the graph (lines 12 and 18) obtaining, for the graph in Fig. 4.11a, the graph $\mathcal{G}^R$ depicted in Fig. 2.15.

4.2.4.3 Generating the recovery subgraph

After we have decided on the type of redundancy used, we have to determine the corresponding recovery graphs and then insert them in the graph $\mathcal{G}^R$. The algorithm in Fig. 4.15 determines online the recovery subgraph R_i for a detection operation D_i. The recovery subgraph R_i contains the redundant operations needed to produce the correct droplets for the operation O_i. The subgraph R_i is inserted in the graph by **ROS**, either using space redundancy (line 12 in Fig. 4.13) or time redundancy (line 18 in Fig. 4.13). For example, the recovery subgraph R_9^{Space}, for detection operation D_9, is illustrated in Fig. 4.14c.

Starting from the considered detection D_i, the algorithm uses the breadth-first search (**BFS**) technique to traverse the graph (line 3 in Fig. 4.15). All explored operations are inserted in the recovery subgraph R_i. The search stops when no more operations can be inserted, i.e., the root nodes (which are dispensing operations in our case) are reached. The subgraph R_i is updated online by taking into account the redundant droplets stored on the biochip (lines 9–22). These droplets can be by-product droplets intended for discarding (e.g., produced by a dilution operation) or droplets generated by the redundant operations inserted for recovery. The list L_{stg} keeps track of the by-product droplets and of the ones produced by previous redundant operations. These steps are done offline and the resulted subgraphs are stored for each operation, to be used by **ROS** online. The subgraph R_i is traversed using **BFS**, see the repeat loop (lines 9–22). For each explored operation O_i, the algorithm checks the list of redundant droplets L_{stg}. In case a matching droplet n is found for O_i, the subgraph R_i is pruned (line 14) and L_{stg} is updated (line 15). If no matching droplet is found in the storage units for O_i, then all the unexplored predecessors of O_i are enqueued to be explored. The algorithm stops when there is no operation to be explored.

DetermineRecoverySubgraph(D_i, $\mathcal{G}$)

```
1: L_stg - the list of stored droplets
2: Q - the list of recovery operations
3: R_i = BFS(D_i, G)
4: for each operation O_i in R_i do
5:    if O_i has no successors then
6:       Q.push(O_i)
7:    end if
8: end for
9: repeat
10:   O_i = Q.pop()
11:   label O_i as explored
12:   n = FindStoredDroplet(L_stg, O_i)
13:   if n ≠ ∅ then
14:      PruneGraph(R_i, O_i)
15:      Remove(L_stg, n)
16:      for each predecessor O_j of O_i do
17:         if O_j is not explored then
18:            Q.push(O_j)
19:         end if
20:      end for
21:   end if
22: until Q = ∅
23: return R_i
```

Figure 4.15: Determine recovery subgraph algorithm

In the example from Fig. 2.13a, L_{stg} consists of the unused droplets produced by dilution operations O_3 and O_8. In this case, the algorithm uses the stored droplets and prunes the recovery subgraph R_{11}. Consequently, the size of R_{11} is reduced from 11 operations (Fig. 2.13b) to 7 operations (Fig. 2.13c), leading to a shorter recovery time. The structure of the recovery subgraph depends on the current error scenario, as redundant droplets can result from previous recovery operations.

4.2.5 Error recovery strategy with a CCD detection system

The CCD camera-based detection system is proposed in [47] as an error detection alternative to capacitive sensors. Using a CCD camera, images of the droplets on the biochip are captured periodically and analyzed, using pattern matching, in order to locate the position and the size of the droplets. The main advantage of using a CCD camera-based detection system over a sensor-based detection, is that, since the detec-

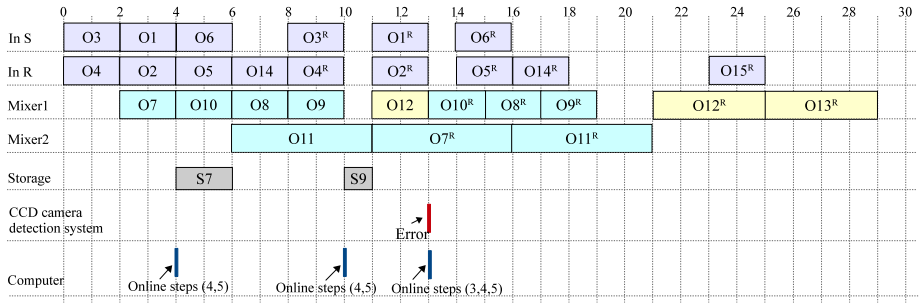


Figure 4.16: The execution of the application from Fig. 4.11a, with CCD detection system

tion is performed simultaneously and continuously, the error is detected immediately when it occurred. When using a sensor, the detection operations are scheduled at specific times and, therefore, the error can be detected long after its occurrence. The online recovery steps are taken as soon as the error is detected. Hence, when using a sensor, the recovery is delayed, resulting in longer completion times. Moreover, the use of a CCD-camera based detection system eliminates the need for routing the droplets to a specific location, or to wait in case there are not enough available sensors. **ROS** is able to optimize the introduction of redundancy because it makes use of the information about fault occurrences. Both situations are important for **ROS**: if an error has happened and if an error has not occurred. With the setup in Fig. 4.10b, ROS would be called only if faults are occurring. Our strategy is to introduce in the application graph places where **ROS** would be called, so it could take informed decisions about how to allocate redundancy. We use the same approach we have used to insert the detection operations in line 3 in Fig. 4.13, see Section. 4.2.4.2, but instead of detection operations we introduce "triggering" operations, which will invoke **ROS** at runtime.

The general strategy of our online recovery approach when using a CCD camera-based detection system is presented in Fig. 4.10b. Images are captured continuously throughout the execution of the bioassay. When the image processing module signals an error, the execution of the bioassay is interrupted and the online steps 3, 4 and 5 are executed. Note that using a CCD camera-based detection system does not require introduction of detection operations in the application graph, as is the case with a capacitive sensor. Hence, **ROS** can be triggered during any operation in step 4, as soon as an error is detected after the recovery in step 3.

Considering the example discussed in Section 4.2.3, if a CCD camera-based detection system is used, the application completes in 29 s, which is 9.3% faster compared to using capacitive sensors. The execution of the application at runtime when using a CCD camera-based detection is depicted in Fig. 4.16. The reduction in completion time comes from detecting the error when it occurred, during O_{12}, at $t = 14$ s. When

a capacitive sensor is used, the detection is scheduled at $t = 19$ s (see Fig. 4.12b), so the error is detected with a delay. For the situations when the additional equipment for image capturing and processing is available, and portability is not required, a detection system based on CCD cameras, provides the fastest results at the moment. However, our proposed **ROS** does not depend on a specific detection method and can be integrated with any available technology.

4.3 Experimental results

For experiments we used seven synthetic benchmarks (SB_{1-7}) [48] and four real-life applications: (1) in-vitro diagnostics on human physiological fluids (IVD, 28 operations) [76]; (3) the colorimetric protein assay (CPA, 103 operations) [76]; (4) the interpolation dilution of a protein (IDP, 71 operations) [95] and (5) the sample preparation for plasmid DNA (PDNA, 19 operations) [45]. In the first two sets of experiments, we ignored the deadline of the applications and the optical detection operations, and the dilution operations were represented as a mix operation followed by a split operation. Hence, for the first two sets of experiments, IVD and CPA have 25 and 134 operations, respectively. The algorithms were implemented in Java (JDK 1.6) and run on a MacBook Pro computer with Intel Core 2 Duo CPU at 2.53 GHz and 4 GB of RAM.

For the first set of experiments we were interested to evaluate the proposed compilation approach in terms of worst-case application completion time $\delta^s_{\mathcal{G}}$, as the number of faults s increases. For this, we have compared the $\delta^s_{\mathcal{G}}$ obtained by our FTScheduling with $\delta^{SFS}_{\mathcal{G}}$ obtained by the Straightforward Scheduling (SFS) approach, considering the same binding and placement, produced by DMBCompilation in line 1 in Fig. 4.5. SFS generates a fault-tolerant schedule by inserting slack, as discussed in Section 4.1.1. Thus, we insert in the application graph $\mathcal{G}$ a "slack" operation after each split operation. The slack execution time is calculated using the formula Equation (4.1). We then apply the LS algorithm from Fig. 3.3 to obtain the fault-tolerant schedule.

We run the first set of experiments on SB_{1-7}, IVD, and CPA applications. The results are presented in Table 4.3, where we have, in separate columns, the schedule lengths of both SFS and FTC approaches for s number of faults varying from 2 to 5. The first three columns contain the application size given in number of operations, the considered biochip area and the number of sensors placed on the biochip, respectively. We can see that the FTC approach results in reduced application completion times compared to SFS, especially as s increases. For $s = 5$ we have obtained an average improvement of 52.4% in the FTC completion time compared to SFS.

Our proposed FTC has three steps: (1) running the adapted implementation from [49] for the specified times (60–1,800 s), (2) generating the fault-tolerant graph, which takes

very little time, and (3) obtaining the fault tolerant schedules. The CPU overhead of the last step increases exponentially with the number of faults s and the number of split operations. For example, for IVD application, which has 4 split operations [76], the CPU execution times for 1 to 5 faults are 0.15 s, 0.45 s, 0.82 s, 1.51 s, 2.65 s, respectively.

For the second set of experiments, we were interested in the impact of reducing costs (in terms of chip area and number of sensors) on the application completion time. The results presented in Table 4.4 are obtained for the IVD application, for a fixed number of faults, $s = 4$. The application is executed initially on a large biochip area of 18×18 on which there are placed 4 sensors, for which we obtained an improvement of 12.1% with FTC over SFS. For the next evaluations, we have reduced the area and the number of sensors. As expected the schedule length increases with the reduced area and number of sensors. However, our proposed FTC approach produces significantly better schedules than SFS, thus allowing us to save costs. For example, in the most constrained case, a biochip of a 12×12 area and 3 sensors, we have obtained an improvement of 48.3% compared to SFS.

For the next sets of experiments we considered the following deadlines for the applications: d_{PDNA} = 60 s, d_{CPA} = 300 s, d_{IDP} = 200 s, respectively. Also, in the next two sets of experiments we have considered only a capacitive sensor for detection, i.e., not a CCD camera system.

In the third set of experiments we were interested to determine if is is important to use a combination of redundancy techniques (i.e., time and space redundancy) and if ROS is able to optimize their allocation. Hence, we have compared (a) our redundancy optimization approach ROS with two cases where we have used (b) only time redundancy for recovery, called TIME, and (c) only space redundancy (SPACE). The

Table 4.3: Comparison between SFS and FTC

App. (ops.)	Area	n_{sns}	$s = 2$		$s = 3$		$s = 4$		$s = 5$	
			SFS	FTC	SFS	FTC	SFS	FTC	SFS	FTC
SB_1 (10)	6×6	1	46	41	56	46	66	51	76	53
SB_2 (20)	8×8	2	37	29	47	36	57	46	67	56
SB_3 (30)	8×12	3	40	36	55	37	70	56	85	76
SB_4 (40)	10×8	2	37	33	48	38	58	40	68	45
SB_5 (50)	8×12	3	44	38	57	43	73	49	87	51
SB_6 (60)	12×10	4	50	45	59	50	65	50	79	52
SB_7 (70)	10×12	4	65	60	82	63	102	66	122	74
IVD (25)	10×10	2	36	31	41	36	51	36	61	41
CPA (134)	15×15	6	88	68	114	73	145	76	176	84

Table 4.4: Results for IVD

Area	Sensors	Schedule length (s)	
		SFS	FTC
18×18	4	46	41
16×16	4	47	41
14×14	3	46	36
12×12	3	46	31

recovery subgraphs for case (b) and (c) were assigned statically offline for all the detection operations. For this set of experiments, we used 3 different biochips (column 1 in Table. 4.5), with sizes of 7×7, 8×9 and 10×10 electrodes. Next to the sizes, we also present in parentheses the numbers of reservoirs for the three reagents (respectively $R1$, $R2$ and $R3$) used by PDNA, see [45] for details. The techniques are compared in terms of the application completion time $\delta_{\mathcal{G}}$ obtained for PDNA. Since a particular error can be favorable to a certain redundancy technique, in the interest of a fair comparison, we have generated randomly 50 error scenarios, and we used for comparison the average value of $\delta_{\mathcal{G}}$ obtained over all scenarios.

Thus, we have simulated the execution of PDNA on each of the three biochips, and we have randomly inserted $q = 1$ and 2 errors in the operations. The obtained average $\delta_{\mathcal{G}}$ for the three cases (a)–(c) are presented in Table 4.5, columns 2, 3 and 5, respectively: (a) $\delta_{\mathcal{G}}^{ROS}$, (b) $\delta_{\mathcal{G}}^{TIME}$ and (c) $\delta_{\mathcal{G}}^{SPACE}$. The reported $\delta_{\mathcal{G}}$ times take into account the runtime overhead required by re-compilation (for all cases) and the runtime of the redundancy optimization, performed only in the case of ROS.

From Table 4.5 we see that ROS, which uses an optimized combination of space and time redundancy, is able to obtain much better results than using a single form of re-

Table 4.5: Comparison between recovery techniques for PDNA

Arch.	(a) $\delta_{\mathcal{G}}^{ROS}(s)$	(b) $\delta_{\mathcal{G}}^{TIME}(s)$	Deviation (%)	(c) $\delta_{\mathcal{G}}^{SPACE}$ (s)	Deviation (%)
7×7 (2, 1, 2)	34.08	56.28	39.4	64.07	46.8
8×9 (2, 1, 2)	32.68	55.61	41.2	58.72	44.3
10×10 (2, 1, 2)	27.5	54.8	49.8	57.92	52.5

dundancy, TIME or SPACE. Compared to TIME, ROS leads to an improvement of 39% (standard deviation) for the 7×7 biochip, 41% for 8×9 and 49% for 10×10 (see column 4). The improvement over SPACE is 46%, 44% and 52%, respectively. Better results were obtained for larger biochip areas, as ROS uses the available space to optimize the introduction of space redundancy and reduce the recovery time. All the considered areas are, however, too small to use space redundancy exclusively. As the biochip area increases, from 7×7 to 10×10, all techniques benefit of the extra area and use it to improve $\delta_{\mathcal{G}}$, hence the decrease in $\delta_{\mathcal{G}}$ as area increases. However, the percentage deviation between ROS and the others gets larger, as ROS is better at exploiting the extra area. The 10×10 area is still too small to use space redundancy exclusively, hence SPACE gives worse results than ROS. Regarding the deadline, all solutions obtained with ROS meet the deadline, i.e., $\delta_{\mathcal{G}}^{ROS} \leq d_{PDNA}$, whereas the deadline is satisfied only in 56% of cases for TIME and 49.4% of cases for SPACE. This experiment shows that by using our proposed ROS, which decides online between the introduction of time and space redundancy, we obtain better results compared to using a single redundancy technique.

In the fourth set of experiments we were interested to compare ROS to the related work. Thus, we compared the completion time $\delta_{\mathcal{G}}^{ROS}$ obtained by ROS with the $\delta_{\mathcal{G}}^{DICT}$ obtained by using the previously proposed dictionary-based error recovery (DICT) [44]. DICT determines offline the recovery needed for an error and the corresponding changes to the electrode actuation sequence for the operations, then, it stores the results in a dictionary, to be used online, when an error is detected. Hence, DICT has negligible runtime overhead for applying the recovery. In contrast, ROS determines both the required recovery and the changes to the electrode actuation sequence (what we call re-compilation) online, during the execution of the biochemical application. We ran experiments for CPA and IDP, using the same error scenarios and biochip configuration as in [44]. The results are presented in Table 4.6 for CPA, and in Table 4.7 for IDP. The completion time $\delta_{\mathcal{G}}^{DICT}$ is presented in column 2, and $\delta_{\mathcal{G}}^{ROS}$ in column 3.

The completion time $\delta_{\mathcal{G}}^{ROS}$ contains the runtime execution overhead of ROS. This overhead is also reported separately in the tables in column 4. These runtimes are cumulative, a summation for all invocations of ROS in the given scenario, and are measured

Table 4.6: Comparison of dictionary-based error recovery [44] and ROS for CPA

Errors (ops.)	$\delta_{\mathcal{G}}^{DICT}(s)$	Total time (s)	CPU time (s)	Deviation (%)
Dlt_{39}	228	212.21	0.98	6.92
Dlt_{12}, Dlt_{31}	220	192.19	0.9	12.64
DsB_4, Dlt_{14}	219	192.25	1.12	12.21
Dlt_{21}, Mix_5	223	219.26	1.06	1.67

Table 4.7: Comparison of dictionary-based error recovery [44] and ROS for IDP

Errors (ops.)	$\delta_{\mathcal{G}}^{DICT}(s)$	Total time (s)	CPU time (s)	Deviation (%)
Dlt_8, Dlt_{16}	208	161	1.7	22.5
Dlt_2, Dlt_{29}	212	175.86	1.5	17
Dlt_{19}, DsB_{23}	207	163.77	0.5	20.8
Dlt_{16}, Dlt_{18}	209	163.65	0.4	21.7

on a typical PC, which is used to control the biochip. As the results in Tables 4.6 and 4.7 show, our approach (ROS) is able to obtain much better results compared to the related work DICT (more than 20% reduction for a third of the cases). The percentage improvement of ROS over DICT (standard deviation) is shown in the last column in the two tables. The improvement of our proposed online redundancy approach comes from the optimized use of recovery techniques employed. For example, for IDP, where a larger biochip area is available for operations, ROS has used space redundancy for carefully selected operations, which trades off area for time, in order to improve the results.

As mentioned in the problem formulation, with ROS we are interested to maximize the number of transient faults tolerated within the application deadline. An application tolerates the faults if the deadline is satisfied, i.e., $\delta_{\mathcal{G}} \leq d_{\mathcal{G}}$, in all the fault scenarios. Thus, in the last set of experiments we were interested to find out if ROS can compile online a fault-tolerant implementation which meets the deadline as the number of faults q increases.

We ran the last set of experiments for all three benchmarks: PDNA, IDP and CPA. We ran the experiments using both detection methods presented previously: the sensor-based detection (Table 4.8) and the CCD camera-based detection system (Table 4.9). The biochip sizes used for each application is presented in column two. Next to the sizes, we also present in parentheses the numbers of reservoirs for the sample, buffer and reagents. We have generated a large number of error scenarios covering possible combinations of q faults and operations. The $\delta_{\mathcal{G}}^{ROS}$ values reported are the shortest completion time (min), the longest completion time (max) and the average completion time (avg.) over all the simulation runs.

The results for $q = 1$, 2 and 3 are presented in Table 4.8 in columns three, four and five, respectively, for the case when a capacitive sensor-based detection is used, and in columns three, four and five in Table 4.9, for the case when CCD camera-based detection is used. As we see from the table, ROS is able to successfully tolerate an increasing number of faults, producing online fault-tolerant implementations which meet the deadline in all cases (the maximum value of $\delta_{\mathcal{G}}^{ROS}$ is less than the deadlines of the respective benchmarks). The redundancy required for fault-tolerance and the runtime

Table 4.8: ROS results for $q = 1$, 2 and 3 faults (capacitive sensor)

App. (ops.)	Arch.	$\delta_{\mathcal{G}}^{ROS}(s)$ q = 1	$\delta_{\mathcal{G}}^{ROS}(s)$ q = 2	$\delta_{\mathcal{G}}^{ROS}(s)$ q = 3
PDNA (19)	7×7 (1, 2, 2)	min 30.24 max 37.25 avg. 32.62	min 30.28 max 37.25 avg. 33.33	min 32.25 max 37.4 avg. 34.62
IDP (71)	9×9 (1, 2, 2)	min 159.66 max 166.63 avg. 161.97	min 159.75 max 177.61 avg. 166.52	min 160.71 max 182.66 avg. 168.04
CPA (103)	11×11 (1, 2, 2)	min 192.65 max 219.78 avg. 198.69	min 192.8 max 219.93 avg. 209.71	min 213.38 max 244.95 avg. 219.61

Table 4.9: ROS results for $q = 1$, 2 and 3 faults (CCD detection)

App. (ops.)	Arch.	$\delta_{\mathcal{G}}^{ROS}(s)$ q = 1	$\delta_{\mathcal{G}}^{ROS}(s)$ q = 2	$\delta_{\mathcal{G}}^{ROS}(s)$ q = 3
PDNA (19)	7×7 (1, 2, 2)	min 25.14 max 33.64 avg. 29.46	min 25.14 max 34.42 avg. 30.38	min 25.22 max 37.19 avg. 31.15
IDP (71)	9×9 (1, 2, 2)	min 139.61 max 169.11 avg. 157.27	min 139.7 max 174.98 avg. 159.87	min 141.9 max 178.98 avg. 160.27
CPA (103)	11×11 (1, 2, 2)	min 192.53 max 215.74 avg. 197.72	min 192.6 max 218.61 avg. 207.03	min 194.06 max 236.79 avg. 217.68

execution of ROS will introduce an overhead. However, it is important to notice that $\delta_{\mathcal{G}}^{ROS}$ increases slowly with q, which means that ROS can successfully tolerate an increasing number of faults. This is because ROS is able to use the fault occurrence information at runtime to optimize the introduction of redundancy, such that the delays on the application completion time $\delta_{\mathcal{G}}$ are minimized. It follows that it is important to use an online redundancy optimization and re-compilation approach if we want to have fault-tolerant biochip implementations.

Finally, by comparing the results from Tables 4.8 and 4.9 we see the difference between the two sensor setups: using a capacitive sensor, which requires the introduction of detection operations (the columns labeled "sensor"), versus using an imaging CCD camera-sensor which can instantly detect an error, the columns labeled "CCD". As

expected, using a CCD camera-sensor leads to better results, because the errors are detected immediately. Our ROS approach can use both setups, and is able to intelligently introduce the detection operations required by the capacitive sensor setup, reducing its inherent delays.

CHAPTER 5

Synthesis of Application-Specific Architectures

Most of the related research so far has considered general-purpose biochip architectures, due to their high reconfigurability. However, in practice, application-specific architectures are preferred because of their reduced cost. In this chapter we present our solutions to the application-specific architecture synthesis problem. Starting from an initial architecture, our synthesis solutions use metaheuristics that search through the solution space to find a minimum cost architecture that can satisfy the timing constraints of the application and tolerate a given number k of permanent faults.

5.1 Problem formulation

The problem can be formulated as follows. Given as input a biochemical application $\mathcal{G}$, modeled as a directed acyclic graph, see Section 2.3, with a deadline $d_{\mathcal{G}}$, a component library $\mathcal{M}$, a fluidic library $\mathcal{F}$ and a maximum k permanent faults to be tolerated, we are interested to synthesize a fault-tolerant physical architecture $\mathcal{A}$, such that the cost of $\mathcal{A}$ is minimized and the application completion time $\delta^k_{\mathcal{G}}$ is within the deadline $d_{\mathcal{G}}$ for any pattern of the k faults.

Table 5.1: Example of component library $\mathcal{M}$ [7]

Name	Unit cost	Area (mm^2)	Time (s)
Electrode	1	1.5×1.5	N/A
Dispensing Reservoir (1 μL)	6.66	5×3	2
Dispensing Reservoir (10 μL)	16.6	7.5×5	2
Dispensing Reservoir (50 μL)	33.3	7.5×10	2
Dispensing Reservoir (100 μL)	52	15×7.5	2
Capacitive Sensor	1	1.5×4.5	0
Optical Detector	9	4.5×4.5	30

Table 5.2: Fluidic library $\mathcal{F}$ for PCR [68]

Fluid Name	Unit Cost/μL
Sample (DNA1)	2.47
Sample (DNA2)	3.3
Reagents*	0.6

*ChargeSwitch gDNA kit from Invitrogen Corp.

We assume that the designer will provide a component library $\mathcal{M}$ and a fluidic library $\mathcal{F}$. The library $\mathcal{M}$ contains a list of the physical components available to design a biochip. An example component library is Table 5.1, where, for each physical component mentioned in column 1, column 2 presents the costs expressed in the unit cost of an electrode, column 3 presents the dimensions and column 4 presents the execution time. As seen in Table 5.1, a dispensing reservoir of 1 μL has a cost of 6.66 units, occupies an area of 15 mm^2 and can dispense one droplet in 2 s. The electrode component (row 1 in Table 5.1) can be reconfigured to perform various operations, thus the electrode has a "N/A" execution time. The operations that can be performed on the electrode components and their execution times are specified in the module library (see Table 5.3). A fluidic library $\mathcal{F}$, such as the one in Table 5.2, contains for each input fluid the cost per μL expressed in the same units as the cost of the components in library $\mathcal{M}$, i.e., the unit cost of an electrode.

We consider that the operations execute on circular-route modules (CRMs), defined in Section 2.2.1, since CRMs can use better a non-regular architecture than rectangular modules. In Section 2.2.1 we presented how to determine the operation execution time on a CRM. We also assume that we know the position of the droplets during the execution, i.e., the operation execution is "droplet-aware". The "droplet-aware" approach [53] has the advantage of improved reconfigurability in case of permanent faults: the droplets are simply instructed to avoid the faulty electrodes.

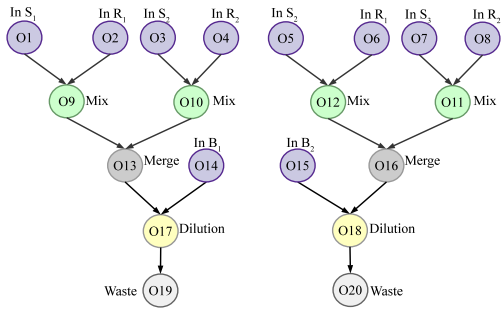


Figure 5.1: Example application graph $\mathcal{G}$ for architecture synthesis

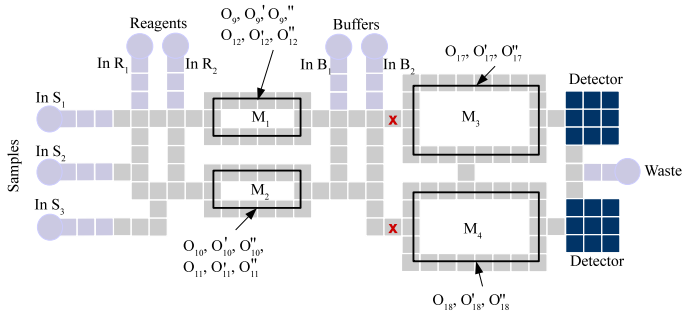


Figure 5.2: Application-specific biochip architecture

Let us consider an application graph $\mathcal{G}$ obtained by repeating three times the graph from Fig. 5.1. We are interested to synthesize a physical architecture for this application, considering $k = 1$ permanent faults, such that the cost is minimized and a deadline of $d_{\mathcal{G}} = 22$ s is satisfied.

So far, researchers have considered only general-purpose biochips of rectangular shape. To complete $\mathcal{G}$ within deadline $d_{\mathcal{G}}$ using a rectangular architecture, let us denote it with $\mathcal{A}_2$, we need an array of 9×16 electrodes and eight reservoirs: two for the reagent, two for the buffer, three for the sample and one for the waste. The rectangular architecture $\mathcal{A}_2$ has 168 electrodes. We used the module library in Table 5.3 and obtained an execution time for $\mathcal{G}$ on $\mathcal{A}_2$ of 18.78 s, which is satisfying the deadline.

However, the number of electrodes can be reduced if we use an application-specific architecture $\mathcal{A}_1$, such as the one in Fig. 5.2, of only 128 electrodes, reducing with 23.8% the number of electrodes of $\mathcal{A}_2$. Since $\mathcal{A}_1$ and $\mathcal{A}_2$ have the same number of reservoirs, i.e., both architectures have identical fluidic cost, we compare $\mathcal{A}_1$ and $\mathcal{A}_2$ only in terms of number of electrodes.

Table 5.3: Library $\mathcal{L}$ of rectangular modules

Op.	Shape	Time (s) no faults	Time (s) $k=1$	Time (s) $k=2$
Mix	3×6	2.52	2.71	3.77
Mix	5×8	2.05	2.09	2.3
Mix	4×7	2.14	2.39	2.51
Mix	5×5	2.19	2.28	2.71
Mix	8×8	1.97	2	2.09
Mix	5×5×1	2.19	2.73	3.92
Mix	5×5×2	3.98	5.82	7.56
Dilution	3×6	4.4	4.67	4.11
Dilution	5×8	3.75	4.76	6.3
Dilution	4×7	3.88	4.22	4.46
Dilution	5×5	3.98	4.12	4.67
Dilution	8×8	3.63	3.66	3.8
Split	1×1	0	0	0
Storage	1×1	N/A	N/A	N/A

For architecture $\mathcal{A}_1$, we determined manually the following worst-case execution times in case of $k=1$ permanent faults: 2.59 s for a mix operation on M_1 and M_2, 5.16 s for a dilution operation on M_1 and M_2, 2.4 s for a mix operation on M_3 and M_4 and 4.47 s for a dilution operation on M_3 and M_4. The binding of operations in the application is shown in the figure; we replicate three times the graph in Fig. 5.1, hence for every O_i, we have O_i' and O_i''. The completion time of $\mathcal{G}$ on architecture $\mathcal{A}_1$ is $\delta_{\mathcal{G}}$ = 18.87 s, within the deadline $d_{\mathcal{G}} = 22s$. In addition, $\mathcal{A}_1$ is also fault-tolerant to k=1 permanent faults, i.e., in the worst-case fault scenario, when the fault is placed such that it leads to the largest delay on $\delta_{\mathcal{G}}$, the application completes in $\delta_{\mathcal{G}}^{k=1}$ = 20.01 s, which satisfies the deadline.

We assume that our architecture synthesis is part of a methodology, presented in detail in Chapter 3 and outlined in Fig. 3.1. We mention again only the steps that regard the architecture synthesis problem, i.e., we omit steps 5 and 6, as follows.

1. *Architecture design.* We synthesize an application-specific architecture $\mathcal{A}$ for an application $\mathcal{G}$ with a deadline $d_{\mathcal{G}}$, considering a maximum number of permanent faults k that have to be tolerated. Since the architecture synthesis is performed before the fabrication (step 2) and testing (step 3), the locations of permanent faults are *not known* during the architecture synthesis.

2. *Fabrication.* We fabricate the biochips with application-specific architecture $\mathcal{A}$, obtained during the previous step.

3. *Testing.* All the biochips are tested to determine if they have permanent faults using testing techniques such as the ones proposed in [88]. If there are more than k faults, the biochip is discarded. The exact locations of permanent faults are determined during this step.

4. *Compilation.* We perform a compilation of application $\mathcal{G}$ on $\mathcal{A}$ to obtain the electrode actuation sequence. Since the locations of permanent faults are known, we can use any compilation implementation, such as the one proposed by [51], to determine the actual completion time $\delta^k_{\mathcal{G}}$. In case $\delta^k_{\mathcal{G}}$ exceeds the application deadline $d_{\mathcal{G}}$, the biochip is discarded.

In this chapter we focus on the first step of the methodology: the architecture synthesis problem, for which we propose two solutions based on metaheuristics, namely Simulated Annealing (SA) and Tabu Search (TS). The flow of our metaheuristic implementations is outlined in Fig. 5.3. A metaheuristic explores the solution space using design transformations called moves, which are applied to the current architecture solution in order to obtain neighboring architecture alternatives.

Next, out of the newly generated architectures, one architecture alternative is selected to be the current solution. Hence, each architecture alternative is evaluated using an objective function defined in terms of the architecture cost and the timing constraints of the application. Generally, a new architecture solution is accepted if it improves the current solution. However, in some cases, the SA metaheuristic accepts worse architecture solutions, in order to escape local minima. The metaheuristic continues to apply the moves on the determined current solution and use the objective function to evaluate the obtained neighboring architectures. The search terminates when a stop criterion is satisfied.

Hence, in this thesis, we propose two solutions to the architecture synthesis problem, as follows.

- Our first architecture synthesis approach uses a SA metaheuristic to search the solution space and generate new architectures. We propose for the evaluation of each architecture solution, a List Scheduling (LS)-based compilation (see Section 5.3.1), which determine the worst-case schedule length in case of maximum k permanent faults, by considering that *each* operation in the application is affected by k faulty electrodes. Instead of using rectangular modules for operation execution, the SA-based synthesis considers rectangular routes of varying thickness, such as the ones in Fig. 5.4, since they can better exploit the non-regular layout of an application-specific architecture. However, due to their non-regular

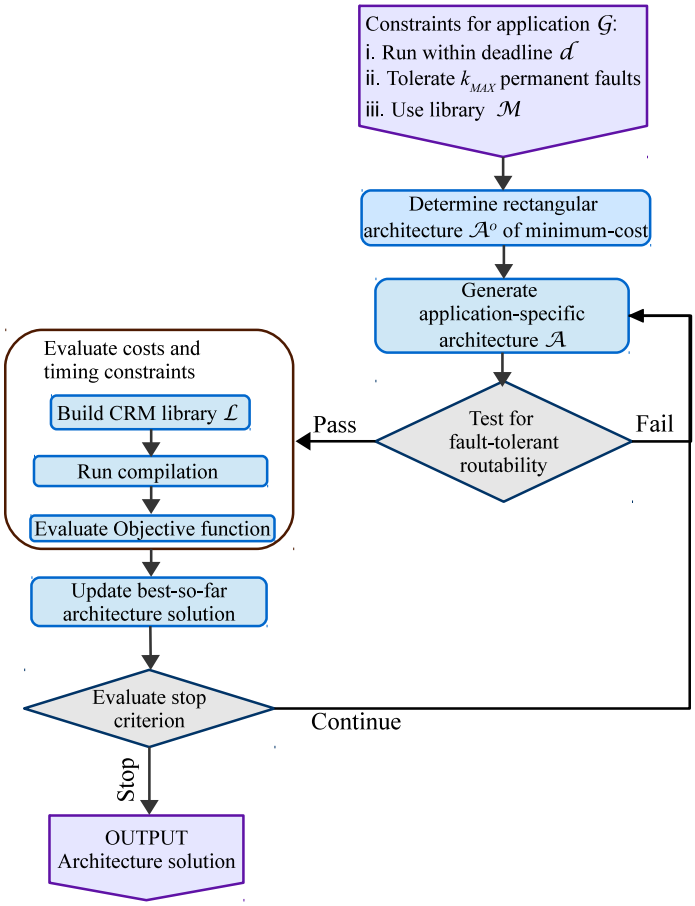


Figure 5.3: Architecture synthesis

shape, CRMs can take full advantage of an application-specific architecture and thus, our second architecture synthesis considers CRMs for operations execution.

- Our second architecture synthesis approach is based on a TS metaheuristic and considers CRMs for operation execution, as CRMs have a non-regular shape, and thus can better exploit the non-regular layout of an application-specific architecture. As mentioned, the SA-based synthesis determines the worst-case operation execution time, which is a safe, but pessimistic approach, as it may result in eliminating potentially good architecture solutions. Instead, we take a different approach for the TS-based synthesis, that is, we propose an algorithm to determine the impact of faults on the operation execution times (see Section 2.2.3) by using a method that is less pessimistic than the worst-case. The latter estima-

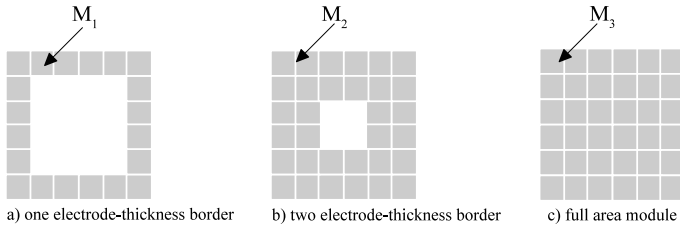


Figure 5.4: Rectangular routes of varying thickness

tion method is faster, and hence more suitable to be used inside a metaheuristic. According to our proposed methodology (see Fig. 3.1 in Chapter 3), in case during fabrication a certain pattern of faults occurs such that the architecture cannot execute the application within the timing constraints, then the biochip is discarded. For each visited architecture solution, the TS-based synthesis builds a library of CRMs, by incrementally updating the previously determined library. This approach, which is faster than building from scratch a library of CRMs, takes advantage of the similarities between neighboring architecture solutions. Our proposed method to incrementally build a library of CRMs is presented in Section 5.4.2.

Next, we present the moves used to generate architecture alternatives and the objective function used to evaluate each architecture alternative. Both architecture synthesis methods proposed (SA and TS) use the same moves and the same objective function.

Space exploration moves. The moves used by the metaheuristics are divided in two classes: (1) adding and removing electrodes and (2) adding, removing and changing the placement of devices, such as reservoirs for dispensing and detectors. We define a chain of electrodes $\mathcal{R}$ as a set of consecutive neighboring electrodes that are all situated on the same coordinate axis (vertical or horizontal). For example, in the case of the application-specific biochip depicted in Fig. 5.11a, the electrodes marked with "x" form a chain, while the ones marked with "y" do not form a chain. Note that a chain can also consist of a single electrode. The moves are denoted with capital letters in the following paragraphs.

(1) We define ten moves that are performed by adding and removing electrodes, as follows:

- adding a chain of electrodes at a random position (*ADDRND*);
- removing a chain of electrodes at a random position (*RMVRND*);

- adding a chain of electrodes at the sides of the architecture, namely at the top ($ADDTOP$), bottom ($ADDBTM$), right ($ADDRGT$) and left ($ADDLFT$);
- removing a chain of electrodes at the sides of the architecture, namely at the top ($RMVTOP$), bottom ($RMVBTM$), right ($RMVRGT$) and left ($RMVLFT$);

(2) Assuming that the considered application uses m samples, r reagents and b buffers, we define the following moves for devices:

- adding a reservoir for samples ($ADDS_i$, $i = 1$ to m), reagents ($ADDR_i$, $i = 1$ to r) and buffers ($ADDB_i$, $i = 1$ to b);
- removing a reservoir for samples ($RMVS_i$, $i = 1$ to m), reagents ($RMVR_i$, $i = 1$ to r) and buffers ($RMVB_i$, $i = 1$ to b);
- adding a detector ($ADDD$);
- removing a detector ($RMVD$);
- modifying the placement a detector ($REPD$), since it can impact the completion time of the bioassay by improving the routing times;

For example, let us consider the architecture from Fig. 5.5a as the current solution $\mathcal{A}^{\text{current}}$. By applying the following moves: add a reservoir for buffer ($ADDB$), add bottom-row of electrodes ($ADDBTM$) and re-place detector ($REPD$), we obtain the neighboring architectures from Fig. 5.5b, c and d, respectively. After applying on $\mathcal{A}^{\text{current}}$ from Fig. 5.5a both the moves on the non-reconfigurable devices, i.e., of type (1), and the moves on reconfigurable components. i.e., of type (2), we obtain a neighborhood $\mathcal{N}$ of 19 architecture solutions.

Objective function. As mentioned, our proposed architecture syntheses use the moves described above to generate architecture alternatives. Each architecture alternative is evaluated using the following objective function:

$$Objective(\mathcal{A}) = Cost_{\mathcal{A}} + W \times max(0, \delta^k_{\mathcal{G}} - d_{\mathcal{G}}), \tag{5.1}$$

where $Cost_{\mathcal{A}}$ is the cost of the architecture $\mathcal{A}$ currently evaluated and $\delta^k_{\mathcal{G}}$ is the completion time in case of k faults of $\mathcal{G}$ on $\mathcal{A}$ obtained with our LS-based compilation. If $\mathcal{G}$ is schedulable, the second term is 0, otherwise, we use a penalty weight W (a large constant) to penalize invalid architectures (leading to unschedulable applications). The SA-based architecture synthesis uses Equation (5.3) for $Cost_{\mathcal{A}}$ and the LS-based compilation presented in Section 5.3.1. The TS-based architecture synthesis uses Equation (5.2) for $Cost_{\mathcal{A}}$ and the LS-based compilation presented in Section 5.4.1.

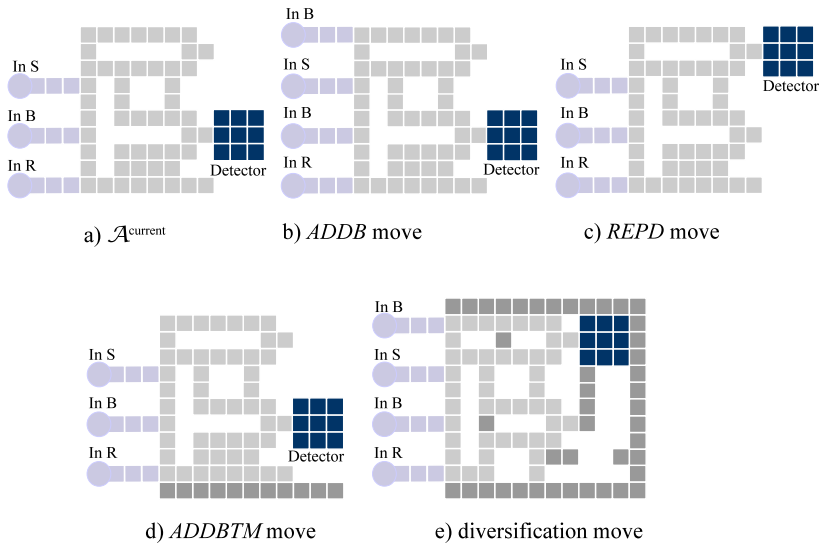


Figure 5.5: Example of neighboring architectures

5.2 Architecture evaluation

As mentioned, to solve the architecture synthesis problem, we propose two solutions based on metaheuristic approaches, which generate architecture alternatives that have to be evaluated in terms of (1) routability, (2) cost and (3) timing constraints. Each of the evaluation criteria is presented in the next paragraphs.

(1) Due to permanent faults, an architecture can become disconnected, i.e., routing of droplets to the desired destination is no longer possible. This is the case for the biochip in Fig. 5.2, considering the two faulty electrodes marked with a red "x". If a biochip architecture can be disconnected by k faults, it should be discarded and in this case the evaluation of the application completion time is no longer meaningful. We want to guarantee that the architecture solution can run the application regardless of the location of k permanent faults. Therefore, we introduce a routability check, which is applied before the completion time evaluation and which verifies if the architecture under evaluation can be disconnected by k faults.

We say that an architecture passes the fault-tolerant routability test, if, in any scenario of k permanent faults, there is at least one route that connects each non-faulty electrode to the other non-faulty electrodes. We used the polynomial time $O(kn^3)$ algorithm from [18] that tests the k-vertex connectivity of a graph, to check the fault-tolerant routability of an architecture. For this purpose we model the architecture as a graph, in

which the nodes represent the electrodes and the edges represent the direct connection between them. Note that an electrode is considered connected only to its top, bottom, left and right neighbors, and not diagonally, since a droplet cannot be moved diagonally with EWOD. The algorithm from [18] tests if the graph remains connected in case of removal of k nodes. For example, the architecture in Fig. 5.2 is still connected for $k = 1$, but becomes disconnected for $k = 2$, e.g., if the 2 faults happen as indicated with the red "x".

(2) The goal of the architecture synthesis is to obtain application-specific biochips of minimum cost. We define the cost of an application-specific architecture using the following equation:

$$Cost_{\mathcal{A}} = \sum N_{M_i} \times Cost_{M_i} + \sum N_{R_i} \times Cost_{R_i}, \tag{5.2}$$

where N_{M_i} is the number of physical components of type M_i, $Cost_{M_i}$ is the cost of M_i, N_{R_i} is the number of reservoirs of type R_i and $Cost_{R_i}$ is the cost of the input fluid for R_i.

The first term of Equation (5.2) calculates the cost of the physical components and the second term calculates the cost of the input fluids. The physical components (e.g., electrodes, reservoirs and detectors) and their unit cost are provided by the designer in a library $\mathcal{M}$ (see Table 5.1 for an example). The unit cost of the input fluids, used by the biochemical application, are specified in a fluidic library $\mathcal{F}$, such as the one in Table 5.2. The assumption is that *all* the reservoirs integrated in the cartridge are fully loaded. We ignore the cost of the controller platform because, regardless of its cost, the controller platform is acquired only once, thus having its cost amortized over time.

(3) We want our synthesis to determine architecture solutions that can tolerate k permanent faults. The application completion time $\delta^k_{\mathcal{G}}$ depends on the location of the k permanent faults. This problem of finding the worst-case schedule length has been addressed in the context of transient faults on distributed multiprocessor systems, and researchers have used "fault-tolerant process graphs" to model all possible fault-scenarios [40]. Such a modeling of all possible fault-scenarios is not feasible in our case because of the interplay between the faulty-electrodes and the allocation, binding, scheduling and placement of operations that can be affected by these faults.

Our first architecture synthesis solution, based on SA, determines the worst-case schedule length in a pessimistic, but safe way, by assuming that each operation in the application is affected by k permanent faults. The LS-based compilation used by the SA-based synthesis is presented in Section 5.3.1. However, since the SA-based synthesis assumes the worst-case pattern of faults for all evaluated architectures, it may also eliminate potentially good low-cost architectures which, after fabrication, when the pattern of faults is known, would have proven able to run the application within its deadline.

Hence, for our second architecture synthesis solution, based on TS, we propose, in Section 5.4.1, a method to determine an estimation of $\delta^k_{\mathcal{G}}$, which is less pessimistic

than considering k faults in each operation. The estimation of $\delta^k_{\mathcal{G}}$ is not safe, i.e., it may sometimes return smaller values than the worst-case ones. As a consequence, the TS-based synthesis may sometimes obtain an architecture solution that for a certain pattern of faults, will not complete the application within the required deadline. However, the actual application completion time is determined through the compilation (see step 4 in the methodology depicted in Fig. 3.1), which is performed after fabrication and testing, when the pattern of faults is known. In case the architecture obtained by the TS-based synthesis fails to complete the application within the deadline, the biochip is discarded.

5.3 SA-based architecture synthesis

As mentioned, to solve the architecture synthesis problem, which is the first step in the methodology (see Fig. 3.1), we initially propose a SA-based approach. SA [12] takes as input the application graph $\mathcal{G}$, the component library $\mathcal{M}$, the number of permanent faults to be tolerated k and the module library $\mathcal{L}$ and produces a fault-tolerant application-specific architecture $\mathcal{A}$ of minimum cost. For the evaluation of each architecture alternative, the SA-based synthesis uses a simplified cost function which does not consider the cost of the input fluids. The cost function used by SA is defined as follows:

$$Cost_{\mathcal{A}} = \sum N_{M_i} \times Cost_{M_i}, \tag{5.3}$$

where N_{M_i} is the number of components of type M_i and $Cost_{M_i}$ is the cost of the *physical* component M_i from the library $\mathcal{M}$.

In the context of application-specific architectures, using rectangular modules for operation execution cannot take full advantage of the biochip area, because of its non-regular layout (see the application-specific architecture example in Fig. 5.2). For that reason we propose CRMs for operation execution on application-specific architectures. CRMs, presented in Section 2.2.1, are circular-routes of non-regular shape, and thus are able to make better use of the non-regular area of an application-specific architecture.

However, the SA-based synthesis proposes a simplification for operation execution by using Rectangular Routes of varying Thickness (RRTs), see Fig. 5.4. RRTs are not as flexible as CRMs, but still can better exploit an application-specific architecture than rectangular modules. Note that rectangular modules are a subset of RRTs, hence SA-based synthesis does not exclude the use of rectangular modules.

For example, two RRTs are shown in Fig. 5.4a and b: module M_1, of one-electrode thickness border and module M_2 of two-electrode thickness border. If the thickness of the border is not mentioned, then we consider that the module occupies the whole rectangular area, as it is the case with module M_3 in Fig. 5.4c.

Our SA-based synthesis generates new alternative architectures by performing moves on the current solution, see Section 5.1 for a detailed presentation of the moves. Each new solution is tested for routability and timing constraints, as presented in Section 5.2. An architecture solution is accepted if it improved the current solution, i.e., it minimizes the objective function in Equation (5.1). However, SA also accepts worse solutions, with a probability that depends on the objective function and the control parameter called *temperature*. We allow SA to explore invalid solutions, in the hope to escape local minima and guide the search towards valid architectures. For each biochemical application, we calibrated the cooling schedule, defined by initial temperature *TI*, temperature length *TL* and cooling ratio ε. The algorithm stops when the *temperature* is cooled down to 1.

The next subsection presents the LS-based compilation used by SA to determine the worst-case completion time of the application $\mathcal{G}$ when it is executed on the application-specific architecture $\mathcal{A}$ with maximum k permanent faults.

5.3.1 Worst-case application completion time analysis

We perform a compilation of the biochemical application $\mathcal{G}$ on the architecture under evaluation $\mathcal{A}$ to determine the worst-case completion time $\delta^k_{\mathcal{G}}$ in case of k permanent faults. We use a LS-based heuristic to perform the binding and scheduling of the operations in $\mathcal{G}$. During scheduling, we also perform placement and routing.

Our LS-based heuristic, called **LSPR** (List Scheduling, Placement and Routing) takes as input the application graph $\mathcal{G}$, the biochip architecture $\mathcal{A}$, the module library $\mathcal{L}$, and the number of permanent faults k to be tolerated, and outputs the worst-case completion time $\delta^k_{\mathcal{G}}$. The library $\mathcal{L}$ contains for each module M_i, the worst-case operation execution times C^k_i considering k permanent faults. The value of C^k_i has to be determined only once, when the module library $\mathcal{L}$ is characterized. In Section 2.2.2, we propose a method to determine C^k_i, for modules of rectangular shape with varying electrode-thickness border (see examples of such modules in Fig. 5.4). An example library $\mathcal{L}$ is Table 5.3. Columns 4 and 5 in Table 5.3 present the values of C^k_i for k=1 and 2, respectively. We have also added to $\mathcal{L}$ in Table 5.3 (rows 6, 7) modules with borders of 1 and 2 electrode-thickness, and empty inside. **LSPR** extends the LS algorithm presented in Section 3.1.3 by using for operation execution the worst-case values from library $\mathcal{L}$, obtained as discussed in Section 2.2.2.

For the placement of operations we have adapted the Fast Template Placement (FTP) algorithm from [10], which uses: (i) free-space partitioning manager that divides the free space in maximal empty rectangles (MERs) and (ii) a search engine that selects the best-fit rectangle for each module. FTP takes as input the module M that needs

to be placed on the biochip architecture $\mathcal{A}$ and the list of MERs L_{rect}. The search engine evaluates all MERs from L_{rect} that can accommodate M, and selects the one which is the nearest to the bottom-left corner of the biochip. We have adapted FTP for application-specific architectures such that we can place modules of rectangular shape with border of varying electrode-thickness such as the ones in Fig. 5.4.

We also need to determine the routes of the droplets between the modules. In case of the black-box approach (see Section 2.2), the droplets have to avoid the modules, thus the route of a droplet can be obstructed by a module.

However, by using the "droplet-aware" approach, we can allow the droplets to pass through the modules. Hence, in our evaluation of $\delta_{\mathcal{G}}^{k}$ we are not interested in the actual routes, only in their length. For that purpose, we have adapted the "filling phase" of Hadlock's algorithm [66] to determine the route lengths, considering the missing electrodes in the array (gaps) as the obstacles to be avoided, and including the worst-case overhead for the detours needed to avoid k faults. As mentioned in Section 2.2, we consider that routing a droplet from one electrode to another takes 0.01 s [60].

5.4 TS-based architecture synthesis

Our second solution to the architecture synthesis problem uses the TS [23] metaheuristic. The features that differentiate the TS-based synthesis from the SA-based synthesis are: (i) a more realistic cost function, (ii) an estimation of the application completion time which is less than the worst-case values and (iii) an algorithm to incrementally update the library of CRMs for operation execution.

(i) As part of the architecture cost, we also consider the fluidic cost, specified in a fluidic library $\mathcal{F}$ (Table 5.2). The cost of reagents is generally expensive and can reach up to 70% of the biochip cost [68, 35]. Hard-to-obtain samples (e.g., from newborn babies, endangered species, unique specimens), also have high cost. When an application is executed, all dispensing reservoirs are fully loaded, thus fluidic cost depends on the number and volumetric capacity of the dispensing reservoirs. Our architecture synthesis varies the number of reservoirs when generating new architecture solutions, in order to increase the parallelism and thus complete the application faster. This is possible because the dispensing operation executes slower that mixing/dilution operations (e.g., for the colorimetric protein assay dispensing executes in 7 s, while mixing executes in 3 s on a 2×3 mixer) [76]. Hence it is important to minimize the use of samples and reagents in order to reduce the total cost of a biochip architecture. Section 5.2 presents in detail the cost function used by the TS-based synthesis when evaluating an architecture.

(ii) The SA-based synthesis used **LSPR**, see Section 5.3.1, which determined the worst-case schedule length by considering that *each* operation in the application is affected by k permanent faults. The approach used by **LSPR** is pessimistic, and it results in rejecting potentially good architecture solutions. Instead, we propose an estimation method for the application execution time, which is less pessimistic than the worst-case values. Our estimation method is faster and thus more suitable to be used inside the TS-based architecture synthesis.

(iii) The SA-based synthesis considered rectangular modules with varying border thickness. Although such modules are suitable for placement on application-specific biochips, their rectangular shape does not permit an effective use of the area on the biochip, due to its non-regular layout. Hence, TS uses circular-route modules (CRMs) for operation execution. In addition we have proposed an algorithm that starts from an existing library $\mathcal{L}'$ determined for the previously visited architecture $\mathcal{A}'$, and incrementally updates $\mathcal{L}'$ for the current architecture $\mathcal{A}$.

The TS-based architecture synthesis takes as input the application graph $\mathcal{G}$, the physical components library $\mathcal{M}$, the number of permanent faults to be tolerated k and the CRM library $\mathcal{L}$ and produces the architecture $\mathcal{A}$ that minimizes the objective function (see Equation (5.1)). TS explores the solution space using design transformations, called moves, to generate the neighborhood $\mathcal{N}$ of the current solution. To prevent cycling due to revisiting solutions, tabu moves are stored in a short-term memory of the search, namely the *tabu list*, which has a fixed dimension, called *tabu tenure*.

However, it may happen that most of the search is done locally, exploring only a restricted area of the search space. In that case, TS uses *diversification* to direct the search towards unexplored regions. Thus, a diversification move is applied to the current solution, and the search is *restarted* from that point.

Fig. 5.6 illustrates our TS-based architecture synthesis. We start the search from the rectangular architecture of minimum cost $\mathcal{A}^{\text{rect}}$ that can run the application within deadline (line 1). The initial solution $\mathcal{A}^{\text{rect}}$ is obtained using exhaustive search by starting from the rectangular architecture of minimum acceptable size and incrementally increasing the dimensions until we obtain an architecture that can run the application within the deadline. To explore the design space, **GenerateNeighborhood** (line 5) generates new neighbor architectures by applying moves to the current solution $\mathcal{A}^{\text{current}}$. We use the moves presented in Section 5.1.

GenerateNeighborhood applies one by one all the moves under the limits conditioned by the execution of the biochemical assay (e.g., at least one reservoir for each input fluid).

However, applying some of the moves can lead to re-visiting solutions, and, consequently, to cycling between already evaluated architectures. To avoid this situation,

TS($\mathcal{G}$, $\mathcal{M}$, k, $d_{\mathcal{G}}$)
1: $\mathcal{A}^{\text{rect}}$ - rectangular architecture that minimizes the Objective function
2: $\mathcal{A}^{\text{best}} = \mathcal{A}^{\text{current}} = \mathcal{A}^{\text{rect}}$
3: $t = 0$
4: **while** $t \leq$ time limit **do**
5: $\mathcal{N}$ = GenerateNeighborhood($\mathcal{A}^{\text{current}}$, *TabuList*)
6: Update(*TabuList*, $\mathcal{N}$)
7: $\mathcal{A}^{\text{current}}$ - solution from $\mathcal{N}$ that minimizes Objective function
8: **if** Objective($\mathcal{A}^{\text{current}}$) < Objective($\mathcal{A}^{\text{best}}$) **then**
9: $\mathcal{A}^{\text{best}} = \mathcal{A}^{\text{current}}$
10: **else**
11: **if** diversification is needed **then**
12: $\mathcal{A}^{\text{current}}$ = ApplyDiversificationMove($\mathcal{A}^{\text{best}}$)
13: Restart(*TabuList*, $\mathcal{A}^{\text{best}}$, $\mathcal{A}^{\text{current}}$)
14: **end if**
15: **end if**
16: **end while**
17: **return** $\mathcal{A}^{\text{best}}$

Figure 5.6: Tabu Search-based architecture synthesis

such moves are considered *tabu*, and are stored in a tabu list. An example of a tabu move is adding a dispensing reservoir after having removed the same reservoir during the previous iteration. Hence, at each iteration, we apply only the moves that are not tabu, and we determine the tabu moves for the next iteration (line 6 in Fig. 5.6).

Each of the architectures from the neighborhood $\mathcal{N}$ is evaluated using the objective function from Equation 5.1, where $\delta^k_{\mathcal{G}}$ is the completion time of the application $\mathcal{G}$ on $\mathcal{A}$ obtained with the proposed **FA-LSR** presented in Section 5.4.1. The new solution $\mathcal{A}^{\text{current}}$ is obtained by selecting the architecture from $\mathcal{N}$ that minimizes the **Objective** function (line 7 in Fig. 5.6). If the currently found solution $\mathcal{A}^{\text{current}}$ is better than the best-so-far $\mathcal{A}^{\text{best}}$, then the latter is updated accordingly (lines 8–10).

In case the search does not find an architecture solution better than $\mathcal{A}^{\text{best}}$ for a number of iterations, then TS uses *diversification* (line 12). A diversification move, composed of two or more non-tabu moves, is applied on $\mathcal{A}^{\text{best}}$ in order to guide the search towards unexplored regions of the search space.

For example, a diversification move composed of the following moves, enumerated in the order of application: *REPD*, *ADDB*, *ADDBTM*, *ADDRND* (9 times), *RMVRND* (6 times) was applied to the architecture from Fig. 5.5a, resulting in the architecture from Fig. 5.5e. The added electrodes are marked in Fig. 5.5e with a darker shade of gray.

Next, the search is restarted from $\mathcal{A}^{\text{current}}$ obtained by applying the diversification move on $\mathcal{A}^{\text{best}}$. The **Restart** function (line 13) updates, if necessary, the architecture $\mathcal{A}^{\text{best}}$ and the tabu list (deletes the previous elements and adds the tabu moves due to diversification). The search continues until the time limit is reached, when our TS-bases architecture synthesis returns $\mathcal{A}^{\text{best}}$.

5.4.1 Application completion time estimation

To estimate the application completion time, we propose a LS-based compilation called **FA-LSR** (Fault-Aware List Scheduling and Routing), which takes as input the architecture under evaluation $\mathcal{A}$, the application $\mathcal{G}$, the library $\mathcal{L}$ and the number of permanent faults k to be tolerated, and outputs the estimated completion time $\delta^k_{\mathcal{G}}$.

FA-LSR distinguishes itself from the LS-based compilation **LSPR** (see Section 5.3.1), as follows. The main difference is concerned with considering the k permanent faults. As mentioned, we do not know the position of the k faults during the architecture synthesis (they will be known after fabrication and testing), so our evaluation has to *estimate* $\delta^k_{\mathcal{G}}$. Instead of considering the *worst-case* scenario as **LSPR**, **FA-LSR** uses an *estimate* for the operation execution, calculated as discussed in Section 2.2.3. The second difference is an extension to the placement of operations, considering CRMs. As explained in Section 2.2.1, the modules on which operations execute consist of circular routes, which do not have to be rectangular. Hence, for the placement of operations we use the algorithm presented in Fig 5.10, Section 5.4.2, which determines the library $\mathcal{L}$ for the currently evaluated architecture $\mathcal{A}$.

Fig. 5.7 presents **FA-LSR**. Every node from $\mathcal{G}$ is assigned a specific priority according to the critical path priority function (line 1) [67]. *List* contains all operations that are ready to run, sorted by priority (line 3). An operation is ready to be executed when all input droplets have been produced, i.e. all predecessor operations from graph $\mathcal{G}$ finished executing. The algorithm takes each ready operation O_i and iterates through the library $\mathcal{L}$, to find the *CRM* M_i that can be placed at the earliest time and executes the operation the fastest (line 6). After O_i is bound to M_i (line 7), **CalculateRoute** (line 8) determines the route that brings the necessary droplets to M_i and O_i is scheduled (line 10). Since the droplets can pass through CRMs when routing (we use a droplet-aware approach), we need to determine only the routing time and not the actual routes. For that purpose, **CalculateRoute** adapts the Hadlock's algorithm [66] to determine the route lengths. *List* is updated with the operations that have become ready to execute (line 11). The repeat loop terminates when *List* is empty (line 12).

Next, in order to obtain an estimate of the application completion in case of maximum k faults, we use the operation execution times determined for each CRM M_i, through the method we propose in Section 2.2.3. As mentioned, when the synthesis is performed,

FA-LSR($\mathcal{G}$, $\mathcal{A}$, $\mathcal{L}$, k)
1: CriticalPathPriority($\mathcal{G}$)
2: $t = 0$
3: $List$ = GetReadyOperations($\mathcal{G}$)
4: **repeat**
5: O_i = RemoveOperation($List$)
6: M_i = FindCRM($\mathcal{L}$)
7: Bind(O_i, M_i)
8: $route_i$ = CalculateRoute($O_i, M_i, \mathcal{G}$)
9: t = earliest time when M_i can be placed
10: $\mathcal{S}$ = Schedule(O_i, t, $route_i$, M_i, $\mathcal{L}$)
11: UpdateReadyList($\mathcal{G}$, t, $List$)
12: **until** $List = \emptyset$
13: L_{CP} = CriticalExecutionPath($\mathcal{S}$)
14: $FaultTable$ = DistributeFaults(L_{CP}, k, $\mathcal{S}$)
15: **for** O_i in $FaultTable$ **do**
16: k_i = number of faults for O_i
17: UpdateSchedule($\mathcal{S}$, O_i, k_i, $\mathcal{L}$)
18: **end for**
19: **return** $\delta^k_{\mathcal{G}}$

Figure 5.7: Fault-Aware List Scheduling and Routing

the location of the permanent faults is not known. Consequently, we do not know which operations are affected by faults and what is the worst-case fault scenario. **LSPR** used a pessimistic approach, by considering that every operation in the application suffers from k faults. Because the length of schedule $\mathcal{S}$ is given by the critical path, which is the path in graph $\mathcal{G}$ with the longest execution time, the approach we propose is to consider that the faults affect the operations that are on the critical path—scenario that will impact most the application completion time. The k faults are distributed among the operations on critical path by the **DistributeFaults** function such that the impact of the faults is maximized.

DistributeFaults takes as input the list L_{CP}, which contains the operations on the critical path, the number of faults k and the schedule $\mathcal{S}$. **DistributeFaults** uses a greedy randomized approach [21] that takes each of the k faults and after evaluating each operation in L_{CP}, distributes the fault to the operation that delays the most the application completion time. Depending on the criticality of specific operations, it may be the case that an operation is affected by more than one fault. Furthermore, if an operation $O_i \in L_{CP}$ is assumed to have a fault, i.e., O_i executes on a faulty CRM M_i, then all operations executing on CRMs that intersect M_i will also be considered affected by a fault. The faulty operations and their corresponding number of faults are stored in $FaultTable$. Finally, for each operation $O_i \in FaultTable$ affected by k_i faults, the

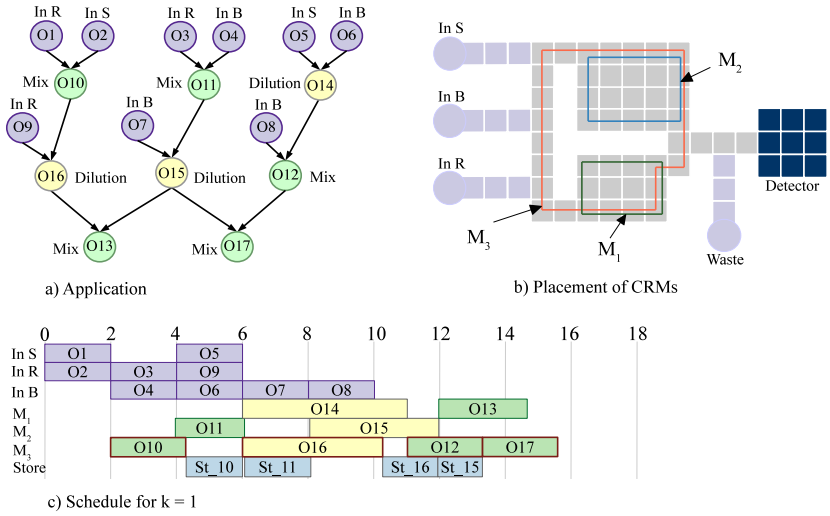


Figure 5.8: Compilation example

schedule is updated (line 17) with the corresponding estimated execution time, determined as explained in Section 2.2.2, and stored in the library $\mathcal{L}$. The application completion time $\delta_{\mathcal{G}}^{k}$ is the finishing time of the last operation in the schedule table (line 19).

Let us assume that we have to compile the application $\mathcal{A}$ from Fig. 5.8a on the architecture from Fig. 5.8b considering $k = 1$ permanent faults. We use the algorithm presented in Section 5.4.2 to determine for $\mathcal{A}$ the CRM library $\mathcal{L}$ shown in Table 5.4, which contains the placement of CRMs, the execution time for $k = 0$ (no faults) and the estimated execution time for $k = 1$ and $k = 2$. For simplicity reasons, in this example we ignore routing and consider that there are no contamination constraints. In order to avoid congestion, the dispensing operations are scheduled only when the corresponding dispensed droplets are needed. At time $t = 2$ s mixing operation O_{10} has the highest priority among all the ready operations (an operation is ready if all its input droplets have arrived). For O_{10}, the CRM M_3 (see Fig. 5.8b) is selected from library $\mathcal{L}$ (Table 5.4), since it finishes the mixing operation the fastest. At time $t = 4.08$ s, operation O_{10} finishes executing. However, the successor of O_{10}, operation O_{16}, is not ready to execute because the other predecessor operation O_{09} has not finished executing. At $t = 6$ s, O_{09} finishes executing, and *List* is updated with operation O_{16}, which becomes ready to execute.

First, **FA-LSR** will produce a schedule of 15.16 s (lines 4–12 in Fig. 3.3). Next, **DistributeFaults** will distribute the $k = 1$ faults to operation O_{17}, since it results in the greatest increase in schedule length. Consequently, operations O_{10}, O_{16} and O_{12}, which

Table 5.4: Fault-tolerant CRM library $\mathcal{L}$ for the architecture in Fig. 5.8b

Operation	CRM	Time (s) k=0	Time (s) k=1	Time (s) k=2
Mix	M_1	2.7	4	15.81
	M_2	2.1	2.4	3
	M_3	2.08	2.3	2.64
Dilution	M_1	5	6.8	16.68
	M_2	3.92	4.44	5.25
	M_3	3.9	4.14	4.4

execute on the same CRM as O_{17}, suffer from $k = 1$ permanent faults. The schedule length is updated with the execution times for the faulty operations $C_i^{k=1}$, taken from library $\mathcal{L}$ (column 4 in Table 5.4). As shown in the schedule from Fig. 5.8c, the completion time $\delta_{\mathcal{G}}^{k=1}$ is 15.6 s.

5.4.2 Incremental build of a CRM library

For our application-specific synthesis we use a TS-based metaheuristic (see Section 5.4), which searches through various architecture solutions in order to find the minimum-cost architecture that satisfies the timing constraints even in the presence of maximum k permanent faults. Each architecture solution is evaluated in terms of routability, cost and timing constraints, as explained in Section 5.2. In order to determine the application completion time $\delta_{\mathcal{G}}^{k}$, and thus check if the timing constraints are satisfied, we need to determine the CRM library $\mathcal{L}$ which is used during the compilation as presented in Section 5.4.1. For each CRM determined during this process, its shape and the corresponding placement on the biochip are also recorded. Hence, the placement task does not need to be implemented during the compilation step. For example, the CRM M_3 determined for the architecture in Fig. 5.8b, has the shape of the following dimensions: $8 \times 7 \times 6 \times 2 \times 3 \times 6$, with the corners placed at coordinates: (0,0), (0,7), (6,7), (6,2), (5,2) and (5,0).

In Section 3.2 we have proposed an algorithm, **BuildLibrary**, that builds a library $\mathcal{L}$ of CRMs for the application-specific architecture $\mathcal{A}$, which does not take faults into account. **BuildLibrary** is only used to build the initial library $\mathcal{L}^0$ for the initial architecture solution $\mathcal{A}^0$, from which TS starts exploring the design space. **BuildLibrary** does not consider faults, hence we use the method presented in Section 2.2.3, to determine for each CRM $\in \mathcal{L}^0$, the operation execution time in case of faults. Since **BuildLibrary** is time consuming and hence cannot be used inside a metaheuristic search, we propose an Incremental Library Build (**ILB**) algorithm that starts from an existing library $\mathcal{L}'$ de-

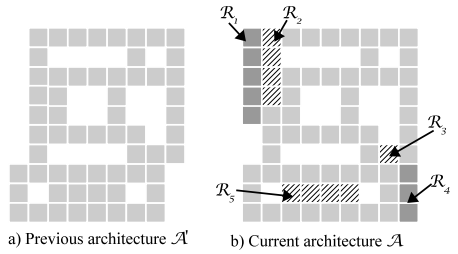


Figure 5.9: Example diversification move

termined for the previously visited architecture solution $\mathcal{A}'$, and incrementally updates it for the current architecture $\mathcal{A}$. This is possible because during the TS-based design space exploration, a new architecture $\mathcal{A}$ is generated by applying *gradual* design transformations to the previous solution $\mathcal{A}'$. Hence, the newly generated architecture $\mathcal{A}$ shares a similar layout with $\mathcal{A}'$. Consequently, the corresponding CRM library $\mathcal{L}$ can be built by incrementally updating $\mathcal{L}'$, that is the library previously determined for $\mathcal{A}'$.

Fig. 5.10 presents the proposed **ILB** algorithm, which takes as input the architecture under evaluation $\mathcal{A}$, the previously determined library $\mathcal{L}'$, the set of electrodes $\mathcal{E}$ included in the transformation and the maximum number of faults k. **ILB** outputs the newly determined library $\mathcal{L}$.

A CRM is defined as a circular-route of electrodes (see Section 2.2.1), and we denote a CRM M as a set of distinctive electrodes $\{e_1, e_2, ..., e_n\}$. An electrode e_n is a neighbor-electrode to e_m, if e_m can be reached directly from e_n. Note that a droplet cannot move diagonally.

Our proposed **ILB** is general, i.e., it can be used for any transformation involving a set of electrodes $\mathcal{E}$, which is decomposed in chains of electrodes (line 1). Let us consider as example the application-specific architecture $\mathcal{A}'$ in Fig. 5.9a. We obtained the architecture $\mathcal{A}$ in Fig. 5.9b after applying a "diversification" move composed of several moves, to $\mathcal{A}'$. In Fig. 5.9b, the added electrodes are marked with a darker shade of gray, while the removed electrodes are hashed. The set of electrodes used by the transformation can be decomposed into the following: adding the chains of electrodes $\mathcal{R}_1$ and $\mathcal{R}_4$ and removing the chains of electrodes $\mathcal{R}_2$, $\mathcal{R}_3$ and $\mathcal{R}_5$.

Each chain of electrodes $\mathcal{R}_j \in \mathcal{E}$, can be in one of the two cases: (1) $\mathcal{R}_j$ is added to $\mathcal{A}'$ or (2) $\mathcal{R}_j$ is removed from $\mathcal{A}'$. In both cases we first determine the CRMs from $\mathcal{L}'$ on which $\mathcal{R}_j$ has an impact (line 3) and then we adjust those CRMs (lines 4–12) so that the adjustment will improve the operation completion time. Next, for each newly adjusted CRM, **ILB** estimates the operation execution time in case of maximum k permanent faults (line 13) and updates the library accordingly (line 14).

ILB($\mathcal{A}$, $\mathcal{L}'$, $\mathcal{E}$, k)

```
1:  L_C = DecomposeInChains(E, A)
2:  for R_j ∈ L_C do
3:      L_M = DetermineCRMList(L', R_i)
4:      for M_i ∈ L_M do
5:          if R_j is added then
6:              H_i = FindNeighborChain(M_i, R_j)
7:              AdjustCRM(M_i, H_i)
8:          end if
9:          if R_j is removed then
10:             route = DetermineRoute(M_i, A)
11:             ReconstructCRM(M_i, route)
12:         end if
13:         EstimateOpExecution(M_i, k)
14:         UpdateLibrary(L, M_i)
15:     end for
16: end for
17: return L
```

Figure 5.10: Incremental Library Build algorithm

Let us present in detail how our proposed algorithm works. First, **ILB** decomposes the set of electrodes $\mathcal{E}$ in chains of electrodes which are stored in the list L_C. Next, for each chain of electrodes $R_j \in \mathcal{E}$, **DetermineCRMList** (line 3) determines L_M—the list of CRMs on which $\mathcal{R}_j$ has an impact. The strategy used by the **DetermineCRMList** function depends on whether $\mathcal{R}_j$ is *added* or *removed*.

For the first case, when $\mathcal{R}_j$ is *added* to $\mathcal{A}'$, **DetermineCRMList** selects from the library $\mathcal{L}'$, the CRMs impacted by this move, i.e., the CRMs that include at least one neighbor-electrode to an electrode in $\mathcal{R}_j$. Those CRMs are stored in the list L_M. Let us consider the example from Fig. 5.11b, where $\mathcal{R}_j$ is the chain of the electrodes colored with a darker shade of gray. The CRM labeled with M_1 is neighboring three electrodes from

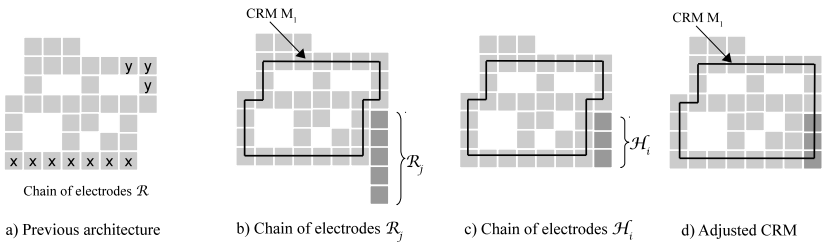


Figure 5.11: Adjusting a CRM in case (1)

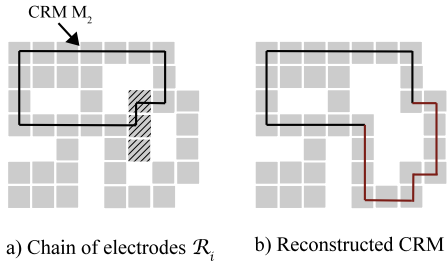


Figure 5.12: Reconstructing a CRM in case (2)

$\mathcal{R}_j$, and consequently M_1 is added to L_M. In case (2), when $\mathcal{R}_j$ is *removed* from $\mathcal{A}'$, **DetermineCRMList** (line 3) adds to L_M the CRMs that contain any electrode in $\mathcal{R}$. Let us consider the application-specific biochip from Fig. 5.12a, where the chain of electrodes to be removed $\mathcal{R}_j$ is hashed. The CRM labeled with M_2 contains electrodes in $\mathcal{R}_j$, and consequently M_2 is included in L_M.

Next, **ILB** tries to adjust each $M_i \in L_M$ so that the operations will complete faster. Reconfigurable operations (e.g., mixing, dilution) complete faster when the forward movement of the droplets is prioritized and the backward movement is avoided [58]. Hence, for case (1), the newly *added* electrodes $\in \mathcal{R}_j$ are used to adjust the CRMs so that forward movements are prioritized. In order to do that, **FindNeighborChain** (line 6) inspects all electrodes in $\mathcal{R}_j$ to determine $\mathcal{H}_i$—the longest chain of electrodes that has both ends as neighbor-electrodes to an electrode in $\mathcal{M}_j$. **AdjustCRM** (line 7) includes $\mathcal{H}_i$ in M_j only if the adjustment results in a greater count of forward movements. For the example in Fig. 5.11b, we have determined the chain of electrodes $\mathcal{H}_i$ in Fig. 5.11c, and the adjusted M_1 in Fig. 5.11d.

In case (2), when $\mathcal{R}_j$ is removed from $\mathcal{A}'$, the CRMs $M_i \in L_M$ have to be re-routed to avoid the removed electrodes. After the removal of the hashed electrodes in Fig. 5.12a, the route of the CRM labeled M_2 has to be re-routed as shown in Fig. 5.12b. Since **ILB** is used inside a search metaheuristic, we are more interested in finding a new route fast, then in finding the shortest route. Hence, in order to find a new connecting route for M_i, **DetermineRoute** (line 10) uses Soukup's algorithm [66]. In Soukup's algorithm a line segment is drawn starting from the source and moving towards the target. The direction is not changed unless the line segment cannot be further extended. The electrodes neighboring the end of the line segment are searched to find an electrode in the direction of the target. Another line segment is extended from that electrode. The search for a route continues in this manner until the target electrode is reached.

In case such a route cannot be found, M_i is removed from L_M, otherwise **ReconstructCRM** (line 11) includes the route in M_i. For the considered example, Fig. 5.12b

shows the reconstructed CRM, which, due to an increased number of turns, requires a longer time to complete the operation compared to the initial CRM M_2 (see Fig. 5.12a).

Next, for each $M_i \in L_M$, **EstimateOpExecution** (line 13) determines a parametric estimation of the operation execution time in case of maximum k permanent faults. The algorithm used by **EstimateOpExecution** is presented in Section 2.2.3. Finally, the library is updated (line 14) and it can be used by the **FA-LSR** compilation to determine the application completion time.

5.5 Experimental results

To evaluate our strategies we have used five synthetic benchmarks (SB_{1-4}) [48] and five real-life applications: (1) the mixing stage of polymerase chain reaction (PCR, 7 operations) [76]; (2) in-vitro diagnostics on human physiological fluids (IVD, 28 operations) [76]; (3) the colorimetric protein assay (CPA, 103 operations) [76]; (4) the interpolation dilution of a protein (IDP, 71 operations) [95] and (5) the sample preparation for plasmid DNA (PDNA, 19 operations) [45]. The algorithms were implemented in Java (JDK 1.6) and run on a MacBook Pro computer with Intel Core 2 Duo CPU at 2.53 GHz and 4 GB of RAM.

The focus of this chapter is to determine if application-specific architectures are more cost-effective than rectangular architectures. Thus, we have used our SA optimization approach to synthesize architectures for PCR, IVD and CPA applications with deadlines 10 s, 15 s and 100 s, respectively (we ignored the detection operations for this set of experiments). We used the component library from Table 5.5. The results are presented in Table 5.7. Together with the results obtained by SA, we have also determined, using exhaustive search (which varies the $m \times n$ dimensions and the number of reservoirs), the cheapest general purpose architecture (column 3), which can run the application within the deadline. The size of the architectures for $k = 0$, 1 and 2 are presented in columns 2, 5 and 8, respectively (the number in parentheses refer to the numbers of reservoirs for buffer, sample and reagent) and their cost is in columns 4, 7 and 10. Both exhaustive search and our SA-based architecture synthesis used the proposed **LSPR** (see Section 5.3.1) for compilation.

Table 5.5: Component library

Name	Cost	Area (mm^2)	Time (s)
Electrode	1	1.5×1.5	N/A
Dispensing reservoir	3	2.5×2.5	2
Optical Detector	9	4.5×4.5	30

Table 5.6: Fluidic library

Fluid Name	Cost
Sample	1
Buffer	4
Reagent	10

We have run our SA-based synthesis for an hour for each experiment. The results of SA for $k = 0$, 1 and 2 are presented in column 4, 7 and 10, respectively, where C_{SA} is the cost of the architecture. For the cooling schedule we used $TI = 40$, $TL = 100$ and $\varepsilon = 0.97$. As we can see from Table 5.7, our SA is able to produce application-specific architectures which are significantly cheaper than the best general purpose architecture.

In the second set of experiments, we used our TS optimization approach to synthesize architectures for the PCR, PDNA, IDP and SB_1 applications. We have compared the architectures outputted by TS with the minimum cost rectangular architecture obtained with exhaustive search. Both exhaustive search and our TS-based architecture synthesis used the proposed **FA-LSR** Section 5.4.1) for compilation. The CRM library $\mathcal{L}$ was determined using the **BuildLibrary** algorithm, presented in Section 3.2, for exhaustive search and the proposed **ILB** (see Section 5.4.2) for the TS-based architecture synthesis. We have run the experiments for $k = 0$, 1 and 2 faults, estimating the operation execution time as proposed in Section 2.2.3. We used the component library $\mathcal{M}$ and fluidic library $\mathcal{F}$ from Table 5.5 and 5.6, respectively.

The results are presented in Table 5.8. The deadline d_G for each application is presented in column 2, the size of the minimum cost rectangular architectures for $k = 0$, 1 and 2 are presented in columns 3, 6 and 9, respectively (the number in parentheses refer to the numbers of reservoirs for buffer, sample and reagent) and their cost is in columns 4, 7 and 10. Note that the deadlines are different than in the previous set of experiments, where we ignored the detection operations.

We have run our TS-based synthesis for an hour for each experiment. The results of TS for $k = 0, 1$ and 2 are presented in columns 5, 8 and 11, respectively. As we can see from Table 5.8, our TS is able to produce application-specific architectures which are significantly cheaper than the best general purpose architecture. For the PDNA application, our proposed synthesis obtained architectures that reduce the cost with 22.4%, 25.9% and 9.2% for $k = 0$, 1 and 2, respectively. Our proposed methodology

Table 5.7: Application-specific synthesis results obtained by SA

	$k=0$			$k=1$			$k=2$		
App.	Arch	C_{RECT}	C_{SA}	Arch	C_{RECT}	C_{SA}	Arch	C_{RECT}	C_{SA}
PCR	7×10 (1,1,1)	79	60	7×10 (1,1,1)	79	65	9×11 (1,1,1)	108	98
IVD	7×10 (2,2,2)	88	62	7×10 (2,2,2)	88	70	10×8 (2,2,2)	98	85
CPA	7×8 (2,1,2)	71	59	7×8 (2,1,2)	71	66	11×12 (2,1,2)	147	127

*We ignored the detection operations for this set of experiments

Table 5.8: Application-specific synthesis results obtained by TS

		$k=0$			$k=1$			$k=2$		
App.	$d_{\mathcal{G}}$	Arch	C_{RECT}	C_{TS}	Arch	C_{RECT}	C_{TS}	Arch	C_{RECT}	C_{TS}
PCR	10	5×5 (2,1,1,0)	53	47	7×6 (1,1,1,0)	66	54	8×6 (1,1,1,0)	72	61
PDNA	60	5×5 (1,1,1,0)	49	38	6×5 (1,1,1,0)	54	40	6×5 (1,1,1,0)	54	49
IDP	200	5×5 (1,1,1,4)	85	74	5×5 (1,1,1,4)	90	84	5×5 (1,1,1,4)	90	84
SB_1	100	5×5 (1,1,1,3)	76	83	5×5 (1,1,1,4)	85	76	5×5 (1,1,1,4)	85	84

Table 5.9: Comparison between TS-based synthesis and SA-based synthesis

	$k=0$		$k=1$		$k=2$	
App.	C_{SA}	C_{TS}	C_{SA}	C_{TS}	C_{SA}	C_{TS}
PCR	60	43	65	46	98	78
IVD	62	56	70	62	85	78
CPA	59	49	66	63	127	123

*We ignored the detection operations

can also support the designer in performing a trade-off between the yield and the cost of the architecture, by introducing redundant electrodes to tolerate permanent faults. The increase in cost for $k = 1$ and $k = 2$ compared to the cost of the non-fault-tolerant architecture, is presented in columns 8 and 11 for TS. For the PCR application (see row 1 in Table 5.8), introducing redundancy for fault-tolerance resulted in a increase of 12.9% in the architecture cost.

In the third set of experiments, we compared the costs of the architecture obtained by the TS-based synthesis to the cost of the architecture obtained using the SA-based synthesis. For a fair comparison, we used Equation 5.3 cost calculation, which did not consider the cost of the input fluids. Since in this set of experiments we do not consider the optical detection operations, we have adjusted the deadlines to 10 s, 15 s and 100 s for PCR, IVD and CPA, respectively. The results are presented in Table 5.9. As we can see, our TS-based architecture synthesis is able to obtain better results. For example, for IVD (row 2 in Table 5.9), a reduction in cost of 28.3% was obtained for $k = 0$ using our TS-based synthesis.

In the fourth set of experiments we were interested to determine the quality of the proposed LS-based compilations, namely **LSPR** and **FA-LSR** (Sections 5.3.1 and 5.4.1),

Table 5.10: Evaluation of the LSPR/FA-LSR compilations (no faults and rectangular architectures)

App.	Arch.	$\delta^0_{\mathcal{G}}(s)$	CPU time	$\delta^{opt}_{\mathcal{G}}(s)$	CPU time	Deviation (%)
PCR	9×9	11	25 ms	10	60 min	9
IVD	9×10	77	91 ms	73	60 min	5.4
CPA	11×12	219	498 ms	214	60 min	2.3

*We ignored the detection operations for this set of experiments

Table 5.11: Increase in application completion time ($k = 0, 1, 2$)

App.	Cost	$\delta^0_{\mathcal{G}}$ (s)	$\delta^1_{\mathcal{G}}$ (s)	Deviation (%)	$\delta^2_{\mathcal{G}}$ (s)	Deviation (%)
PCR	98	8.42	8.81	4.6	9.43	11.9
IVD	85	12.62	13.11	3.8	14.81	17.3
CPA	129	153.9	169.3	10	190.11	23.5

*We ignored the detection operations for this set of experiments

in terms of the application completion time $\delta_{\mathcal{G}}$. We have compared $\delta_{\mathcal{G}}$ to the nearly-optimal $\delta^{opt}_{\mathcal{G}}$ obtained in [50] using TS for the compilation. Note that $\delta_{\mathcal{G}}$ is determined for the case when there are no faults, since the implementation in [50] does not consider faults. This comparison was only possible for rectangular architectures, a limitation of [50]. Under these simplifications (no faults and rectangular architectures), both **LSPR** and **FA-LSR** have the same implementation. Also, for a fair comparison, we ignored routing and we have used the same module library as in [50].

The results of this comparison are presented in Table 5.10. The deadlines for PCR, IVD and CPA are 10 s, 15 s and 100 s, respectively. Table 5.10 shows that our LS-based compilation is able to obtain good quality results using a much shorter runtime (milliseconds vs 1 hour). The average percentage deviation from the near-optimal result is 5.5%, hence, it can successfully be used for design space exploration.

Next, we wanted to determine the increase in $\delta_{\mathcal{G}}$ computed by **FA-LSR** as the number of permanent faults k increases. Table 5.11 shows the comparison between $\delta^k_{\mathcal{G}}$ for $k = 0$, 1 and 2. As an input to **LSPR** we have used an application-specific architecture, synthesized using our SA approach such that it minimizes the cost for each particular case-study and is tolerant to 2 faults. The cost of this architecture is presented in column 2 of Table 5.11, and the $\delta^k_{\mathcal{G}}$ results are in columns 3, 4 and 6, for k=0, 1 and 2, respectively. As we can see from Table 5.11, the increase in $\delta_{\mathcal{G}}$ is on average 11.8% for each increase in k.

Table 5.12: Evaluation of the CRM approach for compilation

App. (ops.*)	Arch.	$\delta_{\mathcal{G}}^{R}(s)$	$\delta_{\mathcal{G}}^{CRM}(s)$	Deviation (%)
IVD (23)	45 (2,2,2)	18.4	11.73	36
SB_2 (50)	96 (1,2,1)	29.39	23.9	18.6
SB_3 (70)	103 (2,2,2)	31.03	20.15	35
SB_4 (90)	125 (2,2,2)	42.51	27.87	34

*We ignored the detection operations for this set of experiments

In our final set of experiments we were interested to determine the efficiency of our proposed placement of operations (Section 5.4.2) in terms of the application completion time $\delta_{\mathcal{G}}^{CRM}$ obtained after compilation.We compared $\delta_{\mathcal{G}}^{CRM}$ to the completion time $\delta_{\mathcal{G}}^{R}$, obtained by using the routing-based compilation approach from [52], which is the only available synthesis approach that is not limited to rectangular modules and can take advantage of an application-specific architecture.

The results of this comparison are presented in Table 5.12. For the real-life application (IVD), we used the application-specific architecture (column 2) derived with our SA-based architecture synthesis. The application-specific architectures for the synthetic benchmarks were obtained manually. In column 2 we present, for each architecture, the number of electrodes and in parentheses the numbers of reservoirs for sample, buffer and reagent. As we can see from Table 5.12, our placement results in a better completion time $\delta_{\mathcal{G}}^{CRM}$ (column 4) than $\delta_{\mathcal{G}}^{R}$ (column 3) for all the tested benchmarks. For example, for IVD, we obtained a completion time $\delta_{\mathcal{G}}^{CRM} = 11.73$ s, improving with 36% the completion time $\delta_{\mathcal{G}}^{R} = 18.4$ s.

CHAPTER 6

Conclusions and Future work

The conclusions are presented in Section 6.1 and the future work in Section 6.2.

6.1 Conclusions

In this thesis we have proposed compilation methods to address transient faults during the application execution and synthesis methods for the design of application-specific architectures, aiming at fault-tolerance against permanent faults.

The first proposed compilation approach has considered only faults during split operations, which are erroneous if the resulted droplet volumes, detected using sensors, are outside of a given threshold. Recovery from faults is done by merging the droplets back and re-executing the split operation. We have used the compilation strategy from [49] to generate a binding and placement of operation (ignoring faults) and we have focused on generating good quality fault-tolerant schedules.

Hence, we have proposed a fault-tolerant sequencing graph that can capture all the fault scenarios in the application, considering that faults can happen only during split operations. Then, we have devised a scheduling technique, based on the List Scheduling (LS) algorithm, to derive the fault-tolerant schedule table, which contains the schedule for each fault scenario.

During the execution of the application, a detection operation is performed after each split operation. The current fault scenario is updated according to the results of the detection operation and the schedule corresponding to the updated fault scenario is selected from the fault-tolerant schedule table.

We have used two real-life applications and seven synthetic benchmarks to evaluate our proposed fault-tolerant compilation. As the experimental results show, by taking into account fault-occurrence information we can derive better quality schedules, which leads to shorter application completion times even in the worst-case fault scenario. This has the potential to reduce costs, because smaller area biochips and less sensors can be used to implement the application.

Next, we have extended our work to address faults in all types of operations. We have taken into account the parametric faults which can result in operation variability, such as volume variations. We have proposed a fault-tolerant compilation which uses an online Redundancy Optimization Strategy (ROS) for error recovery. The main features of ROS are that it uses a combination of time redundancy (re-executing operations) and space redundancy (producing redundant correct droplets before we know an error will occur), and is able to optimize at runtime their use based on the actual fault occurrences.

We have also proposed a biochemical application model which captures the detection operations needed to detect an error, and the operations that have to be executed for recovery. The error detection operations are introduced at runtime based on an error propagation analysis and the previous fault occurrences. We have developed a LS-based fast online compilation, which is able to exploit the biochip configuration at the moment when faults occur, such that the application completion times, even in case of errors, are minimized. The experiments performed on three real-life case studies show that our ROS can be successfully used to tolerate transient faults in time-sensitive biochemical applications.

In this thesis, we have also addressed the problem of synthesizing an application-specific biochip architecture that is fault-tolerant to permanent faults. Initially, we have proposed a solution based on Simulated Annealing (SA) metaheuristic. Every architecture visited by SA is evaluated in terms of application completion time by using a List Scheduling Placement and Routing (LSPR) heuristics, which performs a compilation of the application on the architecture. LSPR takes into account the worst-case overhead due to permanent faults and performs binding, scheduling, placement and routing evaluation. Our SA-based synthesis outputs the minimum cost architecture that can satisfy the deadline of the application even in the case of permanent faults.

The experimental results show that our synthesis approach is able to significantly reduce the costs compared to general-purpose rectangular architectures. In addition, by synthesizing fault-tolerant architectures, our methodology can help the designer increase the yield of DMBs.

Next, we have proposed an improved solution to the architecture synthesis problem, based on Tabu Search (TS) metaheuristics. In order to evaluate the architecture alternatives visited by TS in terms of their impact on the timing constraints of the application, we have proposed a Fault-Aware List Scheduling and Routing (FA-LSR) compilation to determine the application completion time which depends on the given architecture and on the pattern of permanent faults. FA-LSR uses an estimation method that tries to reduce the pessimism of the worst-case completion time evaluation determined by LSPR. Our previous SA-based architecture synthesis used the worst-case values for operation execution, which is too pessimistic, i.e., the architecture solution search avoided visiting potentially good low-cost architectures. By using an estimate that is less pessimistic than worst-case, we have extended the exploration of the solution space to architectures of lower cost. As the experiments show, FA-LSR proves to be fast and provides good quality solutions.

In in the context of the architecture synthesis problem we have investigated for an operation placement strategy that would make a better use of the irregular layout of an application specific biochip. Hence, we have proposed a placement strategy using circular-route modules that take advantage of the characteristics of the architecture and use effectively the available area. Our algorithm starts from an application-specific architecture, given as input, and builds a library of circular-route modules. The library provides multiple choices, that can be further exploited by the compilation implementation to minimize the application completion time. We have evaluated our proposed placement strategy, by comparing the compilation results with previous work, on several benchmarks. As the experimental results show, our placement strategy is able to significantly reduce the completion time of the applications.

We have integrated the proposed placement strategy into our TS-based architecture synthesis, i.e., we use the algorithm for building a library for the initial architecture solution generated by TS. For the solutions visited by TS during the search, we have proposed a faster method that incrementally updates the library previously determined. The library and the estimated operation execution times are used by FA-LSR to determine the application completion time in case of faults. Experiments show that our TS-based synthesis is able to find application-specific architectures with significantly lower-cost than the minimum-cost rectangular architectures.

6.2 Future work

In this section we present some of the directions in which our work can be extended to address other problems regarding the compilation of biochemical applications on digital microfluidic biochips.

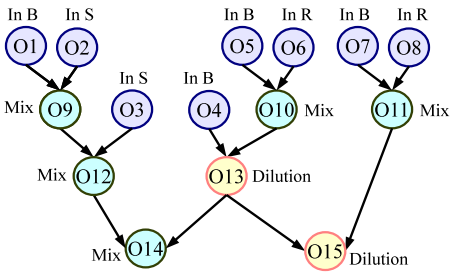


Figure 6.1: Example application $\mathcal{G}$

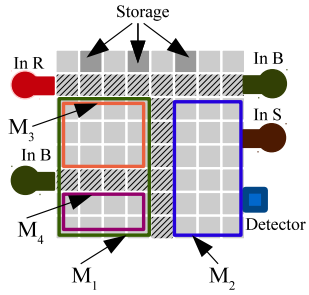


Figure 6.2: Placement of modules

All compilation strategies proposed so far in related research (the only exception is [46]) consider a given module library $\mathcal{L}$, which contains for each operation the worst-case execution time (*wcet*). However, an operation can finish before the *wcet*, due to variability and randomness in biochemical reactions [39, 42]. Such situations, when the actual execution time of the operation is less than *wcet*, result in time slacks in the schedule of operations. These time slacks can be used for executing other operations in the application and thus, reduce the completion time. The shorter the completion time, the more faults can be tolerated, since we can use the gained time to recover from transient faults. In this subsection, we are investigating alternative solutions for exploiting the slack time resulted due to uncertainties in operation execution, such that the application completion time is minimized.

The main assumption is that we can determine if and when an operation finishes before its *wcet*. For example, we can use during the execution of the bioassay a "sensing system" [30, 26], such as the CCD camera-based one, presented in Section 4.2.5, to determine the actual execution times of the operations.

Let us consider as example the application $\mathcal{G}$ in Fig. 6.1 with a deadline $d_{\mathcal{G}} = 13$ s and the *wcet* values from Table 6.1 for operations execution. As Fig. 6.3a shows, the application $\mathcal{G}$ is scheduled to finish executing in 13 s. In Fig. 6.3b, we assume that the operations O_2, O_9, O_{11} and O_{14} will finish in less than their respective *wcet*. The slack time for each of these operations is marked with gray hashed rectangles in Fig. 6.3b. As we can see from Fig. 6.3b, although the mentioned operations finished earlier, by using the same schedules as in Fig. 6.3b, we are not able to exploit the resulted slack. We are interested in a method which can produce a schedule such as the one in Fig. 6.6, which uses the actual execution time values for the operations. The challenge is that we do not know in advance, at design time, which operations will finish earlier and their execution times. These are only known at runtime, as detected by the available sensing system.

The only work that addresses the uncertainties in operation execution problem is [46],

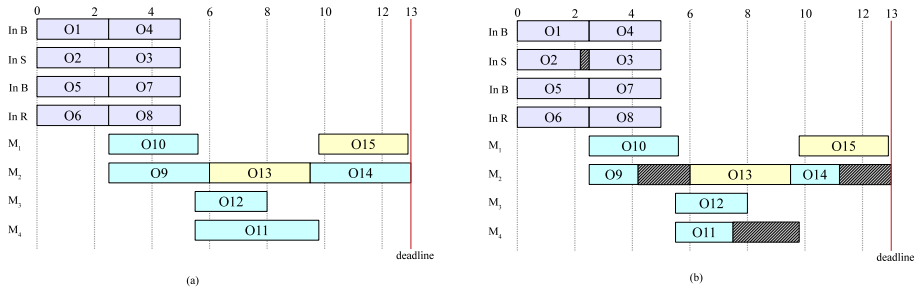


Figure 6.3: Schedule using *wcet* values for operation execution

Table 6.1: Module library $\mathcal{L}$ with *wcet* values

Operation	Module area	*wcet* (s)
In	N/A	2.5
Mix	3×6	3.47
Mix	4×6	3.1
Mix	4×2	4.3
Mix	4×3	2.5
Dilution	3×6	4
Dilution	4×6	3.1
Detection	1×1	5

where an Operation-Interdependency-Aware (OIA) compilation is proposed to derive an offline schedule that is scaled at runtime in case operations do not finish at expected time, i.e., *wcet*. As an alternative to OIA we propose two other approaches: (i) an online compilation method (Section 6.2.2) which is used to update the schedule at runtime when we detect that an operation finishes before its *wcet*, and (ii) an offline compilation method (Section 6.2.3), based on a "quasi-static scheduling" approach which derives offline a set of schedules, considering varying execution times, and switches online to the most appropriate schedule corresponding to the observed execution times.

The strategies employed in each of these approaches are depicted in Fig. 6.4a, b and c, respectively. We discuss in detail these strategies in the following three subsections.

In order to illustrate each of the strategies we will use as example the application graph $\mathcal{G}$ from Fig. 6.1, which has to execute on the 8×8 biochip $\mathcal{A}$ in Fig. 6.2. The deadline for $\mathcal{G}$ is $d_{\mathcal{G}} = 13\ s$. We consider that the operations are executing on rectangular modules which have their area and *wcet* specified in the library $\mathcal{L}$ from Table 6.1. The placement of the modules is presented in Fig. 6.2. In this example, we assume a black-box approach for the operation execution, hence we use a one-thickness segregation border between modules in order to avoid accidental merging (see the placement of

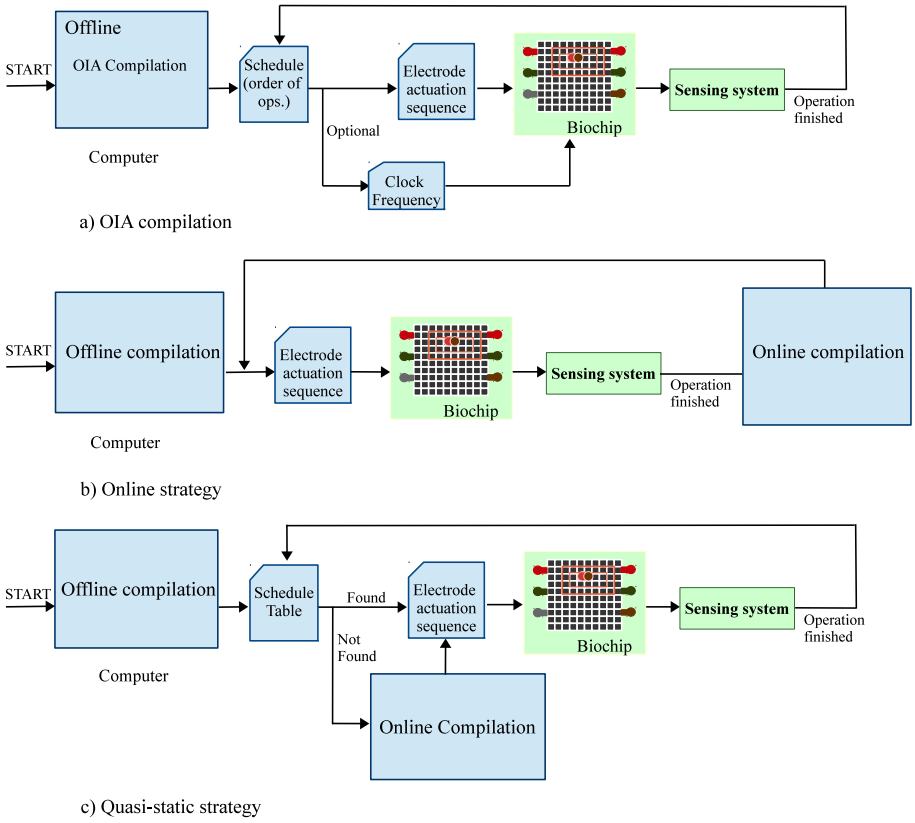


Figure 6.4: Solutions to the problem of uncertainties in operation execution

modules in Fig. 6.2, where the hashed electrodes represent the segregation border). As in the previous examples in this thesis, we ignore routing for simplicity reasons.

We formulate the problem as follows. Given an application $\mathcal{G}$ to be executed on a rectangular biochip architecture $\mathcal{A}$ within a deadline $d_{\mathcal{G}}$, we want a schedule $\mathcal{S}$ of operations which minimizes $\delta_{\mathcal{G}}$ in case of uncertainties.

6.2.1 OIA strategy

A solution to this problem is the OIA strategy proposed in [46] and schematically presented in Fig. 6.4a. As seen in Fig. 6.4a, OIA compilation is used offline to determine the schedule of operations. At runtime, a sensing system is used to signal when an operation has finished executing. The schedule is scaled accordingly to adjust to the

actual execution times of the operations. Note that OIA does not use a module library as input, but instead relies on the sensing system to notify when an operation finished executing. Hence, the schedule decided offline by OIA contains only the order of operations and does not contain the execution times of the operations.

The OIA compilation in [46] has four steps:

(1) First, the application graph $\mathcal{G}$ is partitioned into multiple directed trees by determining the operations in $\mathcal{G}$ with more than one successor and removing all the edges that start from those operations. For the graph in Fig. 6.1, the only operation in $\mathcal{G}$ that has more than one successor is O_{13}. After removing all edges that have O_{13} as source, we obtain the three trees $\mathcal{T}_L$, $\mathcal{T}_C$ and $\mathcal{T}_R$, depicted in Fig. 6.5a.

(2) Next, the OIA compilation is applied to each of the trees obtained at step 1. OIA compilation schedules the operations in phases, namely the transport (T) phase for routing and dispensing operations and the dilution/mixing (D/M) phase for dilution and mixing operations. Each phase executes until all the operations that are part of it are completed. The two phases, T and D/M, alternate with only one being active at a time. One of the advantages of this scheduling approach, is that, in case a biochip with multiple clock frequency is used, then the clock frequency during the T phase can be increased and thus reduce the application completion time with no side-effects (the electrodes deteriorate faster with the number of frequency switches, but in the T phase there is a constant number of switches regardless of the frequency). The schedules obtained using OIA for the three trees $\mathcal{T}_L$, $\mathcal{T}_C$ and $\mathcal{T}_R$ are presented in Fig. 6.5c–e. As mentioned, OIA determines only the order of the operations and not their duration, i.e., the start and finish times. Hence, in order to depict the schedule of the operations in Fig. 6.5d–f, for clarity reasons, we used the actual execution times of the operations, as if they were determined at runtime.

(3) The directed trees are sorted so that they do not present scheduling and placement conflicts. Using the sorting algorithm proposed in [46], we obtained for our example the sorted order in Fig. 6.5b.

(4) Finally, the compilation results for the trees are merged according to the sorted order obtained during the previous step. For our example, we have obtained the final schedule from Fig. 6.5f, where the execution of the application starts with a T phase, containing operations O_5 and O_6. When both operations finish executing, the next phase, containing operation O_{10} starts. Each phase waits for all operations scheduled in the previous phase to finish executing. The order of the operations is fixed, i.e., it is not influenced by any variability in operation execution.

Hence, OIA compilation provides a schedule that is scalable at runtime and does not depend on a module library or an online compilation. Although OIA is able to handle operation execution variability, its aim is not to reduce the schedule length, but to han-

dle situations when a module library is not available and there is no information about the *wcet* of operations. As seen in Fig. 6.5f, the application does not meet the required deadline $d_{\mathcal{G}} = 13\ s$.

6.2.2 Online compilation under execution time uncertainty

In Section 4.2.2 we have proposed an online compilation strategy to tolerate transient faults at runtime. Here we discuss how that strategy can be adapted for handling operation execution time variability. Here we do not, though, address the issue of faults, which is orthogonal to our problem.

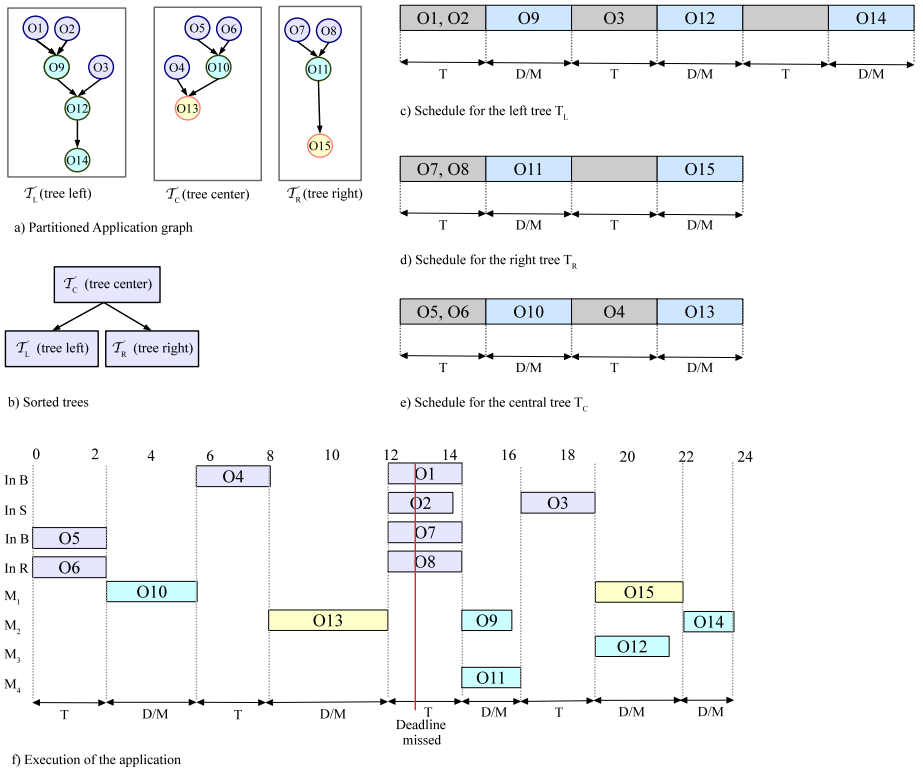


Figure 6.5: OIA compilation example

A second solution to the problem is an online approach, which, whenever an operation finishes at a different time than expected, uses an online compilation to determine a new schedule for the current scenario. The online approach, outlined in Fig. 6.4b, is similar to out proposed online error recovery strategy, presented in Section 4.2.2. As seen in Fig. 6.4b, we use an offline compilation to determine an initial schedule. We consider that we have given a module library containing the *wcet* of operations. Next, the application is executed according to the offline schedule. A sensing system notifies the computer whenever an operation finishes earlier than its *wcet*. Then, an online re-compilation is run to determine a new corresponding schedule. Fig. 6.3a shows the schedule of operations determined offline using *wcet*.

As shown in Fig. 6.3, operation O_2 is scheduled to finish at 2.5 s. However, when executed on the biochip, operation O_2 finishes earlier than expected, at $t = 2.2\ s$. Consequently, at $t = 2.2\ s$, the computer executes an online re-compilation to determine the new schedule of operations. In Fig. 6.6, the thick vertical lines on the row labeled "Computer", mark the re-compilation time, which is negligible in comparison to the operation execution times. When using the online strategy, the application in our example completes in 12.7 s, which is faster than using the *wcet* values (see the schedule in Fig. 6.6).

6.2.3 Quasi-static scheduling

As observed, an online strategy is able to handle uncertainties in operation execution and complete the application within deadline at the expense of a runtime overhead due to online compilation. However, an offline strategy, such as OIA compilation, is preferred when an online strategy cannot be used, e.g., the microcontroller is too slow, the DMB does not have an integrated sensing system, etc.

In this subsection, we present the third strategy, based on quasi-static scheduling, which

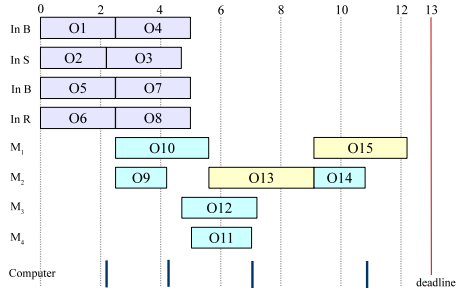


Figure 6.6: Execution of operations using the online strategy

determines an offline a set of of schedule tables from which a particular schedule will be chosen at runtime, corresponding to the current operation execution scenario. The quasi-static scheduling techniques have been previously proposed for real-time systems [16], in the context of scheduling tasks with soft deadlines on multiprocessors.

Let us explain the quasi-static scheduling technique using as example the application $\mathcal{G}_1$ in Fig. 6.7a with a deadline $d_{\mathcal{G}_1} = 40s$. The quasi-static scheduling technique derives the tree of schedules in which the nodes are alternative schedules of operations and the edges are conditioned by the current execution scenario. We used for operation execution the *wcet* values from the library in Table 6.2 and determined for our example the schedule in Fig. 6.7b. The binding of operations is depicted at the left side of the scheduling in Fig. 6.7b. For example, operations O_2, O_5 and O_8 execute on module M_1, which has a size of 3×3 electrodes.

The tree of schedules derived for the application $\mathcal{G}_1$ is depicted in Fig. 6.7c, where the condensed notation for the schedule alternatives represents the order of operations on modules M_1, M_2 and detector, respectively. For example, the root of the tree of schedules in Fig. 6.7c is the schedule alternative from Fig. 6.7b, where operations O_2, O_5 and O_8 execute on module M_1, operations O_1, O_3, O_4, O_7 and O_6 execute on M_2 and operations O_9, O_{10} and O_{11} execute on the detector.

We denote with $\mathcal{S} = \{O_2O_5O_8; O_1O_3O_4O_7O_6; O_9O_{10}O_{11}\}$—the schedule that is the root of the tree of schedules. Thus operations O_1 and O_2 start executing at the same time as $t = 0$ on modules M_1 and M_2, respectively. The module library $\mathcal{L}$ provides the *bcet* and *wcet* for each type of operation. It follows that the completion time intervals for O_1 and O_2 are [4, 7] and [1, 5], respectively. Hence, we consider two cases: (a) O_1 finishes before O_2 and (b) O_1 finishes after O_2. The completion time intervals for O_2 are (7, 10] in case (a) and [5, 7] in case (b). In the tree of schedules the execution scenarios are represented as conditions on the edges, and the alternative scenarios are nodes. As seen in the tree of schedules (Fig. 6.7c), in case (a), corresponding to the edge labeled "O_2:$(7,10]$", the schedule $\mathcal{S}' = \{O_2O_5O_8; O_1O_3O_4O_7O_6; O_9O_{10}O_{11}\}$ is derived. In case (b), corresponding to the edge labeled "O_2:$[5,7]$", the schedule $\mathcal{S}'' = \{O_2O_5O_8; O_1O_3O_4O_6O_7; O_{10}O_9O_{11}\}$ is obtained.

Table 6.2: Module library $\mathcal{L}$ for quasi-static scheduling

Operation	Module area	*bcet* (s)	*wcet* (s)
Mix	3×3	4	7
Mix	1×3	1	5
Dilution	3×3	5	10
Dilution	1×3	3	7
Detection	1×1	5	5

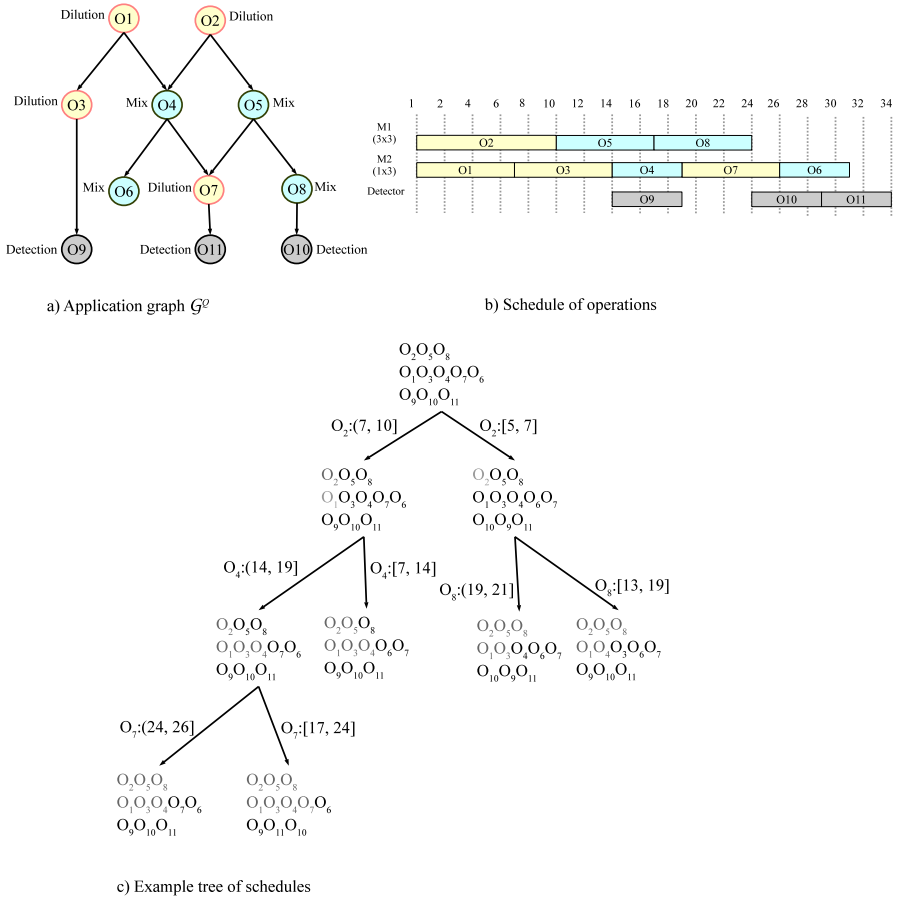


Figure 6.7: Quasi-static scheduling example

Such a tree of schedules, derived statically, captures a number of schedules and switching points. At runtime, the current execution scenario activates a specific path in the tree of schedules. The corresponding schedule alternative is loaded. Hence, the quasi-static scheduling technique has the advantage of deriving schedules that satisfy the timing constraints of the application without the runtime overhead of an online compilation. We consider for future work implementing the online strategy and the quasi-static scheduling technique in order to draw a comparison between them.

Bibliography

[1] Mirela Alistar, Elena Maftei, Paul Pop, and Jan Madsen. Synthesis of biochemical applications on digital microfluidic biochips with operation variability. In *Proceedings of the Symposium on Design Test Integration and Packaging of MEMS/MOEMS*, pages 350–357, 2010.

[2] Mirela Alistar, Paul Pop, and Jan Madsen. Redundancy optimization for error recovery in digital microfluidic biochips. Under review in *Design Automation for Embedded Systems*.

[3] Mirela Alistar, Paul Pop, and Jan Madsen. Synthesis of application-specific fault-tolerant digital microfluidic biochips architectures. Submitted to *Journal on Emerging Technologies in Computing Systems*.

[4] Mirela Alistar, Paul Pop, and Jan Madsen. Online synthesis for error recovery in digital microfluidic biochips with operation variability. In *Proceedings of the Symposium on Design, Test, Integration and Packaging of MEMS/MOEMS*, pages 53–58, 2012.

[5] Mirela Alistar, Paul Pop, and Jan Madsen. Application-specific fault-tolerant architecture synthesis for digital microfluidic biochips. In *Proceedings of the 18th Asia and South Pacific Design Automation Conference*, pages 794–800, 2013.

[6] Mirela Alistar, Paul Pop, and Jan Madsen. Operation placement for application-specific digital microfluidic biochips. In *Proceedings of the Symposium on Design, Test, Integration and Packaging of MEMS/MOEMS*, pages 1–6, 2013.

[7] Advanced Liquid Logic. `http://www.liquid-logic.com`.

[8] Vaishnavi Ananthanarayanan and William Thies. Biocoder: A programming language for standardizing and automating biology protocols. *Journal of biological engineering*, 4(1):1–13, 2010.

[9] Ismail Emre Araci and Philip Brisk. Recent developments in microfluidic large scale integration. *Current Opinion in Biotechnology*, 25:60–68, 2014.

[10] Kiarash Bazargan, Ryan Kastner, and Majid Sarrafzadeh. Fast template placement for reconfigurable computing systems. *IEEE Design and Test of Computers*, 17(1):68–83, 2000.

[11] Karl F. Böhringer. Modeling and controlling parallel tasks in droplet-based microfluidic systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25(2):334–344, 2006.

[12] Edmund K Burke and Graham Kendall. *Search methodologies: introductory tutorials in optimization and decision support techniques*. Springer, 2005.

[13] Krishnendu Chakrabarty, Richard B Fair, and Jun Zeng. Design tools for digital microfluidic biochips: toward functional diversification and more than Moore. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 29(7):1001–1017, 2010.

[14] Krishnendu Chakrabarty and Fei Su. *Digital microfluidic biochips: synthesis, testing, and reconfiguration techniques*. CRC Press, 2006.

[15] Minsik Cho and David Z. Pan. A high-performance droplet routing algorithm for digital microfluidic biochips. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(10):1714–1724, 2008.

[16] Luis Alejandro Cortés. *Verification and scheduling techniques for real-time embedded systems*. PhD thesis, Linköping University, Linköping, Sweden, 2005.

[17] Philippe Dubois. *Les microréacteurs en gouttes de liquides ioniques: génération, manipulation par électromouillage sur isolant et utilisation en synthèse*. PhD thesis, University of Rennes 1, Rennes, France, 2007.

[18] Shimon Even. An algorithm for determining whether the connectivity of a graph is at least *k*. *SIAM Journal on Computing*, 4(3):393–396, 1975.

[19] Richard B. Fair. Digital microfluidics: is a true lab-on-a-chip possible? *Microfluidics and Nanofluidics*, 3(3):245–281, 2007.

[20] Richard B. Fair. Biochip engineering. University Lecture, 2008.

[21] Thomas A. Feo and Mauricio G.C. Resende. Greedy randomized adaptive search procedures. *Journal of global optimization*, 6(2):109–133, 1995.

[22] Yves Fouillet, Dorothée Jary, Claude Chabrol, Patricia Claustre, and Christine Peponnet. Digital microfluidic design and optimization of classic and new fluidic functions for lab on a chip systems. *Microfluidics and Nanofluidics*, 4(3):159–165, 2008.

[23] Fred Glover and Manuel Laguna. *Tabu Search*. Kluwer Academic Publishers, 1997.

[24] Jian Gong, Shih-Kang Fan, Chang-Jin Kim, et al. Portable digital microfluidics platform with active but disposable lab-on-chip. In *Proceedings of the 17th IEEE International Conference on Micro Electro Mechanical Systems (MEMS)*, pages 355–358, 2004.

[25] Jian Gong and Chang-Jin Kim. Two-dimensional digital microfluidic system by multilayer printed circuit board. In *Proceedings of the 18th IEEE International Conference on Micro Electro Mechanical Systems (MEMS)*, pages 726–729, 2005.

[26] Jian Gong and Chang-Jin Kim. All-electronic droplet generation on-chip with real-time feedback control for EWOD digital microfluidics. *Lab on a Chip*, 8(6):898–906, 2008.

[27] Daniel Grissom and Philip Brisk. Fast online synthesis of generally programmable digital microfluidic biochips. In *Proceedings of the 8th International Conference on Hardware/Software codesign and System Synthesis*, pages 413–422, 2012.

[28] Daniel Grissom and Philip Brisk. A high-performance online assay interpreter for digital microfluidic biochips. In *Proceedings of the Great Lakes symposium on VLSI*, pages 103–106, 2012.

[29] Daniel Grissom and Philip Brisk. Path scheduling on digital microfluidic biochips. In *Proceedings of the 49th Annual Design Automation Conference*, pages 26–35, 2012.

[30] B Hadwen, GR Broder, D Morganti, A Jacobs, C Brown, JR Hector, Y Kubota, and H Morgan. Programmable large area digital microfluidic array with integrated droplet sensing for bioassays. *Lab on a Chip*, 12(18):3305–3313, 2012.

[31] Tsung-Yi Ho, Jun Zeng, and Krishnendu Chakrabarty. Digital microfluidic biochips: A vision for functional diversity and more than Moore. In *Proceedings of the International Conference on Computer-Aided Design*, pages 578–585, 2010.

[32] Yi-Ling Hsieh, Tsung-Yi Ho, and Krishnendu Chakrabarty. Design methodology for sample preparation on digital microfluidic biochips. In *Proceedings of the 30th International Conference on Computer Design*, pages 189–194, 2012.

[33] Yi-Ling Hsieh, Tsung-Yi Ho, and Krishnendu Chakrabarty. A reagent-saving mixing algorithm for preparing multiple-target biochemical samples using digital microfluidics. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 31(11):1656–1669, 2012.

[34] Kai Hu, Bang-Ning Hsu, Andrew Madison, Krishnendu Chakrabarty, and Richard B. Fair. Fault detection, real-time error recovery, and experimental demonstration for digital microfluidic biochips. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 559–564, 2013.

[35] Juinn-Dar Huang, Chia-Hung Liu, and Ting-Wei Chiang. Reactant minimization during sample preparation on digital microfluidic biochips using skewed mixing trees. In *Proceedings of the International Conference on Computer-Aided Design*, pages 377–383, 2012.

[36] Tsung-Wei Huang and Tsung-Yi Ho. A fast routability-and performance-driven droplet routing algorithm for digital microfluidic biochips. In *Proceedings of the International Conference on Computer Design*, pages 445–450, 2009.

[37] Tsung-Wei Huang, Chun-Hsien Lin, and Tsung-Yi Ho. A contamination aware droplet routing algorithm for digital microfluidic biochips. In *Proceedings of the International Conference on Computer-Aided Design*, pages 151–156, 2009.

[38] Tsung-Wei Huang, Shih-Yuan Yeh, and Tsung-Yi Ho. A network-flow based pin-count aware routing algorithm for broadcast-addressing EWOD chips. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(12):1786–1799, 2011.

[39] M Iyengar and Mary McGuire. Imprecise and qualitative probability in systems biology. In *Proceedings of the International Conference on Systems Biology*, 2007.

[40] Viacheslav Izosimov, Paul Pop, Petru Eles, and Zebo Peng. Synthesis of fault-tolerant schedules with transparency/performance trade-offs for distributed embedded systems. In *Proceedings of the conference on Design, Automation and Test in Europe*, pages 706–711, 2006.

[41] Simeon Oloni Kotchoni, Emma Wanjiru Gachomo, Eriola Betiku, and Olusola Olusoji Shonukan. A home made kit for plasmid DNA mini-preparation. *African Journal of Biotechnology*, 2(4):109–114, 2003.

[42] Octave Levenspiel. *Chemical reaction engineering*. Wiley New York, 1972.

[43] Yan Luo, Bhargab B Bhattacharya, Tsung-Yi Ho, and Krishnendu Chakrabarty. Optimization of polymerase chain reaction on a cyberphysical digital microfluidic biochip. In *Proceedings of the International Conference on Computer-Aided Design*, pages 622–629, 2013.

[44] Yan Luo, Krishnendu Chakrabarty, and T-Y Ho. Real-time error recovery in cyberphysical digital-microfluidic biochips using a compact dictionary. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 32(12):1839–1852, 2013.

[45] Yan Luo, Krishnendu Chakrabarty, and Tsung-Yi Ho. A cyberphysical synthesis approach for error recovery in digital microfluidic biochips. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 1239–1244, 2012.

[46] Yan Luo, Krishnendu Chakrabarty, and Tsung-Yi Ho. Design of cyberphysical digital microfluidic biochips under completion-time uncertainties in fluidic operations. In *Proceedings of the 50th Annual Design Automation Conference*, page 44, 2013.

[47] Yan Luo, Krishnendu Chakrabarty, and Tsung-Yi Ho. Error recovery in cyberphysical digital microfluidic biochips. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 32(1):59–72, 2013.

[48] Elena Maftei. *Synthesis of Digital Microfluidic Biochips with Reconfigurable Operation Execution*. PhD thesis, Technical University of Denmark, 2011.

[49] Elena Maftei, Paul Pop, and Jan Madsen. Tabu search-based synthesis of dynamically reconfigurable digital microfluidic biochips. In *Proceedings of the 2009 international conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 195–204, 2009.

[50] Elena Maftei, Paul Pop, and Jan Madsen. Tabu search-based synthesis of digital microfluidic biochips with dynamically reconfigurable non-rectangular devices. *Design Automation for Embedded Systems*, 14(3):287–307, 2010.

[51] Elena Maftei, Paul Pop, and Jan Madsen. Droplet-aware module-based synthesis for fault-tolerant digital microfluidic biochips. In *Proceedings of the Symposium on Design, Test, Integration and Packaging of MEMS/MOEMS*, pages 47–52, 2012.

[52] Elena Maftei, Paul Pop, and Jan Madsen. Routing-based synthesis of digital microfluidic biochips. *Design Automation for Embedded Systems*, 16(1):19–44, 2012.

[53] Elena Maftei, Paul Pop, and Jan Madsen. Module-based synthesis of digital microfluidic biochips with droplet-aware operation execution. *Journal on Emerging Technologies in Computing Systems*, 9(1):2, 2013.

[54] Elena Maftei, Paul Pop, Jan Madsen, and Thomas K Stidsen. Placement-aware architectural synthesis of digital microfluidic biochips using ILP. In *Proceedings of the International Conference on Very Large Scale Integration*, pages 425–430, 2008.

[55] Daniel Mark, Stefan Haeberle, Günter Roth, Felix von Stetten, and Roland Zengerle. Microfluidic lab-on-a-chip platforms: requirements, characteristics and applications. *Chemical Society Reviews*, 39(3):1153–1182, 2010.

[56] Rajendrani Mukhopadhyay. Microfluidics: on the slope of enlightenment. *Analytical chemistry*, 81(11):4169–4173, 2009.

[57] Nugen technologies. `http://www.nugen.com`.

[58] Phil Paik, Vamsee K Pamula, and Richard B Fair. Rapid droplet mixers for digital microfluidic systems. *Lab on a Chip*, 3(4):253–259, 2003.

[59] M. G. Pollack. *Electrowetting-based microactuation of droplets for digital microfluidics*. PhD thesis, Duke University, Durham, NC, 2001.

[60] M. G. Pollack, A. D. Shenderov, and Richard B. Fair. Electrowetting-based actuation of droplets for integrated microfluidics. *Lab on Chip*, 2:96–101, 2002.

[61] Hong Ren and Richard B Fair. Micro/nano liter droplet formation and dispensing by capacitance metering and electrowetting actuation. In *Proceedings of the 2nd Conference on Nanotechnology*, pages 369–372, 2002.

[62] Hong Ren, Vijay Srinivasan, and Richard B Fair. Design and testing of an interpolating mixing architecture for electrowetting-based droplet-on-chip chemical dilution. In *Proceedings of the 12th International Conference on Transducers, Solid-State Sensors, Actuators and Microsystems*, pages 619–622, 2003.

[63] Andrew J Ricketts, Kevin Irick, Narayanan Vijaykrishnan, and Mary Jane Irwin. Priority scheduling in digital microfluidics-based biochips. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 329–334, 2006.

[64] Don Rose. Microdispensing technologies in drug discovery. *Drug discovery today*, 4(9):411–419, 1999.

[65] Sudip Roy, Bhargab B Bhattacharya, and Krishnendu Chakrabarty. Waste-aware dilution and mixing of biochemical samples with digital microfluidic biochips. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 1–6, 2011.

[66] Sadiq M Sait and Habib Youssef. *VLSI physical design automation: theory and practice*. World Scientific, 1999.

[67] Oliver Sinnen. *Task scheduling for parallel systems*. John Wiley & Sons, 2007.

[68] Ramakrishna Sista, Zhishan Hua, Prasanna Thwar, Arjun Sudarsan, Vijay Srinivasan, Allen Eckhardt, Michael Pollack, and Vamsee Pamula. Development of a digital microfluidic platform for point of care testing. *Lab on a Chip*, 8(12):2091–2104, 2008.

[69] Ramakrishna S Sista, Allen E Eckhardt, Tong Wang, Carrie Graham, Jeremy L Rouse, Scott M Norton, Vijay Srinivasan, Michael G Pollack, Adviye A Tolun, Deeksha Bali, et al. Digital microfluidic platform for multiplexing enzyme assays: implications for lysosomal storage disease screening in newborns. *Clinical chemistry*, 57(10):1444–1451, 2011.

[70] JH Song, R Evans, Y-Y Lin, B-N Hsu, and RB Fair. A scaling model for electrowetting-on-dielectric microfluidic actuators. *Microfluidics and Nanofluidics*, 7(1):75–89, 2009.

[71] V. Srinivasan, V. K. Pamula, M. Pollack, and R. B. Fair. A digital microfluidic biosensor for multianalyte detection. pages 327–330, 2003.

[72] Vijay Srinivasan, Vamsee K Pamula, and Richard B Fair. An integrated digital microfluidic lab-on-a-chip for clinical diagnostics on human physiological fluids. *Lab on a Chip*, 4(4):310–315, 2004.

[73] Fei Su and Krishnendu Chakrabarty. Architectural-level synthesis of digital microfluidics-based biochips. In *Proceedings of the International Conference on Computer Aided Design*, pages 223–228, 2004.

[74] Fei Su and Krishnendu Chakrabarty. Design of fault-tolerant and dynamically-reconfigurable microfluidic biochips. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 1202–1207, 2005.

[75] Fei Su and Krishnendu Chakrabarty. Unified high-level synthesis and module placement for defect-tolerant microfluidic biochips. In *Proceedings of the 42nd annual Design Automation Conference*, pages 825–830, 2005.

[76] Fei Su and Krishnendu Chakrabarty. Benchmarks for digital microfluidic biochip design and synthesis. *Duke University Department ECE*, 2006.

[77] Fei Su and Krishnendu Chakrabarty. Module placement for fault-tolerant microfluidics-based biochips. *Transactions on Design Automation of Electronic Systems*, 11(3):682–710, 2006.

[78] Fei Su and Krishnendu Chakrabarty. Yield enhancement of reconfigurable microfluidics-based biochips using interstitial redundancy. *Journal on Emerging Technologies in Computing Systems*, 2(2):104–128, 2006.

[79] Fei Su and Krishnendu Chakrabarty. High-level synthesis of digital microfluidic biochips. *Journal on Emerging Technologies in Computing Systems*, 3(4):1, 2008.

[80] Fei Su, William Hwang, and Krishnendu Chakrabarty. Droplet routing in the synthesis of digital microfluidic biochips. In *Proceedings of the Conference on Design, Automation and Test in Europe*.

[81] Fei Su, William Hwang, Arindam Mukherjee, and Krishnendu Chakrabarty. Testing and diagnosis of realistic defects in digital microfluidic biochips. *Journal of Electronic Testing*, 23(2-3):219–233, 2007.

[82] Fei Su, Sule Ozev, and Krishnendu Chakrabarty. Concurrent testing of droplet-based microfluidic systems for multiplexed biomedical assays. In *Proceedings of the International Test Conference*, pages 883–892, 2004.

[83] Fei Su, Sule Ozev, and Krishnendu Chakrabarty. Ensuring the operational health of droplet-based microelectrofluidic biosensor systems. *Sensors Journal*, 5(4):763–773, 2005.

[84] John Robert Taylor. *An introduction to error analysis: the study of uncertainties in physical measurements*. University science books, 1997.

[85] U Twisselmann. Cutting rectangles avoiding rectangular defects. *Applied mathematics letters*, 12(6):135–138, 1999.

[86] Tao Xu and Krishnendu Chakrabarty. Parallel scan-like test and multiple-defect diagnosis for digital microfluidic biochips. *Transactions on Biomedical Circuits and Systems*, 1(2):148–158, 2007.

[87] Tao Xu and Krishnendu Chakrabarty. Integrated droplet routing and defect tolerance in the synthesis of digital microfluidic biochips. *Journal on Emerging Technologies in Computing Systems*, 4(3):11, 2008.

[88] Tao Xu and Krishnendu Chakrabarty. Fault modeling and functional test methods for digital microfluidic biochips. *Transactions on Biomedical Circuits and Systems*, 3(4):241–253, 2009.

[89] Jun-Ichi Yoshida. Flash chemistry: flow microreactor synthesis based on high-resolution reaction time control. *The Chemical Record*, 10(5):332–341, 2010.

[90] Ping-Hung Yuh, Chia-Lin Yang, and Yao-Wen Chang. Placement of defect-tolerant digital microfluidic biochips using the T-tree formulation. *Journal on Emerging Technologies in Computing Systems*, 3(3):13, 2007.

[91] Ping-Hung Yuh, Chia-Lin Yang, and Yao-Wen Chang. Bioroute: A network-flow-based routing algorithm for the synthesis of digital microfluidic biochips. *Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(11):1928–1941, 2008.

[92] Yang Zhao and Krishnendu Chakrabarty. Synchronization of washing operations with droplet routing for cross-contamination avoidance in digital microfluidic biochips. In *Proceedings of the 47th Design Automation Conference*, pages 635–640, 2010.

[93] Yang Zhao and Krishnendu Chakrabarty. Cross-contamination avoidance for droplet routing. In *Design and Testing of Digital Microfluidic Biochips*, pages 27–55. 2013.

[94] Yang Zhao, Krishnendu Chakrabarty, Ryan Sturmer, and Vamsee K Pamula. Optimization techniques for the synchronization of concurrent fluidic operations in pin-constrained digital microfluidic biochips. *Transactions on Very Large Scale Integration (VLSI) Systems*, 20(6):1132–1145, 2012.

[95] Yang Zhao, Tao Xu, and Krishnendu Chakrabarty. Integrated control-path design and error recovery in the synthesis of digital microfluidic lab-on-chip. *Journal on Emerging Technologies in Computing Systems*, 6(3):11, 2010.