DTIP 5-7 May 2010, Seville, Spain

Synthesis of Biochemical Applications on Digital Microfluidic Biochips with Operation Variability

Mirela Alistar, Elena Maftei, Paul Pop, Jan Madsen
Technical Univ. of Denmark, DK-2800 Kgs. Lyngby
email: mali@imm.dtu.dk

***Abstract*—Microfluidic-based biochips are replacing the conventional biochemical analyzers, and are able to integrate on-chip all the necessary functions for biochemical analysis using microfluidics. The digital microfluidic biochips are based on the manipulation of liquids not as a continuous flow, but as discrete droplets. Researchers have presented approaches for the synthesis of digital microfluidic biochips, which, starting from a biochemical application and a given biochip architecture, determine the allocation, resource binding, scheduling and placement of the operations in the application. Existing approaches consider that on-chip operations, such as splitting a droplet of liquid, are perfect. However, these operations have variability margins, which can impact the correctness of the biochemical application. We consider that a split operation, which goes beyond specified variability bounds, is faulty. The fault is detected using on-chip volume sensors. We have proposed an abstract model for a biochemical application, consisting of a sequencing graph, which can capture all the fault scenarios in the application. Starting from this model, we have proposed a synthesis approach that, for a given chip area and number of sensors, can derive a fault-tolerant implementation. Two fault-tolerant scheduling techniques have been proposed and compared. 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. The proposed synthesis approach under operation variability has been evaluated using several benchmarks.**

I. Introduction

Microfluidic biochips represent a promising alternative to conventional biochemical laboratories, and are able to integrate on-chip all the necessary functions for biochemical analysis using microfluidics, such as: transporting, splitting, merging, dispensing, mixing, and detection [2]. Some of the immediate advantages include: higher sensitivity, less likehood of human error due to automation, miniaturized size in comparison to the traditional laboratory equipment, lower price due to usage of small volumes of sample and reagent substances. Applications of biochips include: clinical diagnosis, drug discovery, DNA analysis (e.g., polymerase chain reaction and nucleic acid sequence analysis), protein and enzyme analysis and immunoassays [2].

The "digital microfluidic" biochips (DMBs) are based on the manipulation of liquids not as a continuous flow, but as discrete droplets (hence the term "digital") and are highly reconfigurable and scalable. A DMB is modeled as a two-dimensional array of cells, where each cell can hold a droplet, see Fig. 1b.

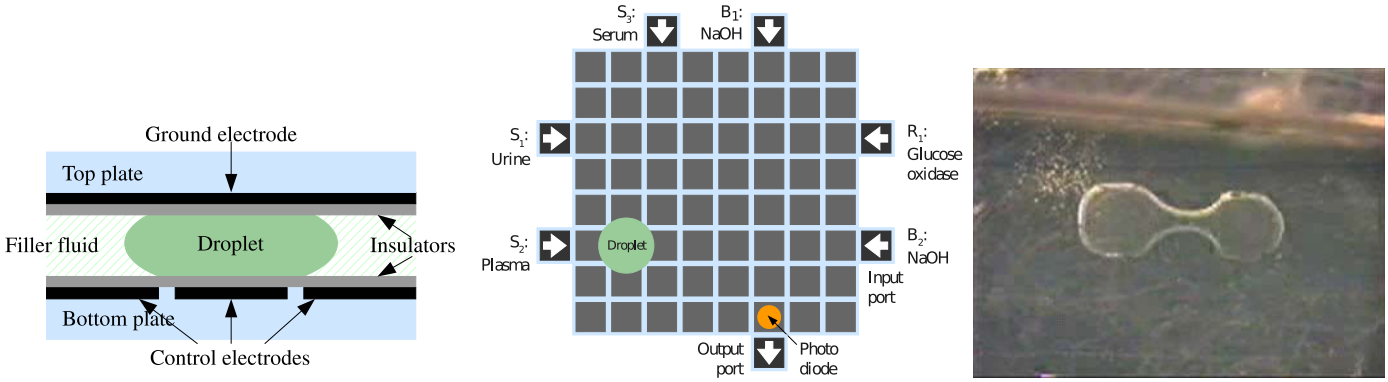
Considering their architecture and the design tasks that have to be performed, the design of digital microfluidic biochips has similarities to high-level synthesis of VLSI systems. Motivated by this similarity, researchers have started to propose approaches for the top-down design of such biochips. The following are the main design tasks that have been addressed [2]:

- During the design of a digital microfluidic biochip, the bioassay protocols have to be mapped to the on-chip modules. The protocols are (i) *modeled* using process graph models, where each node is an operation, and each edge represents a dependency [2].
- Once the protocol has been specified, the necessary modules for the implementation of the protocol operations will be selected from a module library. This is called the (ii) *allocation* step [3].
- As soon as the (iii) *binding* of operations to the allocated modules is decided, the (iv) *scheduling* step determines the time duration for each bioassay operation, subject to resource constraints and precedence constraints imposed by the protocol [4].
- Finally, the chip will be synthesized according to the constraints on the types of resources, cost, area and protocol completion times. During the chip synthesis, the (v) *placement* [5] of each module on the microfluidic array and the (vi) *routing* of droplets from one module to another have to be determined [6] [7].

All of the presented design tasks have to take into account possible defects during the fabrication of the microfluidic biochip. Thus, *testing* [8] [9] and *reconfiguration* [10] have to be performed.

Although researchers have addressed fabrication faults, the current research assumes that on-chip operations, such as splitting a droplet of liquid are fault-free. However, the reality is that these operations have variability margins, which can impact the correctness of the biochemical application. Moreover, because of the complex bioassays performed on biochips, the cells are many times reconfigured and used for different operations, which can lead to the situation where a group of cells fail to function correctly during the operation. We consider that an operation which goes beyond specified variability bounds is faulty (an example of a faulty split operation is presented in Fig. 1c). Error detection is done by routing the droplet to volume sensors and sensing the droplet volume [11]. [12] is the only work so far that has addressed

 ISBN: 978-2-35500-011-9



(a) Cell architecture (b) Biochip model: array of cells (c) Imperfect split operation [1]

Fig. 1: Biochip and architecture

the issue of operation variability. Their approach is to generate extra droplets (checkpoints), which are stored on the chip, in order to recover from errors, rolling back to the saved droplets. Such an approach saves time at the expense of extra chip area and reagent volumes.

The approach we present in this paper trades-off execution time for chip area and reagent volumes, by redoing the failed operation, such as split. The two fault-tolerant techniques are complementary and can be used in conjunction with each other.

Our research has addressed so far tasks (i) to (v) [13]. In this paper we use our synthesis approach from [13] to generate a non-fault-tolerant implementation. Starting from such an implementation, our focus is on generating fault-tolerant schedules and on the optimization and placement of volume sensors. 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.

This paper is organized in six sections. Sections II-A and II-B present the architecture of the digital microfluidic biochip, and the sequencing graph that represents a biochemical application, respectively. We formulate the problem in Section III and use an example to illustrate the synthesis tasks. In Section IV we present the proposed synthesis approach. The results of the fault-tolerant implementation are discussed in Section V, and in the last section we present our conclusions.

II. System Model

A. Biochip Architecture

In a digital microfluidic biochip the manipulation of liquids is performed using discrete droplets. We use electrowetting-on-dielectric (EWD) [14] for droplet movement. A biochip is composed of several cells. The architecture of a cell is presented in Fig. 1a and the biochip architecture in Fig. 1b. With EWD, the movement of droplets is controlled by applying voltages to the required electrodes. For example, in Fig. 1a, turning off the middle control electrode and turning on the control electrode to the right, will force the droplet to move to the right [14]. The chip also contains input and output ports and detectors. The detection can be done by using, for example, a light-emitting diode (LED) beneath the bottom plate and a photodiode on the top plate.

The chip used in Fig. 1b can be used for the diagnosis of metabolic disorders, by measuring the lactate and glucose level in human physiological fluids. Hence, the device contains the necessary input ports for introducing the samples (urine, plasma and serum) and the reagents (lactate and glutamate oxidase and buffer substance NaOH) on the microfluidic array, where the corresponding protocol will be performed.

Using this architecture, and charging correspondingly the voltages, all the required operations, such as transporting, splitting, merging, dispensing, mixing, and detection, can be performed. For example, the mixing operation is done by transporting two droplets to the same location, and then moving them next to each other. A mixing module can be created by grouping adjacent electrodes on which the droplet can be moved. Any cells on the chip can be used for such a purpose. We consider that the designers will build and characterize a module library $\mathcal{L}$, where for each operation there are several options with varying area and execution times, see Table I.

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 [11], the resulting droplet volumes after a split operation might be unbalanced, see Fig. 1c. Applications are very sensitive to volume variations: the required precision in liquid handling, measured by the standard deviation of a set

TABLE I: Module Library

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

 ISBN: 978-2-35500-011-9

of volumes divided by the mean, is ±2% for microdialysis applications and ±10% in drug discovery applications. We consider that a split operation is faulty if it results in droplets with volumes below a given threshold. The threshold is given by the designer and depends on the application.

Our fault model assumes a maximum of k faults, given a biochip architecture and a biochemical application. These faults can appear in any split operation and have to be tolerated. The error is detected using on-chip volume sensors. The sensors have to be placed on the top of existing electrodes. One of the resulted droplets, after a split, has to be routed to the sensor. The sensing operation can take up to five seconds, depending on the sensor type [15].

If an error is detected (the volume 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, followed by sensing and, in case of error, by merge. In the worst-case, a split will have to be performed $k+1$ times, to tolerate the maximum k faults that can happen in the application. The last split does not have to be followed by a sensing operation, since we know it will not experience an error: all faults have already happened. Note, however, that these k faults can happen in any of the split operations of the application.

B. Biochemical Application Model

A biochemical application is modeled using a sequential directed acyclic graph $G(V,E)$ [13], where each node $O_i \in V$ represents an operation. The binding of operations to modules in the architecture is captured by the function $\mathcal{B}\colon V \rightarrow \mathcal{A}$, where $\mathcal{A}$ is the set of the allocated modules from the given library $\mathcal{L}$. An edge $e_{ij} \in E$ from O_i to O_j indicates that the output droplet obtained after O_i finishes, will be used as input for O_j. An operation can be activated after all its inputs have arrived and it issues its outputs when it terminates. We assume that, for each operation O_i, we know the execution time $C_i^{M_k}$ on module $M_k = \mathcal{B}(O_i)$, where it is assigned for execution. Fig. 2 depicts part of an application, which consists of 15 operations O_1–O_{15}, and it involves a series of mixing operations (O_1, O_2, O_3, O_5, O_6) followed by split operations (O_4, O_7). Operations O_{10}–O_{15} are input operations. In operation O_8 one of the resulted droplets after the O_7 split is routed to a waste reservoir and in O_9 we perform a detection operation on the other droplet. Each operation has a predecessor and a successor, thus we have introduced two NOP nodes, as source and sink nodes (i.e., the graph is polar). Let us consider that the operation O_1 is bound to a 2×4 mixing module denoted by M_1 (i.e., $\mathcal{B}(O_1) = M_1$). Then, according to Table I, the execution time for O_1 will be 3 s. We consider routing as part of an operation time. In this paper we use the data from [14], which allows us to approximate that the time required to route the droplet one cell is 0.01 s, an order of magnitude smaller than operation times, see Table I.

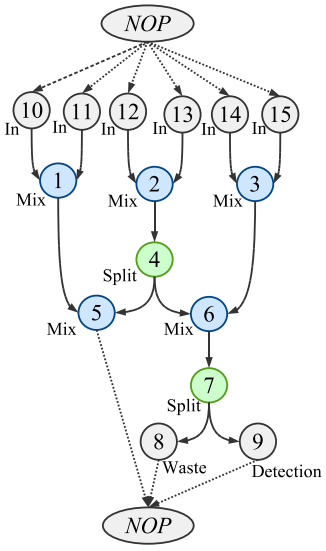


Fig. 2: Biochemical application model example

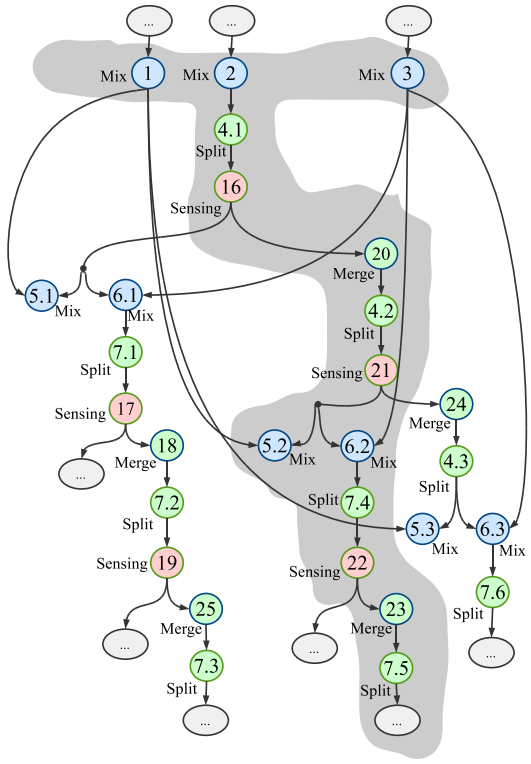
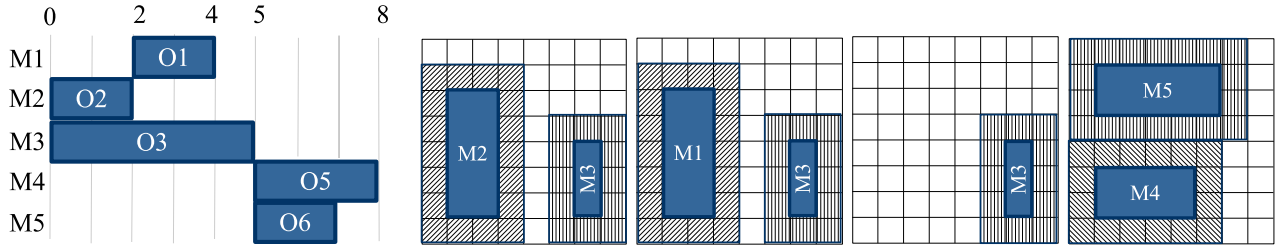


Fig. 3: Fault-Tolerant Sequencing Graph (FTSG)

Such a model does not capture the fault occurrences during split operations. In this paper, we propose a fault-tolerant sequencing graph $\mathcal{G}(\mathcal{V}, E \cup E_C)$, see Fig. 3. In $\mathcal{G}$, each split operation is followed by a sensing operation which detects if a fault has occurred. For example, operation O_{16} in Fig. 3 is a sensing operation for split operation O_4. Note that operations O_8–O_{15} from Fig. 2 are depicted in Fig. 3 as "..." due to space constraints. During a sensing operation, one of the



(a) Schedule (b) Placement at $t = 0$ (c) $t = 2$ (d) $t = 4$ (e) $t = 5$

Fig. 4: Schedule for application graph

droplets resulted from the previous split operation is routed to the sensor for error detection. The overhead added by the routing time to the sensor is considered part of the sensing operation execution time. The number s of the sensors and their placement on the biochip are decided by our synthesis approach.

Each sensing operation is followed by two conditional edges $\in E_C$ corresponding to the faulty and non-faulty split scenarios, respectively. In Fig. 3, in case a fault is detected by sensing operation O_{16}, the condition on edge $O_{16} \rightarrow O_{20}$ 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 sensing operation does not detect a fault, nodes $O_{5.1}$ and $O_{6.1}$ are activated instead.

Section IV-A presents how we derive the fault-tolerant graph $\mathcal{G}$ starting from the application graph G. The fault-tolerant graph $\mathcal{G}$ in Fig. 3 captures all the fault scenarios that can happen during the execution of application G from Fig. 2, considering a maximum number of 2 faults, i.e., $k = 2$. For example, the shaded subgraph captures the fault scenario when one fault happens during O_4 and the second fault happens during O_7.

III. Problem Formulation

In this paper we address the following problem. As input we have a biochemical application modeled as a graph $G(V,E)$, which is performed on a biochip platform represented by a $m \times n$ array $\mathcal{C}$ of cells and a characterized module library $\mathcal{L}$. The fault model is given by the parameter k which denotes the maximum number of faults that can occur. The designer specifies the maximum number s of volume sensors that can be used. We are interested in synthesizing a fault-tolerant implementation Ψ such that the worst-case application completion time δ_G is minimized.

Synthesizing an implementation $\Psi = <\mathcal{A},\mathcal{B},\mathcal{P},\mathcal{S}>$ means deciding on: the allocation $\mathcal{A}$, which determines what modules from library $\mathcal{L}$ are to be used; the binding $\mathcal{B}$ of each operation $O_i \in \mathcal{V}$ to a module $M_k \in \mathcal{A}$;the placement $\mathcal{P}$ of the modules and of the sensors on the $m \times n$ array $\mathcal{C}$; the fault tolerant schedule $\mathcal{S}$ of the application, which contains the start time of each operation O_i on its corresponding module.

Let us illustrate each of these tasks by considering the application graph $G(V,E)$ from Fig. 2 which is performed on an 8×8 biochip such as the one from Fig. 1b.

A. Allocation and Placement

The input and detection operations are already assigned to the corresponding input ports and detection module, respectively. Thus, O_{10} is assigned to the input port S_1, O_{11} to S_2, O_{12} to S_3, O_{13} to B_1, O_{14} to B_2, O_{15} to R_1. Detection operation O_9 is allocated to the photodetector. However, for the remaining mixing operations (O_1, O_2, O_3, O_5, O_6) and split operations (O_4, O_7), our synthesis approach has to allocate the appropriate modules. We consider that a split operation takes place at the same location as the preceding operation.

Let us consider the module library $\mathcal{L}$ provided in Table I. During the allocation phase, certain 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. 4b–e. 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 cells), then they tend to merge into a single droplet. That is the reason why, when a module is placed on the chip, a protection border is needed. The placement for the discussed solution is as indicated in the hashed are from Fig. 4b–e, where, for example, module M_1 is a 2×5 mixer (4×7 with protection borders) placed in the bottom left corner of the 8×8 chip.

B. Binding and Scheduling

Once the modules have been allocated and placed on the cell array, we have to decide where to execute the operations (binding) and in which order (scheduling), such that the application completion time is minimized.

Considering the graph in Fig. 2, the obtained schedule without fault-tolerance is presented in Fig. 4a. 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. For example, operation O_1 is bound to module M_1 and starts immediately after operation O_2 ($t_1^{start} = 2$) and takes 2 s, finishing at time $t_1^{finish} = 4$. The total schedule length is 8 s.

 ISBN: 978-2-35500-011-9

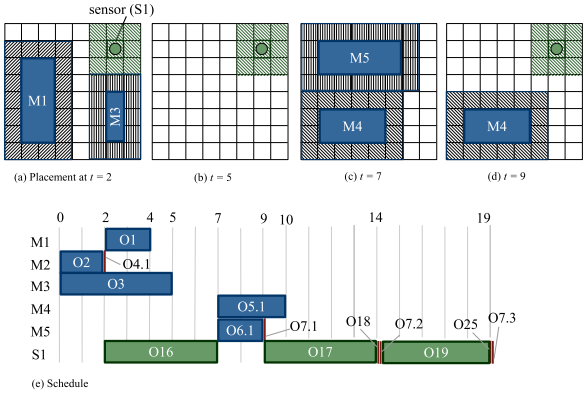


Fig. 5: FTS schedule for two faults in O7

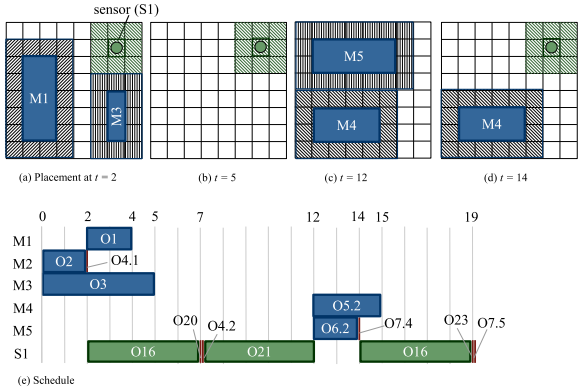


Fig. 7: FTS schedule for faults in O4 and O7

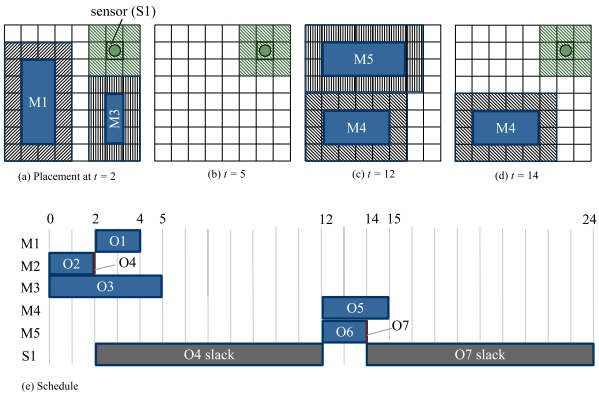


Fig. 6: SS schedule

C. Fault-tolerant Scheduling

However, the presented schedule does not take in account the possibility of fault occurrence during a split operation. Let us consider a maximum number k of faults that can occur during the application execution. The faults are detected using sensors. The sensors differ from the modules described above, as they are real devices, so their placement is not reconfigurable. For the application in Fig. 2, we use one sensor, placed as in Fig. 6a–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. 4a 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 (sensing) and recovery (merging back the droplets, followed again by a split). Considering the worst-case, in which all k faults happen in the same split operation, the required slack time is calculated as:

$$t_{slack} = k \times (t_{sensing} + t_{merge} + t_{split}). \quad (1)$$

We assume that merge and split operations are instantaneous and we use a sensing time of 5 s, see Table. I. Thus, for k = 2, the slack required for recovering the split operation O_4 is 2×5 = 10 s, as depicted in Fig. 6e, with a rectangle labeled "O4 slack". A similar slack is introduced for O_7, thus obtaining the fault-tolerant schedule from Fig. 6e, with a worst-case application completion time δ_G = 24 s. We call such a fault tolerant strategy Straightforward Scheduling (SS). The schedule obtained by using SS wastes a lot of unnecessary time for recovery. For example, for the schedule in Fig. 6e, if both faults happened during the split operation O_4, then the maximum number of faults (k = 2) is reached, and hence there is not need in allocating slack time after split operation O_7.

Therefore, in this paper, we propose an improved fault-tolerant scheduling (FTS) technique, which can take into account the actual fault-occurrence pattern during the execution. By taking into account fault-occurrence information, FTS produces shorter schedules, leading to a reduced worst-case application completion time δ_G. FTS relies on the fault-tolerant sequencing graph $\mathcal{G}$, proposed in Section II, which captures all the possible fault-scenarios. The FTSG from Fig. 3 is build starting from the application graph from Fig. 2 and captures all alternative scenarios for k = 2. Starting from the FTSG $\mathcal{G}$ our FTS 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. 7e).

During runtime, depending on the detected fault occurrences, a scheduler will activate the corresponding operations. For example, for the fault scenario captured by the shaded subgraph in Fig. 3 (first fault in O_4 and the second in O_7), the operations in Fig. 7e 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. 5e. The worst-case application completion time δ_G is 19 s for FTS, compared to 24 s for SS. The difference between FTS and SS results from the sensing operation time: unnecessary sensing operations are avoided by FTS. We have considered that a sensing operation takes 5 s. However, there are capacitance sensor implementations that can detect a droplet volume in shorter time [15]. In this case, SS is preferable over FTS due to its simplicity.

 ISBN: 978-2-35500-011-9

IV. Fault-Tolerant Synthesis Strategy

Our fault-tolerant synthesis approach is outlined in Fig. 8 and has three steps:

1) In the first step, we use our synthesis algorithm from [13], to obtain the allocation $\mathcal{A}^0$, binding $\mathcal{B}^0$ and placement $\mathcal{P}^0$ that minimizes the application completion time without considering faults (line 1 in Fig. 8). We have extended the synthesis strategy from [13] to decide the number of sensors and their placement given the maximum number of sensors s that are available.
2) In the second step, we build a FTSG model $\mathcal{G}$ starting from the application graph G (line 2 in Fig. 8) that captures all fault scenarios for a given k maximum number of faults.
3) During the third step (line 3 in Fig. 8), we obtain a fault-tolerant schedule table $\mathcal{S}$ using the FTS algorithm presented in Section IV-B.

A. Fault-tolerant Graph Generation

The FTSG graph $\mathcal{G}$ is generated by the function **GenerateFTSG** which takes as parameters the application graph G and the maximum number of faults k. For the application graph in Fig. 2, considering $k = 2$, we obtain the FTSG from Fig. 3. Each split operation O_i is transformed into a structure which models all possible fault occurrence scenarios. For example, O_4 is transformed into the structure that starts with node $O_{4.1}$ in Fig. 3. We use the notation convention $O_{i.x}$ to denote the x^{th} copy of the split operation O_i inserted in $\mathcal{G}$. Each such split operation is followed by a sensing operation. For example, $O_{4.1}$ is followed by the sensing operation O_{16}. There are two possibilities: a fault is detected, or no fault is detected. Both scenarios are captured by the split structure, using conditional edges. For example, for the sensing operation O_{16}, we insert the following conditional edges: $O_{16} \rightarrow O_{20}$ under the condition of a fault occurrence (true), and edges $O_{16} \rightarrow O_{5.1}$ and $O_{16} \rightarrow O_{6.1}$, under the condition of no fault occurrence (false), 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 G the subgraphs originating from the split operation.

We continue the transformation with the next split operations, including those introduced in $\mathcal{G}$ by the previous transformations. In Fig. 3, $k = 2$. The split operation $O_{4.2}$ is placed on the faulty branch originating from the sensing operation O_{16}, which means that a fault has already occurred (in $O_{4.1}$). Since $k = 2$, another fault can occur, which means that $O_{4.2}$ has to be followed by a sensing operation, O_{21}. Our construction procedure keeps track of the fault occurrence to build the structure of $\mathcal{G}$. 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 sensing operation, since we are currently in the scenario when both faults have already occurred (first in $O_{4.1}$ and second in $O_{4.2}$).

The process continues until all possible alternative scenarios are built. A scenario represents the fault pattern of maximum k faults that can happen during the split operations from G. The graph in Fig. 3 assumes a maximum number of 2 faults which can occur on 2 split operations. 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 on 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. 3.

FTSynthesis(G, $\mathcal{C}$, $\mathcal{L}$, k)
1 Ψ^0 = DMBSynthesis(G, $\mathcal{C}$, $\mathcal{L}$)
2 $\mathcal{G}$ = GenerateFTSG(G, k)
3 $\mathcal{S}$ = FTScheduling($\mathcal{G}$, $\mathcal{C}$, $\mathcal{B}^0$, $\mathcal{P}^0$, k)
4 **return** $\Psi = < \mathcal{A}^0, \mathcal{B}^0, \mathcal{S}, \mathcal{P}^0 >$

Fig. 8: Fault-tolerant Synthesis

B. Fault-tolerant Scheduling Algorithm

The fault-tolerant schedule table $\mathcal{S}$ is obtained by the **FTScheduling** algorithm from Fig. 9, which takes as input the FTSG graph $\mathcal{G}$ generated in the previous step, the biochip architecture $\mathcal{C}$ and the binding $\mathcal{B}^0$ and placement $\mathcal{P}^0$ obtained in step 1 by our **DMBSynthesis** from [13]. We start by generating all the fault scenarios $\mathcal{F}$ (line 1 in Fig.ẽfalg:FTS), see the previous section for the list of scenarios captured in graph $\mathcal{G}$. Then, we traverse the FTSG and extract all subgraphs corresponding to each possible scenario $F_i \in \mathcal{F}$. 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. 3, 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 $O_{16} \rightarrow O_{5.1}$ and $O_{16} \rightarrow O_{6.1}$. The process continues until all split operations are evaluated. Eventually, for $\{O_4,$

FTScheduling(G, $\mathcal{C}$, $\mathcal{B}$, $\mathcal{P}$)
1 $\mathcal{F}$ = GenerateFaultScenarios($\mathcal{G}$)
2 $\mathcal{S} = \emptyset$
3 **for** each $F_i \in \mathcal{F}$ do **do**
4 $\quad\mathcal{G}' = \mathcal{G}$
5 $\quad O_i$ = source
6 $\quad$**while** $O_i \neq \emptyset$ **do**
7 $\quad\quad$**if** O_i is split operation then **then**
8 $\quad\quad\quad$RemoveBranch($\mathcal{G}'$, O_i, F_i)
9 $\quad\quad$**end if**
10 $\quad\quad O_i$ = BFS($\mathcal{G}'$, O_i)
11 $\quad$**end while**
12 $\quad$*Graph* = $\mathcal{G}'$
13 $\quad\mathcal{S}$ = ListScheduling(*Graph*, $\mathcal{C}$, $\mathcal{B}$, $\mathcal{P}$) $\cup$ $\mathcal{S}$
14 **end for**
15 **return** $\mathcal{S}$

Fig. 9: FTS algorithm

ListScheduling(*Graph*, $\mathcal{C}$, $\mathcal{B}$, $\mathcal{P}$)

1 CriticalPath(*Graph*)
2 **repeat**
3 *List* = GetReadyOperations(*Graph*)
4 O_i = RemoveOperation(*List*)
5 t_i^{start} = Schedule(O_i, $\mathcal{B}(O_i)$, $\mathcal{C}$, $\mathcal{P}$)
6 t = earliest time when a scheduled operation terminates
7 UpdateReadyList(*Graph*, t, *List*)
8 **until** *List* = $\emptyset$
9 **return** $\mathcal{S}$

Fig. 10: List scheduling algorithm

$O_7\}$ we obtain the shaded subgraph in Fig. 3.

After extracting the scenario subgraphs, we schedule each of them (Fig. 9, line 13) by using the list scheduling algorithm from Fig. 10. The **ListScheduling** function takes as input the $m \times n$ chip $\mathcal{C}$, the subgraph *Graph*, the binding $\mathcal{B}^0$ and the placement $\mathcal{P}^0$. Every node from *Graph* is assigned a specific priority according to the critical path priority function (line 1) [16]. *List* contains all operations that are ready to run, sorted by priority. The algorithm takes each ready operation O_i, stored in *List*, and schedules it at the time when corresponding module $M_i = \mathcal{B}(O_i)$ can be placed on the chip $\mathcal{C}$ (line 5). When a scheduled operation finishes executing, the *List* is updated with the operations that have become ready. The **ListScheduling** function outputs the schedule table obtained for *Graph*. For example, for the shaded subgraph in Fig. 3, **ListScheduling** will produce the schedule table from Fig. 7e.

V. Experimental Results

In order to evaluate the proposed synthesis approach, we have used two real life examples and seven synthetic benchmarks. The **FTSynthesis** algorithm was implemented in Java (JDK 1.6), running on a MacBook Pro computer with Intel Core 2 Duo CPU at 2.53 GHz and 4 GB of RAM. The module library used for all experiments is shown in Table. I.

For the first set of experiments we were interested to evaluate the proposed synthesis approach in terms of worst-case application completion time δ_G, as the number of faults increases. For this, we have compared the δ_G^{FTS} obtained by our **FTScheduling** from Section IV-B with δ_G^{SS} obtained by the Straightforward Scheduling (SS) approach, considering the same binding and placement, produced by **DMBSynthesis** in line 1 in Fig. 8. SS generates a fault-tolerant schedule by inserting slack, as discussed in Section III-C. Thus, we insert in the application graph G a "slack" operation after each split operation. The slack execution time is calculated using the formula Eq. (1). We then apply the list scheduling algorithm from Fig. 10 to obtain the fault-tolerant schedule. For the application graph in Fig. 2, and k = 2 we obtained the fault tolerant schedule of 24 s, depicted in Fig. 6.

We have used two real-life applications: (1) In-vitro diagnostics on human physiological fluids (IVD) [7], which has 25 operations and (2) The colorimetric protein assay (PRT) [10] utilized for measuring the concentration of a protein in a solution, which has 134 nodes. For all benchmarks, including the seven synthetic applications from [13], we ignored the detection operations, and the dilution operations were replaced by a mix operation followed by a split operation.

TABLE II: Results

Nodes	Area	s	$k=2$		$k=3$		$k=4$		$k=5$	
			SS	FTS	SS	FTS	SS	FTS	SS	FTS
10	6×6	1	46	41	56	46	66	51	76	53
20	8×8	2	37	29	47	36	57	46	67	56
30	8×12	3	40	36	55	37	70	56	85	76
40	10×8	2	37	33	48	38	58	40	68	45
50	8×12	3	44	38	57	43	73	49	87	51
60	12×10	4	50	45	59	50	65	50	79	52
70	10×12	4	65	60	82	63	102	66	122	74
IVD	10×10	2	36	31	41	36	51	36	61	41
PRT	15×15	6	88	68	114	73	145	76	176	84

The results are presented in Table II, where we have, in separate columns, the schedule lengths of both SS and FTS approaches for k 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 using the FTS approach results in reduced application completion times compared to SS, especially as k increases. For k = 5 we have obtained an average improvement of 52.4% in the FTS completion time compared to SS.

Our synthesis approach has three steps: running the adapted implementation from [13] for the specified times (60–1,800 s), generating the fault-tolerant graph, which takes very little time, and obtaining the fault tolerant schedules. The duration of the last step increases exponentially with the number of faults k and the number of split operations. For example, for the in-vitro diagnostics, a real life application with 4 split operations [7], the 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 implementation quality. The results presented in Table III are obtained for the IVD application, for a fixed number of faults, k = 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 FTS over SS. 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.

TABLE III: Results

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

However, our proposed FTS approach produces significantly better schedules than SS, thus allowing us to save costs. For example, in the most constrained case, an 12×12 biochip area and 3 sensors, we have obtained an improvement of 48.3% compared to SS.

VI. Conclusions

In this paper we have proposed a fault-tolerant synthesis approach for digital microfluidic biochips. We have considered that split operations are faulty 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 redoing the split operation. We have used the synthesis strategy from [13] to generate a binding and placement of operation (ignoring faults) and we have focused on generating good quality fault-tolerant schedules.

We have proposed a fault-tolerant sequencing graph that can capture all the fault scenarios in the application and we have devised a scheduling technique to derive the fault-tolerant schedule tables.

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.

References

[1] R. B. Fair, "Biochip engineering," 2008, lecture notes.

[2] K. Chakrabarty and F. Su, *Digital Microfluidic Biochips: Synthesis, Testing, and Reconfiguration Techniques*. Boca Raton, FL: CRC Press, 2006.

[3] F. Su and K. Chakrabarty, "Architectural-level synthesis of digital microfluidics-based biochips," in *Proceedings of International Conference on Computer Aided Design*, 2004, pp. 223–228.

[4] ——, "Unified high-level synthesis and module placement for defect-tolerant microfluidic biochips," in *Proceedings of the 42nd annual conference on Design automation*, 2005, pp. 825–830.

[5] P.-H. Yuh, C.-L. Yang, and Y.-W. Chang, "Placement of digital microfluidic biochips using the T-tree formulation," in *Proceedings of Design Automation Conference*, 2006, pp. 931–934.

[6] M. Cho and D. Z. Pan, "A high-performance droplet router for digital microfluidic biochips," in *Proceedings of International Symposium on Physical Design*, 2008, pp. 200–206.

[7] F. Su, W. Hwang, and K. Chakrabarty, "Droplet routing in the synthesis of digital microfluidic biochips," in *Proceedings of Design, Automation and Test in Europe*, vol. 1, 2006, pp. 73–78.

[8] T. Xu and K. Chakrabarty, "Functional testing of digital microfluidic biochips," in *Proc. Int. Test Conf.*, 2007, pp. 1 – 10.

[9] H. G. Kerkhoff, "Testing microelectronic biofluidic systems," *IEEE Des. Test Comput.*, vol. 24, no. 1, pp. 72–82, 2007.

[10] F. Su and K. Chakrabarty, "Module placement for fault-tolerant microfluidics-based biochips," *ACM Transactions on Design Automation of Electronic Systems*, vol. 11, no. 3, pp. 682–710, 2006.

[11] H. Ren, R. B. Fair, and M. G. Pollack, "Automated on-chip droplet dispensing with volume control by electro-wetting actuation and capacitance metering," *Sensors and Actuators B*, vol. 98, no. 2–3, pp. 319–327, 2004.

[12] Y. Zhao, T. Xu, and K. Chakrabarty, "Control-path design and error recovery in digital microfluidic lab-on-chip," *ACM Journal on Emerging Technologies in Computing Systems*, 2010.

[13] E. Maftei, P. Paul, and J. Madsen, "Tabu search-based synthesis of dynamically reconfigurable digital microfluidic biochips," in *Proceedings of the Compilers, Architecture, and Synthesis for Embedded Systems Conference*, 2009, pp. 195–203.

[14] M. G. Pollack, A. D. Shenderov, and R. B. Fair, "Electrowetting-based actuation of droplets for integrated microfluidics," *Lab Chip Journal*, vol. 2, pp. 96–101, 2002.

[15] M. G. Pollack, "Electrowetting-based microactuation of droplets for integrated microfluidics," Ph.D. dissertation, 2001.

[16] G. D. Micheli, *Synthesis and Optimization of Digital Circuits*. McGraw-Hill Science, 1994.

 ISBN: 978-2-35500-011-9