Synthesis of Biochemical Applications on Digital Microfluidic Biochips with Operation Execution Time Variability

Mirela Alistar, Paul Pop*

Department of Computer Science and Applied Mathematics, Technical University of Denmark, Lyngby 2800, Denmark

Abstract

Microfluidic-based biochips are replacing the conventional biochemical analyzers, and are able to integrate all the necessary functions for biochemical analysis. The digital microfluidic biochips are based on the manipulation of liquids not as a continuous flow, but as discrete droplets. Several approaches have been proposed for the synthesis of digital microfluidic biochips, which, starting from a biochemical application and a given biochip architecture, determine the allocation, resource binding, scheduling, placement and routing of the operations in the application. Researchers have assumed that each biochemical operation in an application is characterized by a worst-case execution time (*wcet*). However, during the execution of the application, due to variability and randomness in biochemical reactions, operations may finish earlier than their *wcet*s, resulting in unexploited slack in the schedule. In this paper, we first propose an online synthesis strategy that re-synthesizes the application at runtime when operations experience variability in their execution time, exploiting thus the slack to obtain shorter application completion times. We also propose a quasi-static synthesis strategy that determines offline a database of alternative implementations. During the execution of the application, several implementations are selected based on the current execution scenario with operation execution time variability. The proposed strategies have been evaluated using several benchmarks and compared to related work.

Keywords: Microfluidic biochips, Electrowetting on dielectric, Lab-on-a-chip, Computer-Aided Design, Real-time scheduling

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 [1]. 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 [2]. The trend today is towards *microfluidic platforms*, which according to [2], 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) [2–4].

Microfluidic platforms are classified according to the liquid propulsion principle (e.g., capillary, pressure driven, centrifugal, electrokinetic, acoustic) used for operation. In this paper, we are interested in microfluidic platforms that manipulate the fluids as droplets, using electrokinetics, i.e., electrowetting-on-dielectric (EWOD) [5]. We call such platforms *digital microfluidic biochips* (DMBs). DMBs are able to perform operations such as dispensing, transport, mixing, split, dilution and detection using droplets (discrete amount of fluid of nanoliters volume) [6].

To be executed on a DMB, a biochemical application has to be synthesized. There is a significant amount of work on the synthesis of DMBs [4, 7–10], which typically consists of the following tasks: *modeling* of the biochemical application functionality and biochip architecture, *allocation*, during which the needed modules are selected from a module library, *binding* the selected modules to the biochemical operations in the application, *placement*, during which the positions of the modules on the biochip are decided, *scheduling*, when the order of operations is determined and *routing* the droplets to the needed locations on the biochip. The output of these synthesis tasks is the "electrode actuation sequence", applied by a control software to run the biochemical application. The control software executes on a computer connected to the biochip, as schematically represented in Fig. 1a.

All synthesis strategies proposed so far in related research (the only exception is [11]) consider a given module library $\mathcal{L}$, which contains for each operation its worst-case execution time (*wcet*). However, an operation can finish before its *wcet*, due to variability and randomness in biochemical reactions [12, 13]. Such situations, when the actual execution time of the operation is less than the *wcet*, result in time slacks in the schedule of operations. These time slacks can be used for executing

*Corresponding author
Email address: paupo@dtu.dk (Paul Pop)

other operations in the application, thus, reducing the application completion time. Besides reduced costs, due to shorter experimental times, reducing the application execution time can also be beneficial for fault-tolerance. For example, researchers have shown [14, 15] how the slack can be used to introduce recovery operations to tolerate transient faults.

1.1. Related work and contributions

The only work that addresses variability in operation execution is [11], which proposes an Operation-Interdependency-Aware (OIA) synthesis to derive an offline implementation that is scaled at runtime according to the actual operation execution scenario. The strategy of OIA is to group the operations according to their types and scheduling constraints and then to schedule the operations in phases. A dilution/mixing (D/M) phase contains all dilution and mixing operations that can be executed at the same time without violating the fluidic and scheduling constraints. Similarly, a transport (T) phase contains the dispensing and transport operations that satisfy all the necessary constraints to be scheduled concurrently. The D/M and T phases are executed alternatively, and each phase is considered completed when all its operations finish executing.

Although OIA can handle any variability in the operation execution by deriving a scalable schedule, its approach where all operations of the same phase have to wait for each other to finish is overly pessimistic and leads to long application completion times.

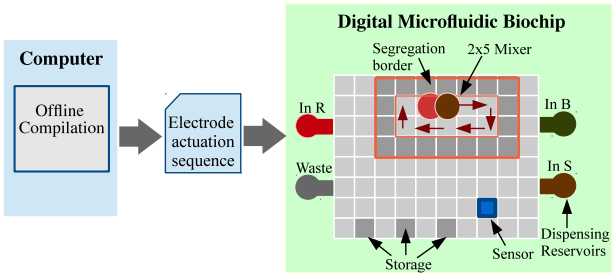
In this paper we first propose an Online Synthesis strategy (called ONS) that, when an operation finishes earlier than its *wcet*, runs a re-synthesis to derive a new solution, aiming at minimizing the application completion time. Because it is executed at runtime, our online synthesis strategy has the advantage of taking into account the actual operation execution times, successfully adapting the binding, placement, routing and scheduling of operations. The disadvantage of an online approach is its overhead due to multiple runtime re-syntheses which add delays to the application completion time. However, the execution of the synthesis tasks on the computer is orders of magnitude smaller compared to typical biochemical operation completion times [7, 15, 16]. Consequently, the runtime overhead is not significant and thus an online re-synthesis is a viable strategy.

An online synthesis strategy also needs a powerful microcontroller or computer attached to the biochip, in order to run the re-synthesis at runtime. To avoid the runtime overheads and the need of powerful microcontrollers, we also propose a Quasi-Static Synthesis strategy (QSS), which derives offline alternative implementations which are stored in a database. During the execution of the application, we select from the database the implementation that matches the best the current execution scenario. Then, the selected implementation is applied and the application continues executing. With QSS we can take advantage of the actual operation execution times without the re-synthesis overhead added by the online approach. However, deriving and storing the complete database of solutions (i.e., containing the solutions for all possible execution scenarios) is feasible only for small applications. Our proposed QSS approach derives only a part of the database, aiming at finding a good balance between the number of stored solutions and the quality of the results in terms of application completion time.

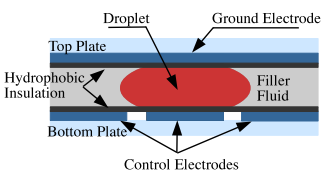
In this paper we propose two synthesis strategies that exploit the slack time resulted due to uncertainties in operation execution, aiming at minimizing the application completion time. This paper is organized in eight sections. Sections 2 and 3 present the architecture and application models, respectively. We formulate the problem in Section 4 and present a motivational example. Next, we discuss our proposed approaches: the online synthesis strategy (Section 5) and the quasi-static synthesis strategy (Section 6). The proposed algorithms are evaluated in Section 7, and Section 8 presents our conclusions.

2. Biochip architecture model

In a DMB, a droplet is sandwiched between a top ground-electrode and bottom control-electrodes, see Fig. 1b. Two glass plates, a top and a bottom one, protect the droplets from external factors. 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. The droplets are manipulated using the EWOD principle [5]. For example, in Fig. 1b, if the control-electrode on which the droplet is resting is turned off,



a) Biochip architecture model example



b) Electrowetting on dielectric (EWOD)

Figure 1: Biochip architecture model example

Table 1: Module library $\mathcal{L}$

Operation	Module area	*bcet* (s)	*wcet* (s)
Dispensing	N/A	1	2
Mix	3×6	2	3.47
Mix	4×6	1.5	2.5
Mix	4×2	2.5	4.3
Mix	4×3	2.3	4
Mix	1×3	4	7
Mix	3×3	1	5
Dilution	3×6	2.3	4
Dilution	4×6	1.5	3.1
Dilution	1×3	5	10
Dilution	3×3	3	7
Store	1×1	N/A	N/A
Transport	1×1	0.01	0.01
Detection	1×1	5	5

and the left control-electrode is activated by applying voltage, the droplet will move to the left. A biochip is typically connected to a computer (or microcontroller) as shown in Fig. 1a. The biochip is controlled by the "electrode actuation sequence" that specifies for each time step which electrodes have to be turned on and off.

A DMB is modeled as a two-dimensional array of identical control-electrodes, see Fig. 1a, where each electrode can hold a droplet. There are two types of operations: reconfigurable (mixing, split, dilution, merge, transport), which can be executed on any electrode on the biochip, and non-reconfigurable (dispensing, detection), which are bound to a specific device such as a reservoir, a detector or a sensor. 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. 1a). Considering the biochip in Fig. 1a, a droplet can only move up, down, left or right with EWOD, and cannot move diagonally. 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.

The biochip contains non-reconfigurable devices such as input (dispensing) and waste reservoirs, sensors and actuators, on which the non-reconfigurable operations are performed. For example, the biochip from Fig. 1a has three dispensing reservoirs, one for buffer, one for sample, one for reagent and one waste reservoir. The location of these non-reconfigurable devices is fixed on the biochip array. Fig. 1a shows the location of reservoirs and a sensor, which are placed on a biochip architecture of 10×8 electrodes.

Each reconfigurable operation is executed in a determined biochip area, called a "module". For example, the two droplets from Fig. 1a are mixing on a 2×5 module, by moving according to the indicated pattern. In case two droplets are on neighboring electrodes, they merge instantly. To avoid accidental merging, each module is surrounded by a "segregation border" of one-electrode thickness, see Fig. 1a. With the exception of [11], all of the research so far has assumed that each operation will execute for a given *wcet*. Significant research work has been done to determine the *wcet* of the fluidic operations such as mixing and dilution [17–21]. Thus, based on experiments, researchers characterize a module library $\mathcal{L}$, such as the one in Table 1, which provides the area and corresponding *wcet* for each operation. As shown in Table 1, the time needed for two droplets to mix on a 3×6 module is 3.47 *s*. These *wcet*s are safe pessimistic values for the execution times of the operations, because so far there has not been any technique to determine when an operation has completed.

Recent work [11, 14] has addressed the cybephysical integration of the biochip and the control system. In such a setup, the biochip is equipped with a "sensing system" [22, 23] that can monitor the execution of an operation, monitoring attributes such as color, volume, diameter and position [11]. We have used such a sensing system to detect if an operation is erroneous, and thus provide fault-tolerance for transient faults [14]. Similar to [11], in this paper, we assume that we are able to also determine the completion time of an operation. We suggest, as in [11], the use of a Charge-Coupled Device (CCD) camera-based system. The CCD camera-based system is used to capture periodically images of the droplets during the execution of the each operation. The images are analyzed in real-time and thus the position, size and concentration of the droplets are determined. By comparing the results with the nominal values, we can determine if an operation has finished executing (e.g., a transport operation finishes when droplets are at the wanted location, a dilution finishes when the product droplet has the desired concentration). Note that the choice of the sensing system is orthogonal to the problem addressed in this paper. The strategies proposed in this paper are general and can work with any available sensing system.

3. Biochemical application model

A biochemical application is modeled using an acyclic directed graph [3], where the nodes represent the operations, and the edges represent the dependencies between them.

In this paper, we denote with $\mathcal{G}$ the biochemical application model, such as the one in Fig. 2. A node in $\mathcal{G}$ represents an operation O_i, thus in Fig. 2 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. The mixing operation O_7 is ready to execute only after operations O_6 and O_{11} have finished executing and the droplets have been transported to the biochip area where O_7 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 1, row 12) 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 synthesis. We use the data from [5], thus we assume that routing a droplet between two adjacent electrodes takes 0.01 *s* (see the "Transport" operation in Table 1). A droplet is dispensed in 2 *s* [20].

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 [8, 24]. We assume that we know for each operation O_i its *wcet* C_i and best-case execution time (*bcet*). We denote with E_i the actual execution time of the operation on the biochip.

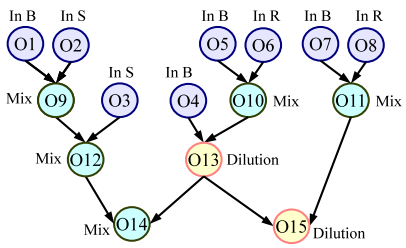


Figure 2: Example application $\mathcal{G}$

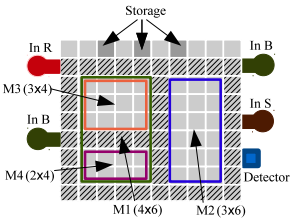


Figure 3: Placement of modules

4. Problem formulation

As an input, we have a biochemical application $\mathcal{G}$ to be executed on a given biochip architecture $\mathcal{A}$. A characterized module library $\mathcal{L}$ containing the area, *bcet* and *wcet* for each operation, is also given as input. We are interested to determine an implementation Ψ, which minimizes the application completion time $\delta_{\mathcal{G}}$ in case of uncertainties in operation execution times. Note that the actual operation execution times will only be known during the execution of the application, once an operation completes.

Deriving an implementation Ψ means deciding on the *allocation* $\mathcal{O}$, the *binding* $\mathcal{B}$, the *placement* $\mathcal{P}$, the *schedule* $\mathcal{S}$ and the *routing* $\mathcal{U}$. During *allocation* $\mathcal{O}$, the modules to be used are selected from the library $\mathcal{L}$. The *binding* $\mathcal{B}$ decides what operations to execute on the allocated modules and the *placement* $\mathcal{P}$ decides the positions of the modules on the biochip architecture $\mathcal{A}$. The *schedule* $\mathcal{S}$ decides the order of operations and the *routing* $\mathcal{U}$ determines the droplets routes to bring the droplets to the needed locations on the biochip.

4.1. Motivational example

In order to illustrate our problem we use as example the application graph $\mathcal{G}$ from Fig. 2, which has to execute on the 10×9 biochip $\mathcal{A}$ in Fig. 3. We consider that the operations are executing on rectangular modules which have their area, *bcet* and *wcet* specified in the library $\mathcal{L}$ from Table 1. For the purpose of this example, we assume that the placement of the modules is fixed as presented in Fig. 3. Also, we ignore routing for simplicity reasons in all the examples in the paper, but we take into account the routing in all our experiments.

Researchers have so far proposed design-time algorithms that use the *wcet*s for the operation execution times. Such a solution is presented in Fig. 4a, and the resulted application completion time $\delta_{\mathcal{G}}$ is 12.94 *s*. 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. For example, operation O_{10} is bound to module M_1 and starts immediately after operations O_5 and O_6 and takes 2.5 *s*.

Let us assume an execution scenario where operations O_{10} to O_{15} finish earlier than their respective *wcet*. In Fig. 4b we show next to each operation O_{10} to O_{15} its actual execution time E_i as observed by the sensing system and its *wcet* C_i (in parentheses) on the respective module. For example, the actual execution time of O_{10} is $E_{10} = 2$ *s*, instead of $C_{10} = 2.5$ *s*. When an operation, such as O_{10}, finishes earlier, we have the opportunity to improve the implementation and thus reduce $\delta_{\mathcal{G}}$. For example, if O_{10} finishes as mentioned in $E_{10} = 2$ *s*, we could start O_{13} sooner (i.e., at $t = 4$) on the faster module M_1, as depicted in Fig. 4b, instead of waiting until $t = 4.5$ and use the slower module M_2 as in Fig. 4a. For the offline solution considering the *wcet*s depicted in Fig. 4a, starting O_{13} at $t = 5.47$ on module M_2 was the best choice possible. However, by knowing

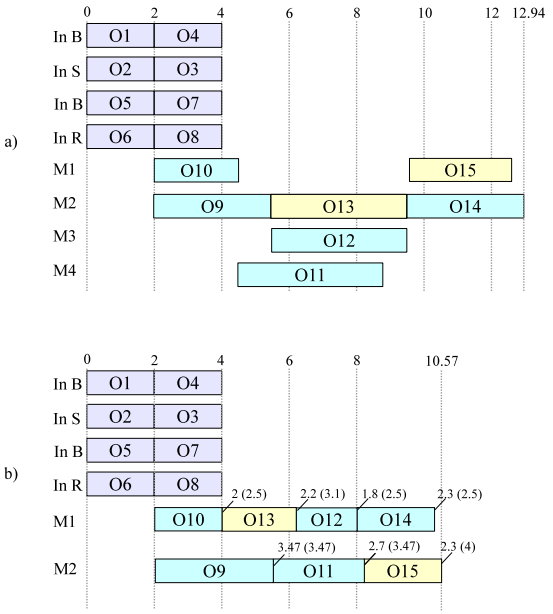


Figure 4: a) Design time (offline) vs b) Runtime (online) synthesis

the actual execution times we can take better decisions (on allocation, binding, placement, routing and scheduling), which reduce $\delta_{\mathcal{G}}$. As we can see in Fig. 4b, by using the actual execution times for the operations, and not their *wcet*s, we can improve $\delta_{\mathcal{G}}$ from 13 *s* to 10.57 *s* (i.e., an improvement of 18.6%). The challenge is that we do not know in advance, at design time, which operations will finish earlier and their execution times. The actual operation execution scenario is only known at runtime, as detected by the available sensing system.

The only work that addresses the uncertainties in operation execution problem is [11], where an Operation-Interdependency-Aware (OIA) synthesis is proposed to derive an offline schedule that is scaled at runtime depending on the actual operation execution times. As an alternative to OIA we propose two other approaches: (1) an Online Synthesis strategy (ONS) which determines at runtime a new implementation Ψ every time we detect that an operation finishes before its *wcet*, and (2) a Quasi-Static Synthesis strategy (QSS), which derives offline a set of implementations, considering varying execution times, and switches online to the most appropriate implementation corresponding to the observed execution times. We discuss in detail these strategies in the following three sections: OIA in Section 7.2, ONS in Section 5 and QSS in Section 6.

5. Online Synthesis strategy

We first propose an Online Synthesis strategy (ONS) to solve the problem formulated in Section 4. As depicted in Fig. 5, ONS is run at runtime each time the sensing system determines that an operation finishes sooner, and it synthesizes a new implementation for the operations that have not yet started or completed.

We use an offline synthesis [25] to determine an initial implementation considering the *wcet*s of operations. Fig. 4a shows

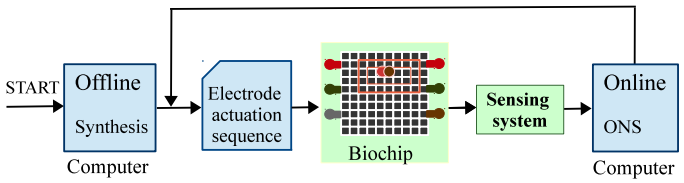


Figure 5: The biochip setup for ONS

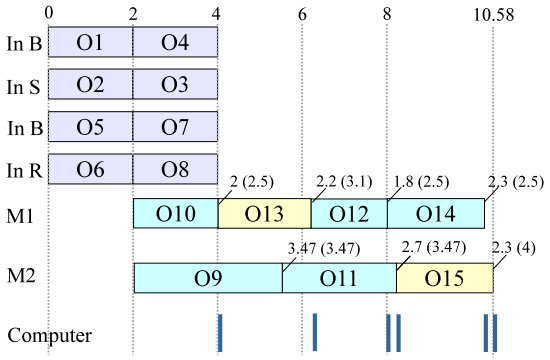


Figure 6: ONS example

the implementation determined offline using *wcet*. At runtime, we start to execute the application according to this offline implementation. Fig. 6 shows how ONS runs considering the example from Section 4. We start from the offline implementation depicted in Fig. 4a. As shown in Fig. 4a, O_{10} has a *wcet* of 2.5 s on M_1. However, when executed on the biochip, the sensing system reports that O_{10} finishes in 2 s instead. In Fig. 6, the thick vertical lines on the row labeled "Computer", mark the runtime overhead of ONS, which is much smaller in comparison to the biochemical operation execution times. For this example we assume that ONS completes in 10 ms. Considering the actual execution times from Fig. 4b, ONS will compute the application in $\delta_{\mathcal{G}} = 10.58$ s.

The synthesis is a NP-complete problem for which several approaches have been proposed. Metaheuristics, such as [4, 25] are able to obtain near-optimal results in terms of application completion time, but take a long time and thus they are not suitable to be executed online. An alternative to such metaheuristics are implementations based on a List Scheduling (LS) [14, 15, 26] heuristic. Although, they cannot guarantee the optimality, LS-based syntheses have been proposed previously for online error recovery [14, 15, 26] and the comparisons to the approaches based on metaheuristics show that LS provides good results in a short time [14]. Hence, we have decided to use a LS-based heuristic for ONS, see the next subsection.

5.1. List Scheduling-based ONS

Our proposed ONS is presented in Fig. 7. ONS takes as input the application graph $\mathcal{G}$, the biochip architecture $\mathcal{A}$, the module library $\mathcal{L}$, the current implementation Ψ and the current time t. The output of ONS is an implementation $\Psi' = \{\mathcal{O}, \mathcal{B}', \mathcal{P}', \mathcal{S}', \mathcal{U}'\}$, where new binding $\mathcal{B}'$, placement $\mathcal{P}'$, schedule $\mathcal{S}'$ and routing $\mathcal{U}'$ are decided. We use the same module allocation $\mathcal{O}$ for all the implementations. Before ONS is run, we sort offline the library $\mathcal{L}$ in ascending order of operation execution time, i.e., the fastest modules come first in the library.

First, ONS adapts the application graph to the current execution scenario. A new graph $\mathcal{G}'$ is obtained by removing the executed operations (line 1). The graph $\mathcal{G}'$ contains the operations that have not yet started or completed. Every node from $\mathcal{G}'$ is assigned a specific priority according to the critical path priority function (line 2 in Fig. 7) [27]. The critical path is defined as the longest path in the graph [27].

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 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. Note that the operations that are interrupted in their execution at the time ONS is triggered are also included in *List*.

The algorithm takes each ready operation O_i (line 5) and performs placement, binding, routing and scheduling. For simplicity, in the examples we have considered a fixed placement that does not change. However, in our implementation the placement may change in each new implementation $\mathcal{P}'$. For the placement of operations we have adapted the Fast Template Placement (FTP) algorithm from [28], which uses: (i) a 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. Hence, the function FTP (line 6) returns the first available module $M_j \in \mathcal{L}$ that can be placed on the biochip $\mathcal{A}$. Since the library has been ordered offline by operation execution time, we know M_j is the fastest available module for O_i.

Next, O_i is bound to M_j (line 7), the routing from the current placement of the input droplets to the location of M_j is determined. Since the routing times are up to three orders of magnitude faster than the other fluidic operation (e.g., routing takes 0.01 s while a mixing operation varies between 2 s and 10 s [20]), in this paper we have approximated the routing overhead as the Manhattan distance between the top-left corners of the modules.

Let us consider the example in Fig. 2. At time $t = 4$ s the

ONS($\mathcal{G}, \mathcal{A}, \mathcal{L}, \Psi, t$)

1: $\mathcal{G}'$ = RemoveExecutedOperations($\mathcal{G}, \Psi$)
2: CriticalPath($\mathcal{G}$)
3: *List* = GetReadyOperations($\mathcal{G}$)
4: **repeat**
5: O_i = RemoveOperation(*List*)
6: $\mathcal{P}'$ = FTP($\mathcal{L}, \mathcal{A}, O_i, t$)
7: $\mathcal{B}'$ = Bind(M_j, O_i)
8: $\mathcal{U}'$ = DetermineRoute(O_i, M_j, $\mathcal{A}$)
9: $\mathcal{S}'$ = Schedule(O_i, $\mathcal{U}'$, t, $\mathcal{L}$)
10: t = the earliest time when an operation finishes
11: UpdateReadyList($\mathcal{G}', t$, *List*)
12: **until** *List* = $\emptyset$
13: **return** $\Psi' = \{\mathcal{O}, \mathcal{B}', \mathcal{P}', \mathcal{S}', \mathcal{U}'\}$

Figure 7: Online synthesis strategy

mixing operation O_{10} finishes earlier than *wcet*. As shown in Fig. 6, the computer will execute ONS to determine a new implementation. Operation O_{13} has the highest priority among all the ready operations. Module M_1 is the fastest available module (i.e., not occupied by other operations), hence O_{13} is bound to M_1. When scheduling the operation O_i, we consider two cases: (1) O_i has not yet started executing and (2) O_i has started executing but has not yet completed (i.e., the execution of O_i was interrupted by ONS). In case (1), the operation O_i is scheduled considering the routing time overhead and the corresponding *wcet* in the module library $\mathcal{L}$. In case (2), O_i has already executed partially, so we calculate the remaining percentage of execution for O_i (assuming it executes up to its *wcet*) and we scale its *wcet* accordingly. Then, we schedule O_i as in case (1).

When a scheduled operation finishes executing, *List* is updated with the operations that have become ready (line 11). The repeat loop terminates when the *List* is empty (line 12).

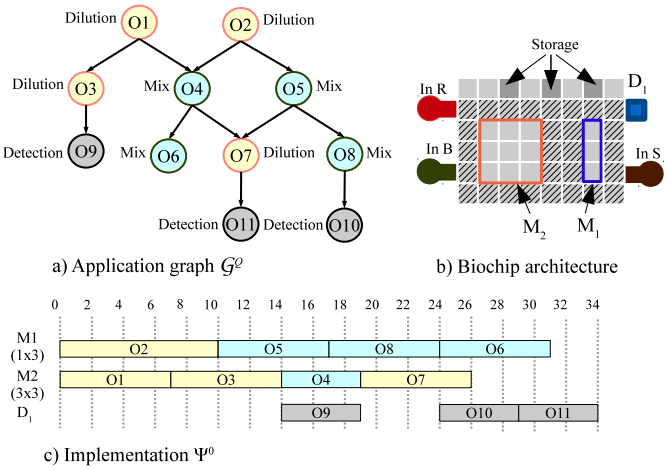


Figure 8: Quasi-static strategy example

6. Quasi-Static Synthesis strategy (QSS)

In this section, we present the Quasi-Static Synthesis strategy (QSS), which determines offline a set of implementations from which a particular implementation will be chosen at runtime, corresponding to the current operation execution scenario. The set of implementations is stored as a tree, where the nodes are the implementations, and the edges represent the conditions under which the controller will switch at runtime to a different implementation. The controller will use the sensing system to determine when the operations complete, and thus if a *switching condition* is fulfilled.

Let us consider the example in Fig. 8, where we have the application $\mathcal{G}^Q$ in Fig. 8a to be executed on the biochip in Fig. 8b, considering the module library in Table 1. The tree $\mathcal{T}^Q$ of alternative implementations for $\mathcal{G}^Q$ is depicted in Fig. 10, considering that the placement remains fixed as in Fig. 8b. We made the latter assumption for simplicity reasons.

To depict a solution Ψ in a node, we use the following notation: $\Psi = \{M_1 : O_1 ... O_m;\ M_2 : O_{m+1} ... O_n; M_k : O_p ... O_l\}$, where $M_i,\ i = 1..k$ are the allocated modules and each element has the structure $M_i : O_{m+1} ... O_n$, with $O_{m+1} ... O_n$ representing the order of the operations bound to M_i. The allocated modules for the example in Fig. 8 are M_1, M_2 and D_1. In Fig. 10, $\Psi^0 = \{M_1 : O_2 O_5 O_8 O_6;\ M_2 : O_1 O_3 O_4 O_7;\ D_1 : O_9 O_{10} O_{11}\}$ corresponds to the implementation in Fig. 8c, where operations O_2, O_5, O_8, and O_6 execute, in this order, on module M_1, operations O_1, O_3, O_4 and O_7 execute on M_2 and operations O_9, O_{10} and O_{11} execute on the detector D_1. The placement for M_1, M_2 and D_1 is presented in Fig. 8b.

As mentioned, an edge in $\mathcal{T}^Q$ captures a switching condition. We denote with t_i the moment in time when an operation O_i finishes executing. A *switching condition* on an edge is expressed as $t_i : I^i$, where $I^i = [t_A, t_B]$ represents an interval. Thus, if the finishing time t_i, as detected by the sensing system for O_i, is within $[t_A, t_B]$, then the controller will switch to the implementation following the edge. For example, in case operation O_2 finishes at $t_2 = 6\ s$, which is in the interval $[5, 7]$, the implementation Ψ^2 is loaded. Considering now Ψ^2 as the active implementation, in case O_4 finishes at $t_4 = 8\ s$, which is in the interval $[7, 12]$, the implementation Ψ^4 is loaded. When an implementation Ψ^i is loaded, we have all the information on allocation, binding, placement, scheduling and routing, which have been decided offline. However, depending on the actual execution time E_i of O_i, we need to adjust online the start times of operation in the schedule (the order does not change), subject to scheduling constraints.

The controller will only switch to a new implementation if a switching condition is active. Otherwise, it continues to run the currently active implementations until its completion. For example, in case of the mentioned scenario ($t_2 = 6\ s$, $t_4 = 8\ s$), the active implementation Ψ^4 continues to run until its completion, thus completing the application in 30 *s*.

Our proposed QSS is presented in Fig. 9 and has both an offline and an online component. Most of the work is done in the offline part, QSS-offline, which takes as input an application graph $\mathcal{G}$, a biochip architecture $\mathcal{A}$ and a module library $\mathcal{L}$ and outputs the tree of implementations $\mathcal{T}^Q$.

The online part QSS-online is responsible for loading the implementation corresponding to the current execution scenario. The QSS-online function from Fig. 9 is called by the controller every time an operation O_i finishes. The function checks if the switching condition $t_i : I^i$ is fulfilled, and if so, it loads the corresponding implementation Ψ^i and adjusts its schedule consid-

QSS-offline($\mathcal{G}, \mathcal{A}, \mathcal{L}$)
1: Ψ^0 = TabuSearchSynthesis($\mathcal{G}, \mathcal{A}, \mathcal{L}$)
2: $\mathcal{T}^Q$ = DTI($\emptyset, \Psi^0, \mathcal{G}, \mathcal{A}, \mathcal{L}$)
3: **return** $\mathcal{T}^Q$

QSS-online(O_i, I^i, $\mathcal{T}^Q$)
1: **if** ($t_i : I^i$) **then**
2: Ψ^i = GetImplementation($\mathcal{T}^Q$, O_i, I^i)
3: AdjustSchedule(Ψ^i, O_i, t_i)
4: LoadImplementation(Ψ^i)
5: **end if**

Figure 9: Quasi-static synthesis strategy

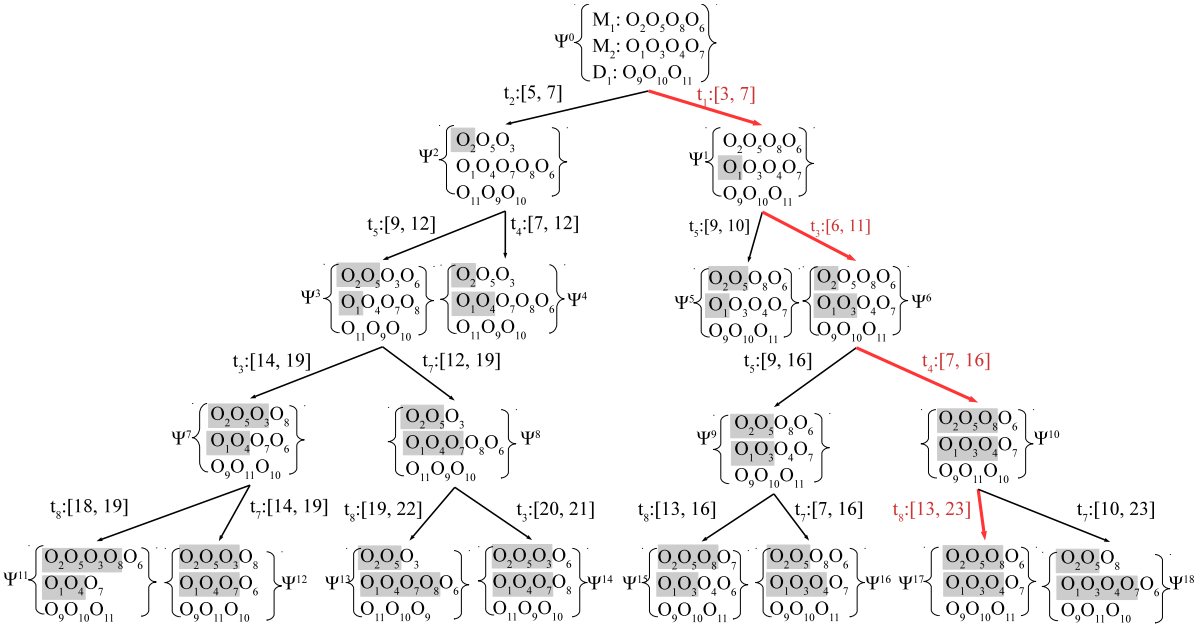


Figure 10: Tree of implementations $\mathcal{T}^{\mathcal{Q}}$

ering the finishing time t_i of O_i.

QSS-offline has two steps: it first determines the implementation Ψ^0 of the application considering the *wcet* of operations (line 1) and then, starting from Ψ^0, QSS-offline builds the tree of implementations $\mathcal{T}^{\mathcal{Q}}$ (line 2). We use the Tabu Search approach from [25] to derive Ψ^0, and the next subsection presents how the tree $\mathcal{T}^{\mathcal{Q}}$ of implementations is determined.

6.1. Determining $\mathcal{T}^{\mathcal{Q}}$

In order to determine the tree of implementations, QSS-offline calls the Determine Tree of Implementations (DTI) function (line 2 in Fig. 9). For the implementation of DTI we have adapted and extended the algorithm in [29] proposed for constructing the optimal tree of implementations for multiprocessor systems. A tree is optimal if it covers all the possible combinations of execution order of operations. However, deriving an optimal tree is infeasible: it may take too long time and may not fit in the microcontroller memory even if compressed. Hence, we first present the approach for determining the optimal tree (Fig. 11) and then discuss a heuristic for reducing the tree's size (Section 6.2).

DTI($\mathcal{T}^{\mathcal{Q}}, \Psi, \mathcal{G}, \mathcal{A}, \mathcal{L}$)

1: $\mathcal{C}$ = DetermineConcurrentOperations(Ψ)
2: **for each** $O_i \in C$ **do**
3: I^i = the interval when O_i completes first
4: $\mathcal{G}^i$ = RemoveFromGraph($\mathcal{G}, O_i, \Psi$)
5: Ψ^i = LSSynthesis($\mathcal{G}^i, \mathcal{A}, \mathcal{L}, \Psi, I^i$)
6: InsertInTree($\mathcal{T}^{\mathcal{Q}}, \Psi^i, I^i$)
7: DTI($\mathcal{T}^{\mathcal{Q}}, \Psi^i, \mathcal{G}, \mathcal{A}, \mathcal{L}$)
8: **end for**
9: **return** $\mathcal{T}^{\mathcal{Q}}$

Figure 11: Algorithm for determining the tree of implementations

The DTI function presented in Fig. 11 is a recursive function that returns $\mathcal{T}^{\mathcal{Q}}$. DTI takes as input the tree of implementations $\mathcal{T}^{\mathcal{Q}}$, the currently determined implementation Ψ, the application graph $\mathcal{G}$, the biochip architecture $\mathcal{A}$ and the module library $\mathcal{L}$.

DTI is called with the current implementation Ψ. For the example in Fig. 8, the implementation is Ψ^0, which is the current implementation that executes on the biochip. DTI needs to determine what alternative implementations can arise from Ψ^0 and which are their switching conditions. The alternative implementations depend on when operations will finish executing. For Ψ^0, we have two concurrent operations that start executing at time 0: O_1 on the 3×3 M_2, which according to Table 1 has a *bcet* and *wcet* of 3 *s* and 7 *s*, and O_2 which on the 1×3 M_1 module has a *bcet* and *wcet* of 5 *s* and 10 *s*. From Ψ^0, two alternatives are possible: Ψ^1 and Ψ^2, see Fig. 10. Thus, DTI identifies the set of concurrent operations $\mathcal{C}$, line 1 in Fig. 11, that currently execute on the biochip. Currently, $\mathcal{C} = \{O_1, O_2\}$.

For each such operation O_i we have an outgoing edge for the current implementation Ψ. These edges are labeled with the switching conditions $t_i : [t_A, t_B]$. As mentioned, when the sensing system determines that O_i has finishes executing, we check if its finishing time t_i is within the interval $I^i = [t_A, t_B]$, and if true, we load the corresponding implementation Ψ^i. The interval I^i is determined in line 3 in Fig. 11. For our example, considering the determined set of concurrent operations $\mathcal{C} = \{O_1, O_2\}$, we determine $I^1 = [3, 7]$ for the case when O_1 finishes first, and $I^2 = [5, 7]$ for the case when O_2 finishes first.

We denote with $\mathcal{G}^i$ the graph that contains only the partially executed operations and the operations that have not executed. The graph $\mathcal{G}^i$ is determined in line 4. For our example, $\mathcal{G}^1$ is obtained by removing O_1 in the graph $\mathcal{G}^{\mathcal{Q}}$ (Fig. 8a).

The obtained $\mathcal{G}^i$ is then synthesized to determine the implementation Ψ^i at line 5. An alternative implementation Ψ^i is obtained through synthesis by deriving a new allocation $\mathcal{O}^i$, binding $\mathcal{B}^i$, placement $\mathcal{P}^i$, schedule $\mathcal{S}^i$ and routing $\mathcal{U}^i$ for the

operations that have not started executing. All the other operations (i.e., finished of currently executing) keep the same O, $\mathcal{B}$, $\mathcal{P}$, $\mathcal{S}$ and $\mathcal{U}$ as in Ψ. Since the tree $\mathcal{T}^Q$ can grow very large, we have decided to use a List Scheduling-based implementation, for the synthesis, which has been shown to provide good quality results in a very short time [14]. The node Ψ^i and edge $t_i : I^i$ are inserted in the tree $\mathcal{T}^Q$ in line 6.

Finally, DTI is called recursively for Ψ^i (line 7 in Fig. 11).

Fig. 10 shows the obtained tree of implementations $\mathcal{T}^Q$ for our example. For each implementation, we marked with gray the operations that have completed executing. At runtime, assuming the execution scenario $t_1 = 5\ s, t_3 = 10\ s, t_4 = 15\ s, t_8 = 20\ s$, we activate in $\mathcal{T}^Q$ the path marked in Fig. 10 with red arrows of thicker width.

Next, we calculate the complexity of our proposed DTI. The function DetermineConcurrentOperations (line 1 in Fig. 11) has the complexity $O(V!)$ [29], where V is the number of nodes in the graph. The List Scheduling-based synthesis has the complexity $O(MN)$ [11], where M, N are the width and the length of the biochip area. We implemented the graphs using incidence lists, hence removing a node from the graph (line 4) has the complexity $O(E)$, where E is the number of edges in the graph. Inserting the node in the graph (line 6) has the complexity $O(1)$. For our problem $M,N<<V$, hence our heuristic has a time complexity $O(V!)$. The space complexity of DTI is also $O(V!)$.

6.2. Reducing the size of the tree

By using DTI from Fig. 11 we obtain the optimal tree of implementations $\mathcal{T}^Q$, i.e., it contains implementations for all possible combinations of execution order of operations. As we discussed, deriving and using the optimal tree is infeasible. However, using a partial tree may result in larger application completion times $\delta_{\mathcal{G}}$, since QSS will not be able to adapt as needed: it will have fewer implementations to choose from. The challenge is to reduce the size of the tree such that the negative impact on $\delta_{\mathcal{G}}$ is minimized. For this purpose, we have adapted the approach from [29] called DIFF (from "different"), which limits the size of the tree to a given limit, and favors implementations that are more different than their parents, because we want a larger variety of implementations stored in $\mathcal{T}^Q$.

DIFF relies on a priority function which gives a higher priority to implementations that have less similarities to their parents in $\mathcal{T}^Q$. Considering the DIFF approach, the following modifications are performed to the DTI function in Fig. 11:

(a) DTI will now stop if the tree reaches a maximum size W, which is decided such that the compressed tree would fit in the biochip microcontroller.

(b) When deciding on the successors of the implementation Ψ (lines 2–8 in Fig. 11), DTI will sort the alternatives based on their priority.

(c) DTI will call itself recursively on the highest priority implementations first.

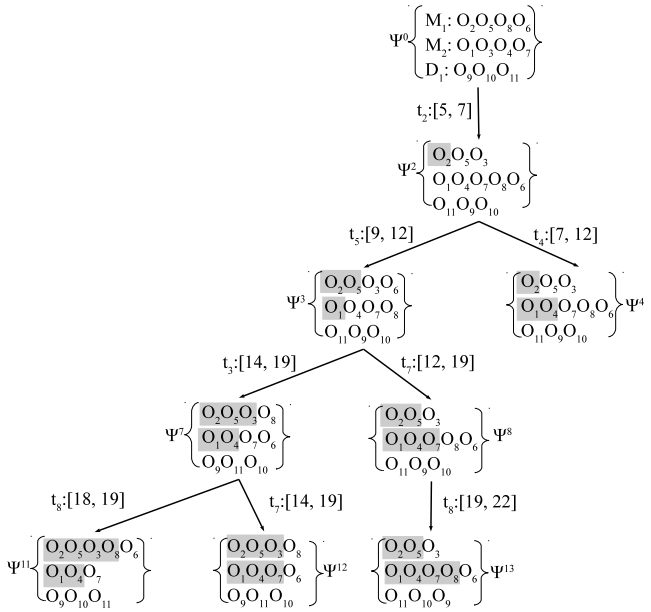


Figure 12: Restricted tree of implementations $\mathcal{T}^R$ (size $W = 9$)

To determine the priority of an implementation Ψ^k, DTI will compare Ψ^k to Ψ by looking at the binding, placement and order of operations. Hence, the priority function is modeled as a weighted sum over all operations in Ψ^k, where a higher weight is considered for case when the binding or the placement are different than in Ψ, than for the case when only the order of operations differs.

For example, let us build the tree $\mathcal{T}^R$ restricted to a maximum size W = 9 for the application where the complete tree is the one in Fig. 10. The restricted tree $\mathcal{T}^R$ is constructed starting with the initial implementation Ψ^0, the root of the tree. From Ψ^0, we derive Ψ^1 and Ψ^2—alternative implementations depending on the completion time of operations O_2 and O_1. The implementations Ψ^1 and Ψ^2 are inserted in $\mathcal{T}^R$. Then, using DIFF, the priority function decides which of the two implementations, Ψ^1 or Ψ^2, is processed next. Implementation Ψ^2 differs from Ψ^0 in the binding and the order of operations, while Ψ^1 is identical to Ψ^0. Thus Ψ^2 is given priority over Ψ^1. The alternative implementations Ψ^3 and Ψ^4 derived from Ψ^2, are added to $\mathcal{T}^R$. The algorithm stops when the tree reaches the maximum size $W = 9$. Fig. 12 depicts the obtained restricted tree $\mathcal{T}^R$.

7. Evaluation results

For experiments we used two synthetic benchmarks (SB_1 and SB_2) and four real-life applications: (1) the mixing stage of polymerase chain reaction (PCR, 7 operations), (2) in-vitro diagnostics on human physiological fluids (IVD, 28 operations), (3) the interpolation dilution of a protein (IDP, 71 operations) and (4) the colorimetric protein assay (CPA, 103 operations). The application graphs and the descriptions of the bioassays can be found in [20] for CPA, PCR and IVD, in [30] for IDP and in [25] for SB_1 and SB_2. The algorithms were implemented in Java (JDK 1.6) and run on a MacBook Pro computer with

Table 2: Comparison of offline (no variability) approach vs. ONS and QSS

App.	Arch.	$\delta_{\mathcal{G}}^{OFF}(s)$	$k = 30\%$			$k = 50\%$			$k = 70\%$		
			$\delta_{\mathcal{G}}^{ONS}(s)$	Imp.(%)	$\delta_{\mathcal{G}}^{QSS}(s)$	$\delta_{\mathcal{G}}^{ONS}(s)$	Imp.(%)	$\delta_{\mathcal{G}}^{QSS}(s)$	$\delta_{\mathcal{G}}^{ONS}(s)$	Imp.(%)	$\delta_{\mathcal{G}}^{QSS}(s)$
PCR	8×8 (1,1,1,0)	8.12	avg. 7.1 dev. 0.01	12.55	avg. 7.12 dev. 0.03	avg. 6.61 dev. 0.48	19	avg. 6.63 dev. 0.48	avg. 6.19 dev. 0.24	24	avg. 6.2 dev. 0.24
IVD	9×8 (1,1,1,1)	193.14	avg. 159.91 dev. 3.24	17.2	avg. 185.21 dev. 10.95	avg. 156.83 dev. 3.99	18.7	avg. 182.64 dev. 11.28	avg. 156.62 dev. 1.91	18.91	avg. 180.86 dev. 10.89
SB_1	10×11 (2,2,2,0)	36.42	avg. 31.7 dev. 0.61	12.95	avg. 36.16 dev. 0.46	avg. 30.7 dev. 1.14	19.66	avg. 34.9 dev. 1.56	avg. 28.18 dev. 1.5	22.63	33.66 dev. 1.99
SB_2	11×12 (2,2,2,2)	76.65	avg. 66.15 dev. 0.68	11.39	avg. 73.28 dev. 1.96	avg. 63.27 dev. 1.77	15.24	avg. 72.18 dev. 2.62	avg. 62.85 dev. 1.69	15.8	avg. 71.34 dev. 2.92

Intel Core 2 Duo CPU at 2.53 GHz and 4 GB of RAM. Both the simulation of the application execution and the online synthesis strategy were executed on the mentioned hardware. Unless specified, we used the experimentally determined module library from Table 3.

7.1. Comparison between ONS and QSS

In the first set of experiments we were interested to determine if the proposed approaches, ONS and QSS, can successfully handle the variability in operation execution times. We have simulated for PCR, IVD and SB_{1-2} applications a series of scenarios where $k = 30\%$, 50% and 70% of the operations finish executing before their *wcet*. We have generated between 35 and 1000 execution scenarios depending on the size of the applications and the number of operations that finish earlier than *wcet*. Table 2 presents the results. The biochip size 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, reagents and optical detectors, respectively.

Thus, we have simulated the execution of PCR, IVD, SB_1 and SB_2 on the specified architectures, and we have randomly generated an execution time between *bcet* and *wcet* for k percentage of operations. For each simulation, we adapt to the variability using both ONS and QSS, obtaining an application completion time $\delta_{\mathcal{G}}^{ONS}$ and $\delta_{\mathcal{G}}^{QSS}$, respectively. In Table 2 we report the obtained average (avg.) application completion time and the mean deviation (dev.) for ONS and QSS approaches as follows: in columns 4, 7 and 10 we present the avg. and dev. over all the simulation scenarios for $k = 30\%$, 50% and 70%. The mean deviation is calculated as the average over the absolute values of deviations from the average completion time.

Table 3: Module library $\mathcal{L}$ used for experiments

Operation	Module area	*bcet* (s)	*wcet* (s)
Mix	2×5	1	2
Mix	2×4	2	3
Mix	3×3	3	5
Mix	1×3	5	7
Mix	2×2	7	10
Dilution	2×5	3	4
Dilution	2×4	2	5
Dilution	3×3	3	7
Dilution	1×3	6	10
Dilution	2×2	7	12
Optical detection	1×1	25	30
Dispensing	N/A	5	7

The reported $\delta_{\mathcal{G}}^{ONS}$ times take into account the runtime overhead required by re-synthesis (for all cases). The ONS runtime varies between 10 *ms* and 270 *ms*. We have ignored the runtime overhead required by QSS. Also, for QSS, we have limited the size of the tree to 100, and we have implemented the DIFF approach (Section 6.2). We are interested to determine the advantages of using ONS and QSS over the offline approach (OFF), which uses the *wcet*s for execution times. The application completion time $\delta_{\mathcal{G}}^{OFF}$ obtained with the offline solution is reported in the third column in Table 2. In columns 5, 8 and 11 we report the percentage improvement (Imp.) of ONS over OFF, calculated as $\frac{\delta_{\mathcal{G}}^{OFF}-\delta_{\mathcal{G}}^{ONS}}{\delta_{\mathcal{G}}^{OFF}}\times100$. As we can see, ONS is able to exploit the slack resulted from operations finishing before their *wcet* to significantly improve the completion time $\delta_{\mathcal{G}}^{ONS}$ over the offline solution, $\delta_{\mathcal{G}}^{OFF}$. For example, for PCR we have obtained a percentage improvement of 12.55%, 19% and 24% for $k = 30\%$, 50% and 70%, respectively. We can see that as the number of operations experiencing variability is increasing, ONS can shorten the application completion times $\delta_{\mathcal{G}}$. Note that the opportunity for improvement is influenced also by the differences between the *bcet* and *wcet* values in Table 3, which are not very far apart for these experiments.

However, there are situations when an online solution cannot be used (e.g., the biochip microcontroller cannot run an algorithm such as ONS) or a design-time solution, such as our proposed QSS is preferred. As we can see from Table 2, QSS is a viable alternative to ONS. The results obtained by QSS are significantly better than OFF. However, QSS (with the DIFF approach that limits the tree size to 100 implementations) obtains worse results than ONS because QSS is limited in its ability to adapt at runtime.

7.2. Comparison to the related work

In [11], the only work that addresses variability in operation execution, which is a synthesis approach called Operation-

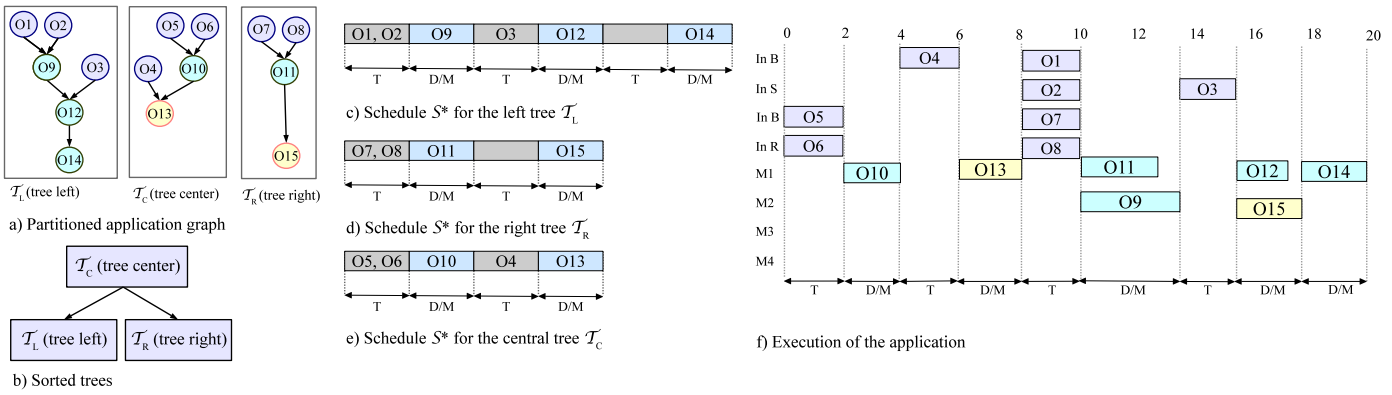


Figure 13: OIA synthesis example

Interdependency-Aware (OIA) used offline to determine an implementation $\Psi = \{\mathcal{O}, \mathcal{B}, \mathcal{P}, \mathcal{S}^*, \mathcal{U}\}$, where $\mathcal{O}, \mathcal{B}, \mathcal{P}$ and $\mathcal{U}$ are the allocation, binding, placement and routing as defined earlier, while $\mathcal{S}^*$ is a partial schedule which contains only the order of operations. At runtime, a sensing system is used to signal when an operation has finished executing. The partial schedule $\mathcal{S}^*$ is scaled accordingly to adjust to the actual execution times of the operations. Let us illustrate OIA, on the same example, i.e., considering the application $\mathcal{G}$ in Fig. 2, executing on the biochip in Fig. 3.

The OIA synthesis in [11] 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. 2, 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. 13a.

(2) Next, the OIA synthesis is applied to each of the trees obtained at step 1. OIA synthesis executes the operations in phases, namely the transport (T) phase and the dilution/mixing (D/M). The T phase consists of the routing and dispensing operations, while the D/M phase consists of the 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. The schedules obtained using OIA for the three trees $\mathcal{T}_L$, $\mathcal{T}_C$ and $\mathcal{T}_R$ are presented in Fig. 13c–e.

(3) The directed trees are sorted so that they do not present scheduling and placement conflicts. Using the sorting algorithm proposed in [11], we obtained for our example the sorted order in Fig. 13b.

(4) Finally, the synthesis results for the trees are merged according to the sorted order obtained during the previous step.

The execution of the application in our example is depicted in Fig. 13f, considering the execution times from Fig. 4b, where operations O_{10-15} finish sooner. The first phase is 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 execution length of a phase is given by the operation O_i that has the longest execution time among all the operations in the same phase. In case the longest operation O_i finishes earlier, then the length of its phase, and thus the application completion time, are reduced. However, if other operations in the phase finish earlier, the application completion time is not reduced. As seen in Fig. 13f, the application completion time $\delta_{\mathcal{G}} = 19.9\ s$, which is actually larger than the completion time obtained by the offline solution (13 s, see Fig. 4a). Note that the advantage of OIA is that it does not need the *wcet*s, so it works even without a library $\mathcal{L}$ of modules.

Table 4: Comparison of proposed strategies (ONS, QSS) and OIA [11]

App.	Arch.	$\delta_{\mathcal{G}}^{OIA}(s)$	$\delta_{\mathcal{G}}^{ONS}(s)$	Improv.(%)	$\delta_{\mathcal{G}}^{QSS}(s)$	Improv.(%)
PCR	8×8	25	18.56	25.76	19.10	23.6
IDP	10×10	154	116.93	24	117.62	23.6
CPA	10×10	172	120.02	30.2	122.15	28.98

In the last set of experiments we were interested to compare ONS and QSS to the related work (OIA). Thus, we compared the completion time $\delta_{\mathcal{G}}^{ONS}$ and $\delta_{\mathcal{G}}^{QSS}$ obtained by the online approach and QSS with $\delta_{\mathcal{G}}^{OIA}$ obtained using the OIA approach. For a fair comparison, we adapted our online synthesis strategy and QSS to match the assumptions in [11] as follows: (i) we have used the same module library from [11], (ii) we have considered the same assumptions for operation execution as in [11], i.e., if an operation O_i finishes earlier than its *wcet*, then $E_i = 1.1 \times bcet$ and (iii) we have considered that the probability that the execution time of an operation finishes before *wcet* is 0.5.

Based on the mentioned assumptions, we have simulated the execution of all the applications considered in [11], namely PCR, IDP and CPA. In Table 4 we present the architecture used for each application (column 2), and the obtained application completion times: $\delta_{\mathcal{G}}^{OIA}$ [11] (column 3), $\delta_{\mathcal{G}}^{ONS}$ (column 4) and $\delta_{\mathcal{G}}^{QSS}$ (column 6). We have used 2 optical detectors for IVD and 4 optical detectors for CPA. The application completion time $\delta_{\mathcal{G}}^{ONS}$, obtained using our online approach, includes the runtime overhead due to re-synthesis. In columns 5 and 7, we show the percentage improvements of ONS and QSS over OIA. The results presented in Table 4, show that our proposed strategies can obtain better results than OIA. For example, for CPA, with ONS we have obtained an improvement of 30.2%, and with QSS and improvement of 28.98%.

8. Discussion and conclusions

In this paper we have addressed the problem of variability in the execution times of the operations. We have proposed two strategies that can handle such variabilities: an online approach and an approach based on a quasi-static strategy. The online approach (Section 5) re-synthesizes the application at runtime whenever an operation experiences variabilities in execution time. Aware of the actual execution times of the operations, the online approach can take full advantage of the current configuration to derive the appropriate implementation such that the application completion time is minimized. The drawback of the online approach is the requirement of a powerful computer connected to the biochip to run ONS and the overhead introduced by the re-syntheses performed at runtime.

Our second proposed strategy, QSS (Section 6), avoids such overhead by deriving offline a tree of alternative implementations. At runtime, the implementation corresponding to the actual execution scenario is loaded. Ideally, the tree of implementations would contain solutions that cover all possible execution scenarios. However, due to limited memory requirements, we use the DIFF technique (Section 6.2) to reduce the size of the table such that it satisfies the memory requirements while still covering a large variety of execution scenarios. Consequently, the drawback of QSS is that the resulted implementation may not take full advantage of the execution time variability.

The experiments performed on four real-life case studies and two synthetic benchmarks show that both ONS and QSS can handle variabilities in operation execution time. The results reflect the advantages and disadvantages of ONS and QSS approaches. We have also compared our strategies with prior work and have obtained better results for both approaches.

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

[2] 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.

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

[4] 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.

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

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

[7] 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.

[8] 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.

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

[10] Pranab Roy, Hafizur Rahaman, Chandan Giri, and Parthasarathi Dasgupta. Modelling, detection and diagnosis of multiple faults in cross referencing dmfbs. In *Proceedings of the International Conference on Informatics, Electronics and Vision*, pages 1107–1112, 2012.

[11] 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.

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

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

[14] 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.

[15] 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.

[16] Tsung-Yi Ho, Krishnendu Chakrabarty, and Paul Pop. Digital microfluidic biochips: recent research and emerging challenges. In *Hardware/Software Codesign and System Synthesis (CODES+ ISSS), 2011 Proceedings of the 9th International Conference on*, pages 335–343. IEEE, 2011.

[17] 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.

[18] 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.

[19] 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.

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

[21] Bhargab B Bhattacharya, Sudip Roy, and Sukanta Bhattacharjee. Algorithmic challenges in digital microfluidic biochips: Protocols, design, and test. In *Applied Algorithms*, pages 1–16. Springer, 2014.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

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

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

[28] 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.

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

[30] 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.