Des Autom Embed Syst (2010) 14: 287–307
DOI 10.1007/s10617-010-9059-x

Tabu search-based synthesis of digital microfluidic biochips with dynamically reconfigurable non-rectangular devices

Elena Maftei · Paul Pop · Jan Madsen

Received: 12 May 2010 / Accepted: 28 June 2010 / Published online: 21 July 2010

Abstract Microfluidic biochips are replacing the conventional biochemical analyzers, and are able to integrate on-chip all the necessary functions for biochemical analysis. The "digital" microfluidic biochips are manipulating liquids not as a continuous flow, but as discrete droplets, and hence they are highly reconfigurable and scalable. A digital biochip is composed of a two-dimensional array of cells, together with reservoirs for storing the samples and reagents. Several adjacent cells are dynamically grouped to form a virtual device, on which operations are performed. So far, researchers have assumed that throughout its execution, an operation is performed on a rectangular virtual device, whose position remains fixed. However, during the execution of an operation, the virtual device can be reconfigured to occupy a different group of cells on the array, forming any shape, not necessarily rectangular. In this paper, we present a Tabu Search metaheuristic for the synthesis of digital microfluidic biochips, which, starting from a biochemical application and a given biochip architecture, determines the allocation, resource binding, scheduling and placement of the operations in the application. In our approach, we consider changing the device to which an operation is bound during its execution, to improve the completion time of the biochemical application. Moreover, we devise an analytical method for determining the completion time of an operation on a device of any given shape. The proposed heuristic has been evaluated using a real-life case study and ten synthetic benchmarks.

Keywords Microfluidics · Biochips · Reconfigurability · Synthesis

1 Introduction

Microfluidic biochips (also referred to as lab-on-a-chip) 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, transport, splitting, merging, dispensing, mixing, and detection [8].

E. Maftei (✉) · P. Pop · J. Madsen
Technical University of Denmark, 2800, Kgs. Lyngby, Denmark
e-mail: em@imm.dtu.dk

Biochips offer a number of advantages over conventional biochemical procedures. By handling small amount of fluids, they provide higher sensitivity while decreasing reagent consumption, hence reducing cost. Moreover, due to their miniaturization and automation, they can be used as point-of-care devices, in areas that lack the infrastructure needed by conventional laboratories [21].

Due to these advantages, biochips are expected to revolutionize clinical diagnosis, especially immediate point-of-care diagnosis of diseases. Other emerging application areas include drug discovery, DNA analysis (e.g., polymerase chain reaction and nucleic acid sequence analysis), protein and enzyme analysis and immuno-assays. Microfluidic devices can also be used for environment monitoring, by pathogen detection in air or water samples [21].

There are two generations of microfludic biochips. The first generation is based on the manipulation of continuous liquid through fabricated micro-channels, using external pressure sources or integrated mechanical micro-pumps [21]. Although adequate for many simple biochemical applications, their integrated micro-structures make continuous-flow biochips unsuitable for more complex applications, requiring complicated fluid manipulations [5]. The second generation is based on the manipulation of discrete, individually controllable droplets, on a two-dimensional array of identical cells. The actuation of droplets is performed without the need of micro-structures, leading to increased scalability and flexibility compared with continuous-flow biochips [14]. This generation is also referred to as "digital microfluidics", due to the analogy between the droplets and the bits in a digital system. Such biochips, consisting of hundreds [1] and thousands [16] of cells have already been successfully designed and commercialized. In this paper, we are interested in the second generation, droplet-based digital biochips.

1.1 Related work

Researchers have initially addressed separately architectural and physical-level synthesis of DMBs. Su and Chakrabarty [17] have proposed an integer linear programming (ILP) model for scheduling and binding, considering a given allocation, but without addressing placement and routing. During the physical-level synthesis, the placement [19, 25] of each module on the microfluidic array and the droplets routes [6, 20] have to be determined.

A unified high-level synthesis and module placement methodology has been proposed in [18], where the focus has been on deriving an implementation that can tolerate faulty cells in the biochip array. Their algorithm was modified in [22] to include droplet-routing-aware physical design decisions. Yuh et al. [24] have proposed a synthesis and placement algorithm which uses a tree-based topological representation and is able to improve on the results from [18]. The algorithm has later been extended to consider defective cells on the biochip array [25]. In [10] we have proposed a unified ILP-based architectural-level synthesis and placement approach for DMBs, that although produces the optimal solution, is only feasible for limited problem sizes.

The combined architectural- and physical-synthesis problem has some similarities with the simultaneous scheduling and placement problem of dynamically reconfigurable field-programmable gate arrays (DR-FPGAs) [2], which is typically formulated as a 3D packing problem that minimizes the volume, seen as area $\times$ execution time. Bazargan et al. [2] have proposed offline algorithms for statically reconfigurable FPGAs and online algorithms for dynamically reconfigurable FPGAs. Yuh et al. [26] use a 3D transitive closure subGraph for the 3D packing problem. Their earlier work on temporal floorplanning using a tree-based topological representation [23] has been extended for DMBs [24].

However, there are three main differences when doing scheduling and placement for digital microfluidic biochips (DMBs): (1) operations that are executed on virtual devices

(created by grouping adjacent cells) can easily be re-assigned to a different group of cells *during the execution of operations* without incurring a significant overhead—see Sect. 2.1 for details; (2) non-reconfigurable devices, such as reservoirs and detectors also have to be considered; and (3) additional operations have to be introduced to temporarily store a droplet in-between operations that are not scheduled at consecutive time-steps.

1.2 Contribution

In this paper, we propose a Tabu Search-based synthesis approach that, starting from a biochemical application modeled as a sequencing graph and a given biochip array, determines the allocation, resource binding, and scheduling of the operations in the application at the same time with module placement. All of previous approaches to DMBs and DR-FPGAs assume that a reconfigurable operation is performed on a rectangular virtual module whose placement and shape remain fixed throughout the execution of the operation. However, our scheduling and placement steps consider that *during its execution*, an operation can be re-assigned to another module having different location and shape, in order to improve the biochemical application completion time on the given biochip array. Moreover, we propose an analytical method for determining the completion time of an operation on a device of any given shape. We show that by allowing operations to be re-assigned to modules of any shape during their execution, significant improvements can be obtained in the application completion time, allowing us to use smaller area biochips and thus reduce costs.

The paper is organized in six sections. Section 2.1 presents the architecture of a digital microfluidic biochip. We propose a method for determining the completion time of an operation on a virtual device in Sect. 2.2. We introduce the abstract model used to capture a biochemical application in Sect. 2.3. We formulate the problem in Sect. 3 and illustrate the design tasks using several examples. The proposed approach is presented in Sect. 4 and evaluated in Sect. 5. The last section presents our conclusions.

2 System model

2.1 Biochip architecture

In a digital microfluidic biochip the manipulation of liquids is performed using discrete droplets. There are several mechanisms for droplet manipulation [8]. Our work considers electrowetting-on-dielectric (EWD) [14], but can be extended to handle other techniques as well. EWD is the most promising technique, and can provide high droplet speeds of up to 20 cm/s [14]. A biochip is composed of several cells, see Fig. 1b. The schematic of a cell is presented in Fig. 1a. The droplet is sandwiched between two glass plates (the top plate and the bottom plate), and moves within a filler fluid. The top plate contains a single ground electrode, while the bottom plate has several control electrodes. The electrodes are insulated from the droplet through an insulation material. With EWD, the movement of droplets is controlled by applying voltages to the required electrodes. For example, turning off the middle control electrode and turning on the right control electrode in Fig. 1a will force the droplet to move to the right. For the details on EWD, the reader is directed to [14].

Several cells are put together to form a two-dimensional array (an example architecture is presented in Fig. 1b). Using EWD manipulation, droplets can be moved to any location without the need for pumps and valves, which are required in a continuous-flow biochip. Besides the basic cell discussed previously, a chip typically contains input and output ports

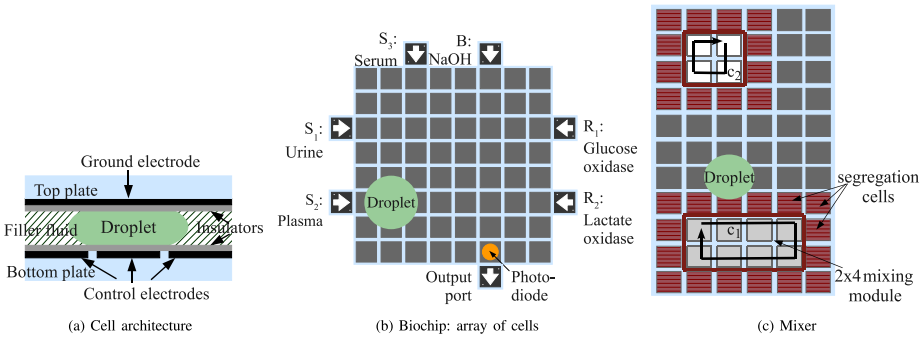


Fig. 1 Biochip architecture

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

Using this architecture, and changing correspondingly the control voltages, all the basic microfluidic operations, such as transport, splitting, merging, dispensing, mixing, and detection, can be performed. For example, mixing is done by bringing two droplets to the same location and merging them, followed by the transport of the droplet over a series of electrodes. By moving the droplet, external energy is introduced, creating complex flow patterns (due to the formation of multilaminates), thus leading to a faster mixing [13]. The operation can be executed by routing the droplet inside a virtual module, created by grouping adjacent cells. Any cells can be used for this purpose, thus we say that the operation is "reconfigurable".

In order to prevent the accidental merging of a droplet with another droplet in its vicinity, a minimum distance must be kept between operations executing on the microfluidic array. These fluidic constraints are enforced by surrounding a module by a 1-cell segregation area (the hashed area), containing cells that can not be used until the operation executing on the device is completed. The role of the segregation area will be further discussed in Sect. 3.1.

An example of a mixer device is shown in Fig. 1c, where a mixing operation is performed by routing the droplet inside the 2×4 module. However, due to reconfigurability of the mixing, the same operation can be executed on a different group of electrodes, for example the 2×2 mixer shown in the same figure, the only difference consisting in the completion time for the operation. We consider that designers will build and characterize a module library $\mathcal{L}$, where for each operation there are several options with varying areas and execution times. For example, the module library in Table 1 shows the completion times for executing the mixing operation on the 2×4 and on the 2×2 devices, based on the experiments performed in [13].

So far, it has been considered that a reconfigurable operation is performed on a rectangular group of adjacent cells on the microfluidic array, representing one of the virtual devices in the module library. Therefore, the number of electrodes used for an operation and their location on the microfluidic array were fixed throughout the execution. However, because of the virtual character of the devices, in this paper we consider that during a reconfigurable

Table 1 Module library

Operation	Area (cells)	Time (s)
Mixing/Dilution	2×4	2.9
Mixing/Dilution	1×4	4.6
Mixing/Dilution	2×3	6.1
Mixing/Dilution	2×2	9.95

operation the droplet can be routed to another group of electrodes, where the operation can be continued, possibly on a device of a different shape.[1] Let us consider the example in Fig. 1c. We assume that 2 s after the mixing operation started executing on the 2×4 virtual module, with the droplet being on the cell denoted by c_1, we decide to change the position at which the operation is performed and the number of electrodes used for mixing. In our example, the droplet will be routed to the nearest position belonging to the new group of cells, c_2, where it will continue executing. As mixing is performed by routing, the operation is not interrupted while being transported between the two positions, however, for simplicity, we ignore the percentage of mixing obtained during transport. As the operation was executed for only 2 s out of the 2.9 s required for completion on the 2×4 module (see Table 1), only 68.96% of the mixing was performed. Therefore, the operation will continue to execute on the new 2×2 group of cells, until is done. Considering the completion time of the mixing operation on a 2×2 module of 9.95 s as shown in Table 1, the remaining 31.04% of the mixing is obtained by routing the droplet inside the 2×2 module for 3.08 s. In the end, the completion time for the operation is 5.08 s.

One aspect that must be considered when changing the location at which an operation is executed is the additional time required to transport the droplet between the two positions. In this paper we consider the data[2] from [14], which allows us to approximate that the time required to route the droplet one cell is 0.01 s, which is an order of magnitude smaller than operation execution times, see Table 1. The routing overhead will be considered during the synthesis process.

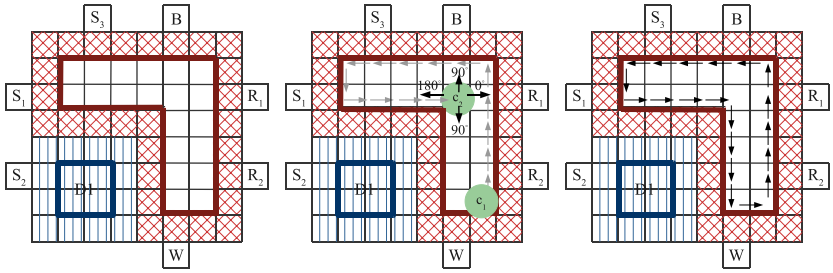
A biochemical application may also contain "non-reconfigurable" operations, that are executed on real devices, such as reservoirs or optical detectors.

2.2 Characterizing non-rectangular virtual modules

Table 1 gives completion times for performing reconfigurable operations on various areas. The experiments have considered a limited set of devices, of rectangular shape. However, reconfigurable operations can be executed by routing the droplet on any route, as shown in Fig. 2a, where a mixing operation is executed on a "L-shaped" virtual module. Since the virtual modules can consist of a varying number of electrodes, arranged in any form, characterizing all devices through experiments is time consuming. Moreover, the completion time of an operation is also influenced by the route taken by the droplet, inside the module, during the execution of the operation. Therefore, in this section we propose an analytical method for determining how the percentage of mixing varies depending on the movement of the droplet inside a module of a given size and shape. Our method provides safe estimates by decomposing the devices from Table 1.

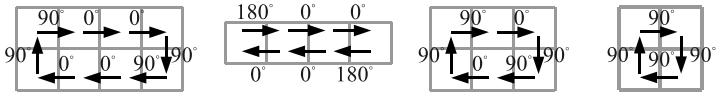
[1] Non-rectangular shaped devices will be discussed in the next section.

[2] Electrode pitch size = 1.5 mm, gap spacing = 0.3 mm, average linear velocity = 20 cm/s.



(a) L-shaped mixer (b) Droplet movement (c) Mixing pattern

Fig. 2 Execution of a mixing operation



(a) 2 × 4 module (b) 1 × 4 module (c) 2 × 3 module (d) 2 × 2 module

Fig. 3 Characterization of module library

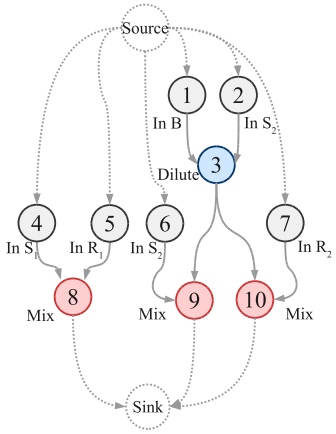
Let us consider that while routing the droplet inside the mixer module in Fig. 2b, it reaches the cell c_2 at time t. The previous movements for the droplet are as shown by the arrows in the same figure. We have five possibilities for $t + 1$: routing the droplet to the left, to the right, up, down or keeping the droplet on c_2. Let us denote with p^0 the percentage of mixing obtained while routing the droplet on an electrode in a forward movement (relative to the previous move), with p^{90} the percentage obtained from a perpendicular movement of the droplet and with p^{180} the percentage of mixing obtained from a backward movement, see Fig. 2b.

Considering Table 1, we can estimate the percentage of mixing over one cell, corresponding to each type of movement (forward, backward, perpendicular). The time required for a droplet to be transported one cell is 0.01 s. In order to approximate p^0, p^{90} and p^{180} we decompose the mixing patterns from the module library in Table 1 in a sequence of forward, backward and perpendicular motions, as shown in Fig. 3. For example, the 2×2 mixer in Fig. 3d can be decomposed in perpendicular movements, because after each move the droplet changes its routing direction by 90°. As shown in Table 1, the operation takes 9.95 s to execute inside the 2×2 module, thus we can safely approximate the percentage of mixing p^{90} to 0.1%.

For the 2×3 module shown in Fig. 3c, the mixing pattern is composed of forward and perpendicular movements. By considering the mixing time shown in Table 1 and $p^{90} = 0.1\%$, we obtain the percentage of mixing resulted from one forward movement $p^0 = 0.29\%$. Note that by decomposing the 2×4 module shown in Fig. 3a, we obtain a different value for p^0: 0.58%. This is because the forward mixing percentage is not constant, but it depends on the number of electrodes used. Therefore we consider that there are two values that estimate the percentage of forward movement: p_1^0, when the forward movement is continued only for one cell as in Fig. 3c, and p_2^0, when the forward movement of the droplet is of at least two cells. This is a safe (pessimistic) approximation, since the value of p^0 will further increase if the droplet continues to move forward.

Considering the percentage of forward movement p_2^0 in the decomposition of the 1×4 module in Fig. 3b, we obtain the (pessimistic) percentage of mixing performed during a

Fig. 4 Application graph



backward motion: $p^{180} = -0.5\%$. The negative mixing is explained by the unfolding of patterns inside the droplet, i.e., the two droplets tend to separate when moved backward.

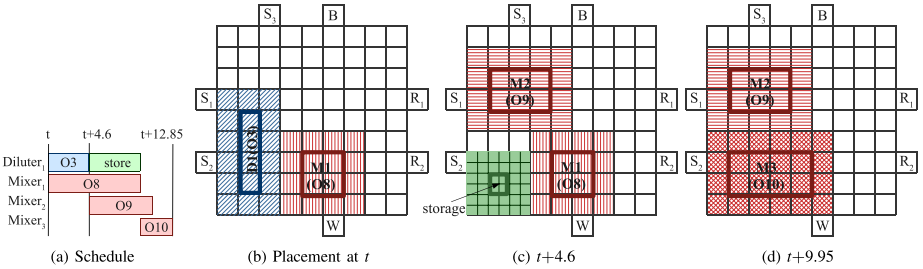
Using these percentages, we can determine the completion time for an operation on a module of any given shape. For example, for the L-shaped module in Fig. 2c, routing the droplet once according to the mixing pattern shown by the arrows leads to 8.72% of mixing. Therefore, in order to complete the mixing operation on the L-shaped module, the droplet will be circularly routed on the showned path 11.46 times, leading to a total time of 2.29 s.

We assume that before synthesis is performed, the set of percentages $\mu = \{p_1^0, p_2^0, p^{90}, p^{180}\}$ is determined through experiments such as the ones in [13] which have produced Table 1. The method presented in this subsection can be applied to any such experimental data, to obtain the completion time of an operation on any module shape.

2.3 Biochemical application model

We model a biochemical application using an abstract model consisting of a sequencing graph [4]. The graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$ is directed, acyclic and polar (i.e., there is a *source node*, which is a node that has no predecessors and a *sink node* that has no successors). Each node $O_i \in \mathcal{V}$ represents one operation. The binding of operations to modules in the architecture is captured by the function $\mathcal{B} : \mathcal{V} \rightarrow \mathcal{A}$, where $\mathcal{A}$ is the set of allocated modules from the given library $\mathcal{L}$.

An edge $e_{i,j} \in \mathcal{E}$ from O_i to O_j indicates that the output of operation O_i is the input of 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. In Fig. 4 we have an example of an application graph with ten operations, O_1 to O_{10}. The application consists of three mixing operations (O_8, O_9 and O_{10}), one diluting operation (O_3) and six input operations (O_1, O_2, O_4, O_5, O_6, O_7). O_3 is a diluting operation that has two outgoing edges, representing an output of two droplets. This requires a split operation. Considering Fig. 1a, a droplet is split by turning on the left and right electrodes and turning off the middle electrode [15]. Thus, the droplet volume will vary during the application execution. We assume that the biochemical application has been correctly designed, such that all the operations will have the required input droplet volumes. Let us consider that the operation O_8 is bound to a 2×3 mixing module denoted by $Mixer_1$ (i.e., $\mathcal{B}(O_8) = Mixer_1$). Then, according to Table 1, the execution time for O_8 on the 2×3 device will be $C_8^{Mixer_1} = 6.1$ s.



(a) Schedule (b) Placement at t (c) t+4.6 (d) t+9.95

Fig. 5 Implementation example

3 Problem formulation

The problem we are addressing in this paper can be formulated as follows. Given (1) a biochemical application modeled as a graph $\mathcal{G}$, (2) a biochip consisting of a two-dimensional $m \times n$ array $\mathcal{C}$ of cells and (3) a characterized module library $\mathcal{L}$, we are interested to synthesize that implementation Ψ, which minimizes the completion time $\delta_{\mathcal{G}}$ (i.e., finishing time of the sink node, t_{sink}^{finish}).

Synthesizing an implementation $\Psi = \langle \mathcal{A}, \mathcal{B}, \mathcal{S}, \mathcal{P} \rangle$ means deciding on: (1) the allocation $\mathcal{A}$, which determines what modules from the library $\mathcal{L}$ should be used, (2) the binding $\mathcal{B}$ of each operation $O_i \in \mathcal{V}$ to one or more modules $M_k \in \mathcal{A}$, (3) the schedule $\mathcal{S}$ of the operations, which contains the start time t_i^{start} of each operation O_i on its corresponding module and (4) the placement $\mathcal{P}$ of the modules on the $m \times n$ array.

The next subsections will illustrate each of these tasks. The presentation order does not correspond to the order in which our synthesis approach performs these tasks.

3.1 Allocation and placement

Let us consider the graph shown in Fig. 4. We would like to implement the operations on the 9×9 biochip from Fig. 1b. We consider the current time as being t. The input operations are already assigned to the corresponding input ports. Thus, O_1 is assigned to the input port B, O_2 to S_2, O_4 to S_1, O_5 to R_1, O_6 to S_2, O_7 to R_2. However, for the mixing operations (O_8, O_9 and O_{10}) and the dilution operation (O_3) our synthesis approach will have to allocate the appropriate modules, bind operations to them and perform the placement and scheduling.

Let us assume that the available module library is the one captured by Table 1. We have to select modules from the library while trying to minimize the application completion time and place them on the 9×9 chip. A solution to the problem is presented in Fig. 5b–d, where the following modules[3] are used: one 2×2 mixer ($Mixer_1$), one 2×3 mixer ($Mixer_2$), one 2×4 mixer ($Mixer_3$) and one 1×4 diluter ($Diluter_1$).

The placement for the solution is as indicated in Fig. 5b–d, where we can notice that modules occupy a space larger than their size, due to the segregation area. If two droplets are next to each other on two adjacent cells, they will tend to merge to form one single droplet. Two approaches have been considered for solving this problem. The first approach, used in [18], consists in having a one-cell distance between any two adjacent modules, which is sufficient for isolating the functional regions on which operations are executing. However, if routing is not considered at the same time with placement, the resulted solution will require

[3] In the figures we denote $Mixer_i$ with M_i and $Diluter_i$ with D_i.

significant modifications for accommodating the necessary routes. As routing needs a 3-cell width channel, one cell between adjacent modules will not be sufficient for creating the necessary routes and thus transporting the droplet. Moreover, the lack of segregation cells between reservoirs and modules placed in their proximity can make the dispensing process impossible. In the second approach, proposed in [3], a segregation area is wrapped around each module. This approach is depicted in Fig. 5b, where $Diluter_1$, which has a size of 1×4 occupies 3×6 cells. As it can be seen, module wrapping provides a 2-cell width channel between any two adjacent modules as well as a 1-cell channel between modules placed at the chip boundary and reservoirs. The advantage of this approach is that the segregation areas can be adjusted during a post-processing step to introduce the necessary paths for droplet movement. In this article, we consider this second approach, and we assume that the routing will be performed in a separate phase, after the positions of the modules have been determined.

Our placement problem has similarities with the placement of DR-FPGAs, where modules can physically overlap on-chip as long as they do not overlap in time, i.e., they are used during different time intervals. After an operation has finished executing on a module, we can reuse the same cells as part of another module. The main difference to DR-FPGAs is that we can easily re-assign operations to devices *during their execution*, as discussed in Sect. 2.1. This property will be used to improve the scheduling, see Sect. 3.3.

3.2 Binding and scheduling

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

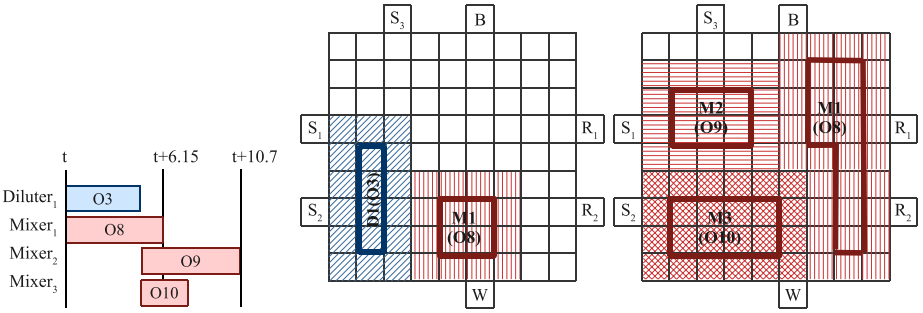
Considering the graph in Fig. 4 with the allocation presented in the previous section, Fig. 5a presents the optimal schedule in the case of static virtual modules, whose placement remains the same throughout their operation. The schedule is depicted as a Gantt chart, where, for each module, we represent the operations as rectangles with their length corresponding to the duration of that operation on the module. For example, operation O_9 is bound to module $Mixer_2$, starts immediately after the dilution operation O_3 (i.e, $t_9^{start} = t + 4.6$) and takes 6.1 s, finishing at time $t_9^{finish} = t + 10.7$ s.

The mixing operation O_{10} cannot start on module $Mixer_3$ until the operation bound to $Mixer_1$ has finished executing, at time $t + 9.95$. Scheduling also decides the access to non-reconfigurable modules, such as input/output ports and detectors, but in this example we have omitted it for simplicity.

Note that special "store" modules have to be allocated if a droplet has to wait before being processed, which is different from DR-FPGAs. Consider the dilution operation O_3, which outputs two droplets corresponding to operations O_9 and O_{10}. As O_{10} is not scheduled immediately after O_3 finishes, a 1×1 storage cell is required to store the droplet until O_{10} can be executed (see Fig. 5c). In general, if there exists an edge $e_{i,j}$ from O_i to O_j such that O_j is not immediately scheduled after O_i (i.e., there is a delay between the finishing time of O_i and the start time of O_j) then we will have to allocate a storage cell for $e_{i,j}$. The allocation of storage cells depends on how the schedule is constructed.

3.3 Synthesis with dynamically reconfigurable modules of any shape

Although the schedule presented in Fig. 5a is optimal for the given allocation and binding, it can be further improved by taking advantage of the property of dynamic reconfiguration of



(a) Schedule (b) Placement at t (c) Dynamic reconfigurability at t+4.6

Fig. 6 Motivational example

the digital biochip. Consider the placement in Fig. 5c. Even though the number of free cells on the microfluidic array at time $t + 4.6$ is higher than the number of cells in $Mixer_3$, the fragmentation of the space makes the placement of $Mixer_3$ impossible. Hence, the operation has to wait until $t + 9.95$, in Fig. 5d, when $Mixer_1$ finishes executing, and there are enough free adjacent cells for accommodating $Mixer_3$.

However, this delay can be avoided by changing the location and the shape of the module $Mixer_1$ on which the mixing operation is performed such that the space fragmentation is minimized. For example, by re-assigning the operation to the "L-shaped" device shown in Fig. 6c and moving the droplet to the new location, we can place $Mixer_3$ at time $t + 4.6$, obtaining the schedule in Fig. 6a. Shifting is done by changing the activation sequence of the electrodes, such that the droplet is routed to the new position, where it continues moving according to the mixing pattern. Considering that at time $t + 4.6$ the mixing operation still had 5.35 s to execute on the 2×2 module out of the total 9.95 s, the rest 53.76% of mixing will be executed on the "L-shaped" mixer. Using the method proposed in Sect. 2.2, the completion time of an operation on the "L-shaped" module is 2.89 s, thus the mixing will complete at time $t + 6.15$.

The overhead that needs to be considered for moving the module is equal to the routing time to the new destination, which, under the assumptions in Sect. 2.1 is 2×0.01 s. In order to constrain the amount of additional routing caused by dynamic routing, our placement approach considers that the routing overhead performed in order to accommodate one device should not exceed a given threshold, $Overhead_{max}$.

4 Tabu Search based synthesis

We have shown that Tabu Search can be used successfully to synthesize good quality solutions in the context of DMBs [11]. In this paper we extend the approach from [11] to consider dynamically reconfigurable non-rectangular devices. Our synthesis strategy, presented in Fig. 7, takes as input the application graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$, the given biochip cell array $\mathcal{C}$, the module library $\mathcal{L}$, and produces that implementation $\Psi = \langle \mathcal{A}, \mathcal{B}, \mathcal{S}, \mathcal{P} \rangle$ consisting of, respectively, the allocation, binding, scheduling and placement, which minimizes the schedule length $\delta_{\mathcal{G}}$ on the given biochip $\mathcal{C}$. As the result of the synthesis process depends on the order of executing the operations, we use priorities Π to decide the scheduling sequence for two or more operations that are ready to be executed at the same time t. In this approach, we use a Tabu Search metaheuristic [9] to decide the allocation $\mathcal{A}$, binding $\mathcal{B}$ and priorities

DMBSynthesis($\mathcal{G}$, $\mathcal{C}$, $\mathcal{L}$)

1 $< \mathcal{A}^{\circ}, \mathcal{B}^{\circ} >=$ InitialSolution($\mathcal{G}, \mathcal{L}$)
2 $\Pi^{\circ} =$ CriticalPath($\mathcal{G}, \mathcal{A}^{\circ}, \mathcal{B}^{\circ}$)
3 $< \mathcal{A}, \mathcal{B}, \Pi >=$ TabuSearch($\mathcal{G}, \mathcal{C}, \mathcal{L}, \mathcal{A}^{\circ}, \mathcal{B}^{\circ}, \Pi^{\circ}$)
4 $< \mathcal{S}, \mathcal{P} >=$ ScheduleAndPlace($\mathcal{G}, \mathcal{C}, \mathcal{A}, \mathcal{B}, \Pi$)
5 **return** $\Psi =< \mathcal{A}, \mathcal{B}, \mathcal{S}, \mathcal{P} >$

Fig. 7 Synthesis algorithm for DMBs

of operations Π (line 3 in Fig. 7). TS starts from an initial solution, where we consider that for the initial allocation $\mathcal{A}^{\circ}$ each operation $O_i \in \mathcal{V}$ is bound to a randomly chosen module $M_i \in \mathcal{L}$ (line 1 in Fig. 7). The initial execution priorities, Π°, are given according to the critical path priority function (line 2 in Fig. 7) [12]. According to this, the priority of an operation is defined as the longest possible schedule length from the execution of the operation to the completion of all the operations in the graph. The next two sections present our proposed scheduling and placement algorithms, respectively, and Sect. 4.3 presents our TS implementation.

4.1 Scheduling heuristic

For given allocation, binding and priorities decided by TS, we use the *ScheduleAndPlace* function presented in Fig. 8 to decide the schedule $\mathcal{S}$ of the operations and the placement $\mathcal{P}$. This section presents the scheduling, and the next section presents the placement algorithm. Our scheduling is based on a List Scheduling (LS) [12] heuristic. LS takes as input the application graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$, the cell array $\mathcal{C}$, the allocation $\mathcal{A}$, binding $\mathcal{B}$ and priorities Π and returns the scheduling $\mathcal{S}$ and the placement $\mathcal{P}$. The List Scheduling heuristic is based on a sorted priority list, L_{ready}, containing the operations $O_i \in \mathcal{V}$ which are ready to be scheduled. The start and finish times of all the operations are initialized to 0 in the beginning of the algorithm (lines 2 and 3 in Fig. 8). A list $L_{execute}$ which contains the operations that are executing at the current time step is created in the beginning of the algorithm (line 4). Initially, L_{ready} will contain those operations in the graph that do not have any predecessors (line 5 in Fig. 8). We do not consider input operations as part of the ready list. As they do not have any precedence constraints, input operations can be executed at any time. However, our algorithm schedules inputs and their successors sequentially, in order to avoid storing the dispensed droplets. Let us consider time $t_{current}$ during the execution of the application. For each operation that finishes executing at $t_{current}$ (line 9) we update the microfluidic array, by removing the device to which the operation was bound to (line 11). The successors of the operation that are ready to be scheduled are added to L_{ready} (line 14 in Fig. 8).

Next, we try and schedule the ready operations, starting with the operation O_j having the highest priority (line 17 in Fig. 8). If the module $\mathcal{B}(O_j)$ to which the operation is bound can be placed on the microfluidic array the placement is updated (line 18 in Fig. 8). If there exists a placed storage module associated with the operation O_j, the storage is removed from the array.

The *ScheduleAndPlace* function (Fig. 8), calls the *DynamicPlacement* function from Fig. 10. Once the placement is known, LS takes into account the routing time, in terms of Manhattan distance, between M_j and the source modules. Once an operation is scheduled it is removed from L_{ready} (line 23 in Fig. 8) and added to $L_{execute}$ (line 24 in Fig. 8). Before the end of the iteration, the storage constraints are considered. If the successors of the operations that finished at $t_{current}$ have not yet been scheduled for execution, a storage unit

ScheduleAndPlace($\mathcal{G}$, $\mathcal{C}$, $\mathcal{A}$, $\mathcal{B}$, Π)

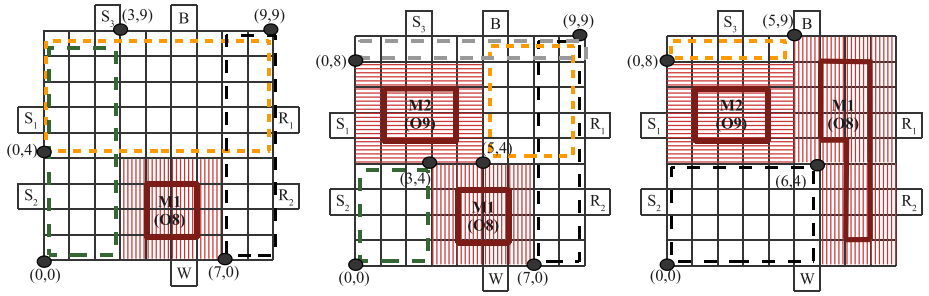
1 $t_{current} = 0$
2 $t_i^{start} = 0, \forall O_i \in \mathcal{G}$
3 $t_i^{finish} = 0, \forall O_i \in \mathcal{G}$
4 $L_{execute} = \emptyset$
5 $L_{reWady} = \text{ConstructReadyList}(\mathcal{G}, \Pi)$
6 *// schedule and place operations*
7 **while** $\exists\ O_i \in \mathcal{G}$ such that $t_i^{finish} = 0$ **do**
8 *// for finishing operations*
9 **for all** $O_j \in L_{execute}$ such that $t_j^{finish} = t_{currenWt}$ **do**
10 *// update placement*
11 UpdatePlacement($\mathcal{C}$, $\mathcal{P}$, $\mathcal{B}(O_j)$)
12 RemoveFromExecuteList(O_j, $L_{execute}$)
13 *// add ready successors to* L_{ready}
14 AddReadySuccessorToList(O_j, L_{ready})
15 **end for**
16 *// schedule ready operations*
17 **for all** $O_j \in L_{ready}$ **do**
18 placed = DynamicPlacement($\mathcal{C}$, $\mathcal{P}$, $\mathcal{B}(O_j)$)
19 **if** placed **then**
20 *// set the start and finish times*
21 $t_j^{start} = t_{current}$
22 $t_j^{finish} = t_j^{start} + C_j^{\mathcal{B}(O_j)}$
23 RemoveFromReadyList(O_j, L_{ready})
24 AddOperationToExecuteList(O_j, $L_{execute}$)
25 **end if**
26 **end for**
27 $t_{current} = t_{current} + 1$
28 **end while**
29 **return** $< \mathcal{S}, \mathcal{P} >$

Fig. 8 List scheduling algorithm for DMBs

is placed on the microfluidic array. TS uses design transformations to search the solution space. Inside TS, we use the *ScheduleAndPlace* function to determine the schedule length $\delta_{\mathcal{G}}$ of each solution.

4.2 Dynamic placement algorithm

We have extended the online placement algorithm from Bazargan et al. [2] for DR-FPGAs to handle DMBs, where we allow dynamic reconfiguration of modules during their execution. Although the algorithm was proposed for online placement, we can use it offline, since we know beforehand all the operations that have to be executed. The algorithm from [2] has three parts: (i) a free space manager which divides the free space on the biochip into a list of overlapping rectangles, L_{rect}, (ii) a search engine which selects an empty rectangle from L_{rect} that best accommodates the module M_i to be placed, according to a given criteria, such as "best fit" and (iii) a placer that inserts M_i on the microfluidic array. Each rectangle can be represented by the coordinates of its left bottom and right upper corners, (x_l, y_l, x_r, y_r). In



(a) Initial placement at $t + 4.6$ (b) Placing $Mixer_2$ (c) Dynamic reconfigurability

Fig. 9 Dynamic placement example

order to allow the placement on the array of modules of any shape we consider in our placement approach that for non-rectangular devices the search engine can select a set R_{chosen} of overlapping free rectangles in which the module can be placed. For rectangular shaped devices R_{chosen} will consist of only one rectangle.

The placement algorithm takes as input the $m \times n$ matrix $\mathcal{C}$ of cells, the current placement of modules $\mathcal{P}$ and the module M_i to be placed, updates the array and returns a boolean value stating if the accommodation of M_i on the array was successful or not. If the module was not placed, LS will have to delay the operation corresponding to M_i.

Let us illustrate the placement algorithm by using Fig. 5c. The ready list consists of the operations in the graph that are ready to be scheduled, hence $L_{ready} = \{O_9, O_{10}\}$. Considering the same allocation and binding presented in Sect. 3.1, the initial priorities for the operations are: $\Pi^{\circ}_{O_9} = C^{Mixer_2}_{O_9} = 6.1$ s and $\Pi^{\circ}_{O_{10}} = C^{Mixer_3}_{O_{10}} = 4.6$ s.

The LS algorithm will select O_9 to be scheduled first and will call *DynamicPlacement* to place $Mixer_2$ on the biochip array. The module $Mixer_1$, which is currently executing at time $t + 4.6$, divides the free space into three overlapping rectangles $L_{rect} = \{Rect_1 = (0, 0, 3, 9), Rect_2 = (0, 4, 9, 9), Rect_3 = (7, 0, 9, 9)\}$, see Fig. 9a (line 2 in Fig. 10). As $Mixer_2$ is a module of rectangular shape, the placement algorithm searches for the smallest rectangle in L_{rect} that fits the 2×3 virtual device. As $Rect_2 = (0, 4, 9, 9)$ is the only rectangle sufficiently large to accommodate the module, $R_{chosen} = \{Rect_2\}$ and $Mixer_2$ will be placed at its bottom corner (line 6 in Fig. 10). Consequently the free space will be updated (line 7) to $L_{rect} = \{Rect_1 = (0, 0, 3, 4), Rect_2 = (5, 4, 9, 9), Rect_3 = (7, 0, 9, 9), Rect_4 = (0, 8, 9, 9)\}$ as depicted in Fig. 9b.

After the scheduling and placement of O_9, the next operation to be considered for scheduling at time $t + 4.6$ is O_{10}. Because of space fragmentation, no free rectangle can accommodate the 2×4 mixer currently assigned to O_{10} and the operation would have to be delayed until $t + 9.95$, as depicted in Fig. 5d, where the mixer is denoted with M_3. However, when no suitable rectangle can be found for accommodating a device, our algorithm, as opposed to [2], will try to decrease the space fragmentation on the microfluidic array by moving and, if necessary, re-assigning operations to modules (possibly of different shape) during their operation.

We use a greedy approach to decide on which modules to move (lines 12–23), until there is space for the current module M_i or a termination criteria is reached. As moving a device requires routing the droplet from the initial position to another one on the array, we place a constraint on the increase in routing time due to moving devices, of one time step, i.e., $Overhead_{max}$ is one second. Therefore, after each move, the variable *RoutingOverhead*,

DynamicPlacement($\mathcal{C}$, $\mathcal{P}$, M_i)

```
1  // construct list of empty rectangles
2  L_rect = ConstructRectList(C)
3  // search for R_chosen that best fits M_i
4  R_chosen = SelectRectangles(L_rect, M_i)
5  if R_chosen ≠ ∅ then
6     placed = UpdatePlacement(P, R_chosen, M_i)
7     UpdateFreeSpace(L_rect)
8  else
9     RoutingOverhead = 0
10    MovesList = ∅
11    // dynamically reconfigure already placed modules
12    while R_chosen = ∅ ∧ RoutingOverhead ≤ Overhead_max do
13       R_chosen = EvaluatePossibleMoves(C, P, L_rect, M_i)
14       if R_chosen ≠ ∅ then
15          placed = UpdatePlacement(P, R_chosen, M_i)
16          UpdateFreeSpace(L_rect)
17       else
18          BestMove = SelectBestMove(C, P, L_rect)
19          PerformMove(BestMove, P, L_rect)
20          RecordMove(MovesList, BestMove)
21          RoutingOverhead = RoutingOverhead + DeterminePerformedRouting(BestMove)
22       end if
23    end while
24    // no placement has been found, restore the original P
25    if R_chosen = ∅ then
26       UndoMoves(P, L_rect, MovesList)
27    end if
28 end if
29 return placed
```

Fig. 10 Placement algorithm for DMBs

capturing the extra routing required for moving the droplet between the two locations is updated (line 21). For example, for a routing time of 0.01 s across one cell, we move modules to accommodate the current module M_i such that routing would not increase with more than 100 cells. The routing distance is calculated based on the Manhattan distance between the left top corners of the old position and the new position of the module considered for moving. In order to have an accurate approximation of the routing overhead, we consider that a module can be moved only if there are no other modules blocking the path between the two locations.

Considering the placement in Fig. 9b, $Mixer_1$ can be moved at most three cells to the left and two to the right while $Mixer_2$ can be moved at most four cells to the right and one up. In order to choose the best move we evaluate all moves that can be performed in a greedy fashion: (i) we check if the new placement obtained after performing one move while maintaining the initial binding can accommodate $Mixer_3$; (ii) if not, we characterize the free space existent on the microfluidic array after the move, considered as a device, and change the shape of the moved device to the new created one; (iii) if no space could be created for accommodating $Mixer_3$ we perform the best move possible, the one minimizing the fragmentation of the space. The moving and, if necessary, re-assigning of operations to

modules continues until the routing constraint is violated (line 12 in Fig. 10). If not enough adjacent cells have been obtained for placing M_i, we restore the initial placement (line 26).

In order to be able to accommodate on the microfluidic array modules of any possible shape, we allow the search engine to group a set of overlapping free rectangles in the case of non-rectangular devices. For example, while evaluating the moves that can be performed on $Mixer_1$ in Fig. 9b (line 13), the algorithm moves $Mixer_1$ two cells to the right. As the move is not sufficient for accommodating $Mixer_3$, we change the module on which $Mixer_1$ is executing. By grouping the free space in the overlapping rectangles $Rect_2 = (5, 4, 9, 9)$ and $Rect_3 = (7, 0, 9, 9)$ we create a new "L-shaped" device, on which $Mixer_1$ can be executed. We assume that the completion time for non-rectangular modules, such as the "L-shape", are computed during the synthesis process, as shown in Sect. 2.2. Once characterized, the devices are added to the given module library for later use. After the re-assignment of $Mixer_1$ to the "L-shape", the free space consists of two rectangles, $Rect_1$ = {0,0,6,4} and $Rect_2$ = {0,8,5,9}. As there are now enough adjacent cells in $Rect_1$, $Mixer_3$ will be placed on the microfluidic array and the placement algorithm will terminate.

4.3 Tabu Search

Tabu Search (TS) is a metaheuristic based on a neighborhood search technique which uses design transformations (moves) applied to the current solution, $\Psi^{current}$, to generate a set of neighboring solutions, N, that can be further explored by the algorithm. Our TS implementation performs two types of transformations: (i) re-binding moves and (ii) priority swapping moves. A re-binding move consists in the re-binding of a randomly chosen operation, O_i, currently executing on module M_i, to another module M_j. Such a move will take care of the allocation, e.g., removing M_i and allocating M_j. A priority swapping move consists in swapping the priorities of two randomly chosen operations in the graph.

In order to efficiently perform the search, TS uses memory structures, maintaining a history of the recent visited solutions (a "tabu" list). By labeling the entries in the list as tabu (i.e., forbidden), the algorithm limits the possibility of duplicating a previous neighborhood upon revisiting a solution. We use two tabu lists, one for each type of move. These are constructed as attribute-based memory structures, containing not the complete recent solutions, but only relevant modified attributes. Hence, if an operation O_i is re-bound to a module M_j as result of a re-binding move, the change of the solution will be recorded in the corresponding tabu list as a pair of the form (O_i, M_j) and if the priorities of two operations O_i and O_j are swapped as part of the diversification process, the move will be recorded as (O_i, O_j).

However, in order not to prohibit attractive moves, an "aspiration criteria" may be used, allowing tabu moves that result in solutions better than the currently best known one. Moreover, in order to avoid getting stuck in a local optima, TS uses "diversification". This involves incorporating new elements that were not previously included in the solution, in order to diversify the search space and force the algorithm to look in unexplored areas. Based on experiments, we have decided to use priority swapping as a diversification move, only when the best known solution does not improve for a defined number of iterations, num_{div}, determined experimentally.

The TS algorithm described in Fig. 11 takes as input the application graph $\mathcal{G}$, the microfluidic array $\mathcal{C}$, the module library $\mathcal{L}$, the initial allocation $\mathcal{A}^\circ$, binding $\mathcal{B}^\circ$ and priorities Π°, and returns the best found implementation, Ψ^{best}. Initially the best solution is considered the initial one, in which each operation is assigned to a randomly chosen device in the module library and has the priority given according to the critical path length. The evaluation of the initial solution Ψ° is performed by the *ScheduleAndPlace* method, which returns

TabuSearch($\mathcal{G}$, $\mathcal{C}$, $\mathcal{L}$, $\mathcal{A}^\circ$, $\mathcal{B}^\circ$, Π°)

1 $< \mathcal{S}^\circ, \mathcal{P}^\circ >$ = ScheduleAndPlace($\mathcal{G}, \mathcal{C}, \mathcal{A}^\circ, \mathcal{B}^\circ, \Pi^\circ$)
2 $\Psi^{best} = \Psi^{current} = \Psi^\circ =< \mathcal{A}^\circ, \mathcal{B}^\circ, \mathcal{S}^\circ, \mathcal{P}^\circ >$
3 $\delta_{\mathcal{G}}^{best} = \delta_{\mathcal{G}}^{current} = \delta_{\mathcal{G}}^\circ$ = GetCompletionTime($\mathcal{S}^\circ$)
4 $TabuList_{dev} = \emptyset$
5 $TabuList_{prio} = \emptyset$
6 $num_{iter} = 0$
7 **while** *timeLimit* not reached **do**
8 N = GenerateNeighborhood($\Psi^{current}, \mathcal{L}$)
9 $\tilde{N}$ = SelectAllowedMoves(N)
10 $(O_i, \mathcal{B}(O_i))$ = SelectBestMove($\tilde{N}$)
11 PerformBestMove($\Psi^{current}$, O_i, $\mathcal{B}(O_i)$)
12 RecordRebindMove(O_i, $\mathcal{B}(O_i)$, $TabuList_{dev}$)
13 $\delta_{\mathcal{G}}^{current}$ = GetCompletionTime($\mathcal{S}^{current}$)
14 **if** $\delta_{\mathcal{G}}^{current} < \delta_{\mathcal{G}}^{best}$ **then**
15 $\Psi^{best} = \Psi^{current}$; $\delta_{\mathcal{G}}^{best} = \delta_{\mathcal{G}}^{current}$
16 **else**
17 $num_{iter} = num_{iter} + 1$
18 **if** $num_{iter} = num_{div}$ **then**
19 (O_i, O_j) = SelectSwapMove($\mathcal{G}$, $\Pi^{current}$, $TabuList_{prio}$)
20 PerformSwapMove($\Psi^{current}$, O_i, O_j)
21 RecordSwapMove(O_i, O_j, $TabuList_{prio}$)
22 $\delta_{\mathcal{G}}^{current}$ = GetCompletionTime($\mathcal{S}^{current}$)
23 **if** $\delta_{\mathcal{G}}^{current} < \delta_{\mathcal{G}}^{best}$ **then**
24 $\Psi^{best} = \Psi^{current}$; $\delta_{\mathcal{G}}^{best} = \delta_{\mathcal{G}}^{current}$
25 **end if**
26 $num_{iter} = 0$
27 **end if**
28 **end if**
29 **end while**
30 **return** Ψ^{best}

Fig. 11 Tabu Search algorithm for DMBs

the schedule length $\delta_{\mathcal{G}}^\circ$ obtained for the given allocation, binding and priorities (line 3 in Fig. 11). Two tabu lists, $TabuList_{dev}$ and $TabuList_{prio}$ are used for recording the re-binding moves, respectively the priority swapping moves. Each list has a given size, $tabuSize_{dev}$ and $tabuSize_{prio}$ correspondingly, specifying the maximum number of moves that can be recorded by the list. Initially, the lists are empty (lines 4–5 in Fig. 11). A variable num_{iter} is used to keep track of the number of iterations passed without the improvement of the best solution, $\Psi_{\mathcal{G}}^{best}$ (line 6).

The algorithm is based on a number of iterations (lines 7–29 in Fig. 11) during which moves are applied to the current solution in order to try and improve the overall best solution $\Psi_{\mathcal{G}}^{best}$. In each iteration, a set of possible candidates N is obtained by applying moves to the current solution, $\Psi^{current}$ (line 8). However, N might contain solutions that are disallowed by TS. According to the aspiration criteria, a tabu move $(O_i, M_j) \in TabuList_{dev}$ is only allowed if it leads to a solution better than the currently best known one. Therefore, all the tabu moves resulting in solutions with schedule lengths $\delta_{\mathcal{G}}^{current}$ worse than the currently best one are removed from N and thus, the set $\tilde{N}$ of allowed moves is created (line 9).

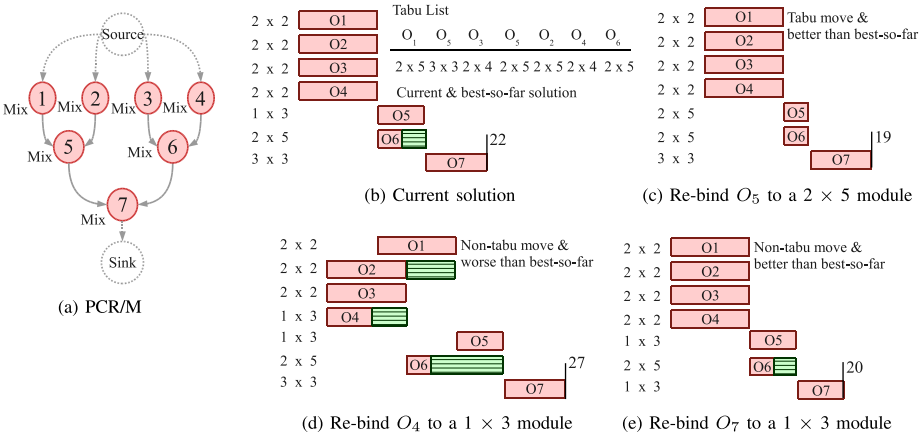


Fig. 12 Tabu Search neighborhood

Operation	Area (cells)	Time (s)
Mixing	2 × 5	2
Mixing	2 × 4	3
Mixing	1 × 3	5
Mixing	3 × 3	7
Mixing	2 × 2	10
Dilution	2 × 5	4
Dilution	2 × 4	5
Dilution	1 × 3	7
Dilution	3 × 3	10
Dilution	2 × 2	12
Dispensing	–	7
Detection	1 × 1	30

(a) Module library for the experimental evaluation

Operation	Label	Area (cells)	Time (s)
Mixing	L_1	4 × 2 × 1	1.92
Mixing	L_2	5 × 2 × 1	1.78
Mixing	T	4 × 3 × 1	2.14
Mixing		1 × 5	1.60
Mixing		1 × 6	1.53
Dilution	L_1	4 × 2 × 1	3.78
Dilution	L_2	5 × 2 × 1	3.57
Dilution	T	4 × 3 × 1	4.10
Dilution		1 × 5	3.22
Dilution		1 × 6	3.12

(b) Library of characterized modules

Fig. 13 Experimental evaluation

The *ScheduleAndPlace* function is used for determining the move $(O_i, M_j) \in \tilde{N}$ leading to the solution with the shortest schedule length $\delta_{\mathcal{G}}^{current}$ among all the moves in $\tilde{N}$. The move is selected and marked as tabu (lines 10–12). If the obtained solution has a better schedule length that the currently known one, the best-so-far solution is updated (lines 14–15). When the best known solution does not improve for a given number of iterations num_{div} a diversification move is considered (line 18), forcing the search into unexplored areas of the search space. The move consists in swapping the priorities of two randomly selected operations O_i and O_j, with $(O_i, O_j) \notin TabuList_{prio}$. If the move results in a new best known solution, Ψ^{best} is updated to $\Psi^{current}$ (line 23). Finally, the variable num_{iter} is reset to 0 (line 26).

Let us use the mixing stage of the polymerase chain reaction (PCR/M) shown in Fig. 12a, and the module library in Fig. 13a to illustrate how each iteration is performed. Consider the current solution as being the one represented by the schedule in Fig. 12b. The current tabu list, presented to the right, contains the recently performed transformations. As all operations are mixing operations, we will denote a module by its area, e.g. O_1 is bound to a mixing module of 2 × 2 cells. Starting from this solution, TS uses re-binding moves to generate

the neighbor solutions (line 8). Out of the possible neighboring solutions we present three in Fig. 12c–d. The solution in Fig. 12c is tabu and the one in Fig. 12d is worse than the current solution (which is the best so far). In the solution in Fig. 12e O_7 is re-bound to a new, 1×3 mixer, which results in a non-tabu solution better than the current one. However, TS will select the move in Fig. 12c, that would change the 1×3 mixer in Fig. 12b for O_5 to a 2×5 mixer module (line 10). Although the move $(O_5, 2 \times 5)$ is marked as being tabu, it leads to a better result than the currently best known one and thus, according to the aspiration criteria, it is accepted (lines 14–15). The evaluation of the new solution is done by using the unified scheduling and placement algorithm presented before which determines the completion time $\delta_{\mathcal{G}}$ of the application graph. If the best solution Ψ^{best} has not been improved for a number of num_{div} iterations, the algorithm diversifies the search space by performing a priority swapping move, which is recorded into $tabuList_{prio}$ (lines 18–28). The algorithm terminates when a given time-limit for performing the search has been reached.

5 Experimental evaluation

In order to evaluate our proposed approach, we have used a real life example and ten synthetic benchmarks. The Tabu Search-based algorithm[4] was implemented in Java (JDK 1.6), running on SunFire v440 computers with UltraSPARC IIIi CPUs at 1,062 GHz and 8 GB of RAM. The module library used for all the experiments is shown in Fig. 13a. For simplicity, we have considered in our implementation that the characterization of new modules is done offline. For example, Fig. 13b contains a set of devices of different shapes, characterized from the given module library in Fig. 13a. The non-rectangular devices (having "L" and "T" shapes) are described by the lengths of the two segments and the thickness. During the synthesis process, the operations can be re-bound to one of the other devices in Fig. 13a or to a new device characterized in Fig. 13b.

In our first set of experiments we measured the quality of the TS implementation, that is, how consistently it produces good quality solutions. Hence, we used our TS-based approach for synthesizing a large real-life application implementing a colorimetric protein assay (103 operations), utilized for measuring the concentration of a protein in a solution.

Table 2 presents the results obtained by synthesizing the protein application on three progressively smaller microfluidic arrays. We present the best solution (in terms of schedule length), the average and the standard deviation obtained after 50 runs of the TS algorithm. Let us first concentrate on the results obtained for the case when we have used a time limit of 60 minutes for the TS. As we can see, the standard deviation is quite small, which indicates that TS consistently finds solutions which are very close to the best solution found over the 50 runs, which will explore differently the solution space, resulting thus in different solutions.

Moreover, the quality of the solutions does not degrade significantly if we reduce the time limit from 60 minutes to 10 minutes and 1 minute. This is important, since we envision using TS for architecture exploration, where several biochip architectures have to be quickly evaluated in the early design phases (considering not only different areas, but also different placement of non-reconfigurable resources such as reservoirs or detectors).

For the second set of experiments we were interested in the gains that can be obtained by allowing the dynamic reconfiguration of the devices during their execution. Hence, we

[4] Values for TS parameters determined experimentally: $num_{div} = 7$, $tabuSize_{dev} = 8$, $tabuSize_{prio} = 8$.

Table 2 Results for the colorimetric protein assay

Area	Time limit	Best (s)		Average (s)		Standard dev.	
		TS	TS^-	TS	TS^-	TS	TS^-
13 × 13	60 min	178.49	182	182.03	189.88	2.53	2.90
	10 min	178.49	182	188.42	192.00	4.53	3.64
	1 min	187.49	191	194.09	199.20	4.07	4.70
12 × 12	60 min	178.49	182	183.38	190.86	3.09	3.20
	10 min	178.49	185	189.99	197.73	4.41	6.50
	1 min	190.50	193	195.13	212.62	8.97	10.97
11 × 12	60 min	178.49	184	189.18	192.50	5.50	3.78
	10 min	178.49	194	193.85	211.72	4.90	14.37
	1 min	191.50	226	225.13	252.19	9.27	15.76

have modified our TS approach to eliminate the possibility of moving devices and changing their shape during their execution from our TS approach. Table 2 presents the comparison between this modified TS approach, denoted by TS^-, and TS for the protein application. As we can see, taking into account the dynamic reconfigurability property of the biochip, significant improvements can be gained in terms of schedule length, allowing us to use smaller areas and thus reduce costs. For example, in the most constrained case, a 11 × 12 array, we have obtained an improvement of 10.73% in the average completion time compared with TS^-, for the same limit of time, 1 minute.

In a third set of experiments we have evaluated our proposed method on ten synthetic applications. Due to the lack of biochemical application benchmarks, we have generated a set of synthetic graphs using Task Graphs For Free (TGFF) [7]. We have manually modified the graphs in order to capture the characteristics of biochemical applications. The applications are composed of 10 up to 100 operations and the results in Table 3 show the best and the average completion time obtained out of 50 runs of TS and TS^- using a time limit of 10 minutes.

For each synthetic application we have considered three areas, from $Area_1$ (largest) to $Area_3$ (smallest). The results in Table 3 confirm the conclusion from Table 2: as the area decreases, considering dynamic reconfiguration becomes more important, and leads to significant improvements. For example, for the synthetic application with 50 operations, in the most constrained case, a 9 × 9 array, we have obtained an improvement of 24.52% in the average completion time compared with TS^-.

6 Conclusions

In this paper we have presented a Tabu Search based-technique for the synthesis of digital microfluidic biochips. The proposed approach considers the unified architectural design (allocation, binding and scheduling) and physical design (placement of operations on a microfluidic array). In this work, we have considered that the binding of operations to virtual devices can be changed during their execution. We have also proposed a method for analytically determining the completion time of a reconfigurable operation on a device of any given shape. A real-life example as well as a set of ten synthetic applications have been used for evaluating the effectiveness of the proposed TS approach. We have shown that by exploiting

Table 3 Results for synthetic benchmarks

Nodes	$Area_1$	$Average_1$ (s)		$Best_1$ (s)		$Area_2$	$Average_2$ (s)		$Best_2$ (s)		$Area_3$	$Average_3$ (s)		$Best_3$ (s)	
		TS	TS^-	TS	TS^-		TS	TS^-	TS	TS^-		TS	TS^-	TS	TS^-
10	9×7	20.23	24.00	15.10	20	7×8	21.39	25.19	16.19	20	8×6	28.60	32.58	24.20	27
20	8×7	54.37	55.16	51.82	55	7×7	55.72	58.61	54.12	58	6×7	63.86	67.33	60.48	67
30	10×11	46.47	56.00	37.20	41	10×10	53.93	60.78	37.30	41	9×11	56.10	66.52	44.49	54
40	10×11	53.55	68.58	49.49	58	10×10	58.81	76.50	53.59	58	9×10	66.25	86.37	54.59	67
50	10×10	101.54	117.78	97.89	104	8×11	107.85	132.44	98.97	112	9×9	108.35	143.56	99.69	119
60	11×12	111.56	113.50	106.69	110	10×11	111.84	115.40	106.69	112	9×10	117.77	125.58	112.09	118
70	12×12	123.01	131.87	119.99	121	11×12	125.09	137.66	120.09	123	10×11	143.43	159.72	123.39	136
80	12×12	146.80	161.60	144.39	154	11×11	176.23	192.86	153.12	165	10×11	187.72	210.90	155.79	168
90	15×15	127.72	128.02	114.79	120	14×14	129.67	135.68	120.01	127	13×13	149.76	164.20	137.29	142
100	15×15	165.29	178.36	157.59	163	14×14	165.81	179.90	159.49	170	13×13	166.68	183.84	159.69	175

the dynamic reconfigurability of digital microfluidic biochips significant improvements can be gained, allowing us to use smaller area biochips and thus reduce costs.

References

1. Advanced Liquid Logic (2010) http://www.liquid-logic.com/technology.html
2. Bazargan K, Kastner R, Sarrafzadeh M (2000) Fast template placement for reconfigurable computing systems. IEEE Des Test Comput 17(1):68–83
3. Chakrabarty K, Su F (2006) Digital microfluidic biochips: synthesis, testing, and reconfiguration techniques. CRC Press, Boca Raton
4. Chakrabarty K, Zeng J (2005) Design automation for microfluidics-based biochips. ACM J Emerg Technol Comput Syst 1(3):186–223
5. Chakrabarty K, Zeng J (2006) Design automation methods and tools for microfluidic-based biochips. Springer, Berlin
6. Cho M, Pan DZ (2008) A high-performance droplet router for digital microfluidic biochips. In: Proceedings of international symposium on physical design, pp 200–206
7. Dick RP, Rhodes DL, Wolf W (1998) TGFF: task graphs for free. In: Proceedings of the sixth international workshop on hardware/software codesign, pp 97–101
8. Fair RB (2007) Digital microfluidics: is a true lab-on-a-chip possible? Microfluid Nanofluid 3(3):245–281
9. Glover F, Laguna M (1997) Tabu search. Kluwer Academic, Dordrecht
10. Maftei E, Paul P, Madsen J, Stidsen T (2008) Placement-aware architectural synthesis of digital microfluidic biochips using ILP. In: Proceedings of the international conference on very large scale integration of system on chip, pp 425–430
11. Maftei E, Paul P, Madsen J (2009) Tabu search-based synthesis of dynamically reconfigurable digital microfluidic biochips. In: Proceedings of the compilers, architecture, and synthesis for embedded systems conference, pp 195–203
12. Micheli GD (1994) Synthesis and optimization of digital circuits. McGraw-Hill Science, New York
13. Paik P, Pamula VK, Fair RB (2003) Rapid droplet mixers for digital microfluidic systems. Lab Chip 3:253–259
14. Pollack MG, Shenderov AD, Fair RB (2002) Electrowetting-based actuation of droplets for integrated microfluidics. Lab Chip J 2:96–101
15. Ren H, Srinivasan V, Fair RB (2003) Design and testing of an interpolating mixing architecture for electrowetting-based droplet-on-chip chemical dilution. In: Proceedings of the international conference on transducers, solid-state sensors, actuators and microsystems, pp 619–622
16. Silicon Biosystems (2010) http://www.siliconbiosystems.com
17. Su F, Chakrabarty K (2004) Architectural-level synthesis of digital microfluidics-based biochips. In: Proceedings of international conference on computer aided design, pp 223–228
18. Su F, Chakrabarty K (2005) Unified high-level synthesis and module placement for defect-tolerant microfluidic biochips. In: Proceedings of the 42nd annual conference on design automation, pp 825–830
19. Su F, Chakrabarty K (2006) Module placement for fault-tolerant microfluidics-based biochips. ACM Trans Des Autom Electron Syst 11(3):682–710
20. Su F, Hwang W, Chakrabarty K (2006) Droplet routing in the synthesis of digital microfluidic biochips. In: Proceedings of design, automation and test in Europe, vol 1, pp 73–78
21. Thorsen T, Maerkl S, Quake S (2002) Microfluidic largescale integration. Science 298:580–584
22. Xu T, Chakrabarty K (2007) Integrated droplet routing and defect tolerance in the synthesis of digital microfluidic biochips. In: Proceedings of design automation conference, pp 948–953
23. Yuh P-H, Yang C-L, Chang Y-W (2004) Temporal floorplanning using the T-tree formulation. In: Proceedings of international conference on computer aided design, pp 300–305
24. Yuh P-H, Yang C-L, Chang Y-W (2006) Placement of digital microfluidic biochips using the T-tree formulation. In: Proceedings of design automation conference, pp 931–934
25. Yuh P-H, Yang C-L, Chang Y-W (2007) Placement of defect-tolerant digital microfluidic biochips using the T-tree formulation. ACM J Emerg Technolog Comput Syst 3(3). doi:acm.org/10.1145/1295231.1295234
26. Yuh P-H, Yang C-L, Chang Y-W, Chen H-L (2007) Temporal floorplanning using three dimensional transitive closure subGraph. ACM Trans Des Autom Electron Syst 12(4). doi:acm.org/10.1145/1278349.1278350