Fast Online Synthesis of Digital Microfluidic Biochips

Daniel T. Grissom and Philip Brisk, *Member, IEEE*

***Abstract*—We introduce an online synthesis flow, focusing primarily on the virtual topology and operation binder, for digital microfluidic biochips, which will enable real-time response to errors and control flow. The objective of this flow is to facilitate fast assay synthesis while minimally compromising the quality of results. In particular, we show that a virtual topology, which constrains the allowable locations of assay operations such as mixing, dilution, sensing, etc., in lieu of traditional placement, can significantly speed up the synthesis process without significantly lengthening assay execution time. We present a base virtual topology and show how it can be leveraged to reduce algorithmic runtimes and guarantee rout ability. We later present several variations of the virtual topology and present experimental results demonstrating best-design practices. We present two binding solutions. The first is a left-edge binding algorithm, while the second is a more intelligent path-based binding algorithm that leverages spatial and temporal locality to produce superior results.**

***Index Terms*—Laboratory-on-chip, microfluidics, placement, routing, scheduling, synthesis.**

I. Introduction

WITH THE emergence of novel, scalable, flexible digital micro fluidic biochips (DMFBs) [17], new features such as end-user programmability and online synthesis will revolutionize micro fluidic applications. Instead of application-specific DMFBs, low-cost, general-purpose DMFBs will provide a reusable, flexible, and programmable platform. With the notion of end-user programmability being introduced to DMFBs, control-flow constructs present exciting, new possibilities for microfluidic applications. Consequently, when control-flow is introduced, synthesis (Fig. 1) will need to be performed online since the order of assays (biochemical reactions) to be executed is no longer deterministic, but instead dependent on live-feedback from the DMFB [2]–[15].

In contrast to offline compilers, which synthesize assays as deterministic state-machines, an online interpreter will act more like a virtual machine that manages the DMFBs resources and interprets assays on-the-fly. Fig. 2 shows the tradeoffs that need to be made when moving synthesis online. During offline compilation, optimized designs are created with little concern to algorithmic runtime (time need for synthesis) since the synthesis process is run once and the compiled "executable binary" is packaged into an application-specific device. With a programmable DMFB, the end-user will have to wait each time a programmed assay is synthesized. Furthermore, each time a branch is taken, the user will have to wait as the target assay of the branch is interpreted online. Thus, new synthesis methods are needed that concede optimality in assay length (i.e., schedules) and area to reduce algorithmic runtimes from seconds/minutes to milliseconds and achieve a greater amount of flexibility [15].

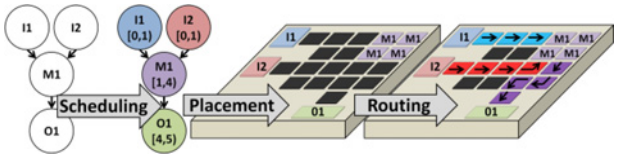

Fig. 1. DMFB synthesis consists of scheduling, placing, and routing (Fig. 3, [11]).

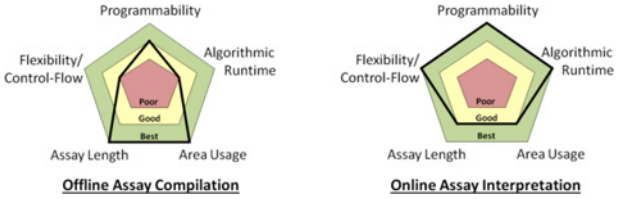

Fig. 2. Offline versus online synthesis tradeoffs.

A. Motivation

We motivate the need for fast, online synthesis methods with an example that is either impossible without this feature, or requires unreasonably complex solutions. Consider a drug-discovery application where a DMFB executes an assay, measures the result and then automatically executes a new assay (or batch of assays) with different concentrations, based on the previous result. This process is repeated thousands of times until a set of concentrations yielding the desired result is discovered.

With offline compilation, a single graph must be created that details every possible execution path, which quickly becomes intractable as the compiler must account for numerous paths

Manuscript received March 15, 2013; revised July 12, 2013 and September 13, 2013; accepted October 13, 2013. Date of current version February 14, 2014. This work was supported in part by the National Science Foundation (NSF) under Grant CNS-1035603 and in part by an NSF Graduate Research Fellowship awarded to Daniel Grissom. This paper was recommended by Associate Editor Y.-W. Chang. A preliminary version of this paper was presented at the 2012 International Conference on Hardware/Software Codesign and System Synthesis.

The authors are with the Department of Computer Science and Engineering, University of California, Riverside, CA 92521 USA (e-mail: grissomd@cs.ucr.edu; philip@cs.ucr.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCAD.2013.2290582

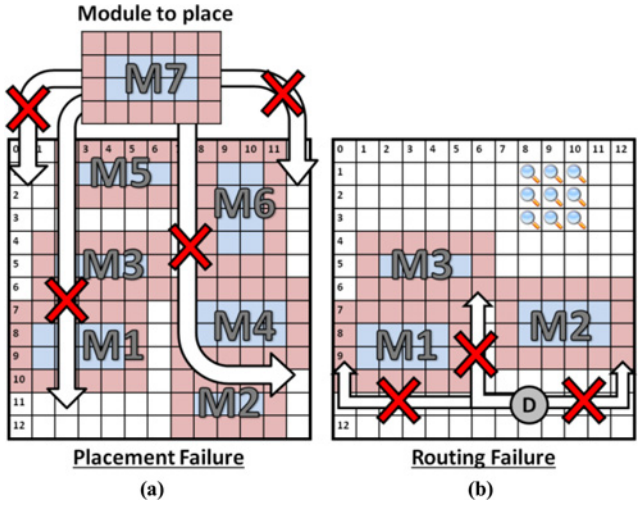

Fig. 3. (a) Placement failure occurs because there is insufficient space for M7 to be placed given the placement of modules M1–M6; (b) Routing failure occurs because the droplet (D) is attempting to reach the detection zone (marked with magnifying glass lenses) but cannot because modules M1–M3 are placed in such a way that block all paths to the destination.

that will never be taken [2]. Instead, upon completion of one assay, an online interpreter could immediately interpret and execute the next assay with only milliseconds of downtime between assays.

Although synthesis has been performed entirely offline up to this point, Ho *et al.* [13] suggest that online systems are forthcoming with the development of specialized heuristics, which can perform synthesis in milliseconds. Luo *et al*. [15] have implemented one such specialized heuristic for an error detection and recovery scheme based on check-pointing: at each checkpoint, a droplet is routed to a sensor that detects whether its concentration is satisfactory; if not, the assay is resynthesized on-the-fly to repeat the sequence of operations that produced the droplet, interleaving the schedule of these newly-introduced operations with concurrent operations that do not depend on the droplet that failed the checkpoint.

B. Problem Formulation

A synthesis tool that converts a micro fluidic assay from a sequencing graph specification to a sequence of electrode activations must solve three NP-Hard problems: scheduling; placement; and routing.

Scheduling: A solution to the scheduling problem determines the time at which each assay operation starts and finishes, including the allocation of temporary storage for intermediate droplets, while ensuring that: 1) the schedule satisfies all precedence constraints in the sequencing graph and 2) the demand for resources at each time-step of the schedule does not exceed the supply of resources in the target DMFB [18], [26].

Placement: At each time-step of the schedule, all of the executing operations and stored droplets must be placed at different locations on the chip while simultaneously ensuring that modules are arranged in such a way as to avoid placement failure [e.g., Fig. 3(a)]. In particular, operations that required specialized external devices, such as heating or detection, must be placed on DMFB locations that are accessible to the appropriate specialized devices [27].

Routing: At different times during assay execution, droplets must be transported to different DMFB locations, e.g., from an input reservoir to where an operation will start, from the location of one operation to another operation, or to a temporary storage location, or to an output reservoir. During routing, droplets in-transit must not inadvertently collide with one another, or collide with other assay operations that are in-progress [29].

In general, the objective of scheduling, placement, and routing is to minimize assay execution time. In addition to these basic requirements, we introduce three new goals: 1) fast algorithmic runtimes; 2) placements that guarantee routability; and 3) deadlock-free routing. Fast algorithmic runtimes are imperative for dynamic synthesis and resynthesis to facilitate control flow and error detection and recovery scenarios in a way that does not cause large delays. Placements must be routable *a priori*, because the computational overhead to detect and rectify unroutable placements [e.g., Fig. 3(b)] is significant. Droplet deadlocks are problematic because no droplet can advance toward its destination, preventing completion of the assay; the computational overhead to detect and rectify deadlock situations that may occur during routing is significant. The usage of the virtual topology seamlessly achieves all three of these objectives by reducing the algorithmic complexity of synthesis and providing the order and constructs necessary to compute routable placements and deadlock-free routes on the first attempt.

C. Contribution

Assays are synthesized by computing solutions for three NP-complete problems, as seen in Fig. 1 Before synthesis, an assay is modeled as a directed acyclic graph (DAG), where the nodes and edges represent operations and operation dependencies, respectively. Each assay operation is then assigned start/stop times during resource-constrained scheduling [21]. During the placement stage, the scheduled operations are assigned specific locations, called modules, on the array [27]. Finally, the routing stage computes droplet paths between subsequent modules and I/O ports so droplets arrive safely at each destination [29].

We present an online synthesis flow that can interpret assays and map them onto a cross-referenced, fully-addressable or active-matrix DMFB [17] in milliseconds, making it appropriate for both offline and online synthesis. Our key contribution is a virtual topology that defines distinct regions for module placement and droplet routing. With our topology in mind, we present several constraints are necessary and apply them to list scheduling [26] and path scheduling [11] to quickly produce schedules. Placement, which has been solved in the past by iterative improvement algorithms [27], [35] or integer linear programming (ILP) [14], is simplified to a binding problem, which can be solved efficiently in polynomial-time. We present two fast binding solutions. The placement defined by the virtual topology provides dedicated routing cells which ease the router's job. We simplify an existing router [22] to compute droplet paths very quickly.

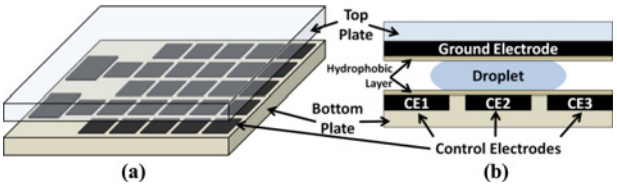

Fig. 4. (a) DMFB with a 2-D array of electrodes. (b) DMFB cross section (Fig. 1, [11]).

D. DMFB Technology Overview

A DMFB manipulates discrete droplets of fluid on a 2-D array of electrodes [Fig. 4(a)] through electrowetting, a process that induces droplet motion [20]. Fig. 4(b) shows a droplet sandwiched between a ground electrode and a set of independent control electrodes. The droplet is centered over one electrode (CE2), but overlaps adjacent electrodes CE1 and CE3. If a voltage is applied to CE1 or CE3, the surface energy gradient induces motion and the droplet moves to the center of the newly activated electrode(s). During each droplet-actuation cycle, a set of electrodes is activated, which moves droplets from electrode to electrode. Basic assay operations such as transport, mixing, merging, and splitting are performed through the appropriate sequence of electrode activations over a number of cycles.

There are several classes of DMFBs that provide varying levels of droplet control. Typical direct-addressing (fully-addressable) DMFBs have one control pin for each electrode (i.e., $(M \times N)$ control pins for an $(M \times N)$ array of electrodes) so each electrode (droplet) can be independently controlled at all times. However, the wiring cost of independently controlled electrodes, especially as array sizes grow, has motivated cheaper designs [32].

Cross-referencing DMFBs use $(M + N)$ control pins to control an $(M \times N)$ array of electrodes [7]. In this scheme, each row and each column has a single control pin; when a particular column m and row n are activated, the electrode at (m, n) is activated. Multiple columns and rows can be simultaneously activated, but may cause superfluous electrode activation, yielding undesired droplet movement [31]. Thus, once a route for a direct-addressing DMFB is computed, each droplet-actuation cycle is serialized across multiple droplet-actuation cycles, resulting in prolonged routing times and increased algorithmic complexity.

Pin-constrained DMFBs represent another addressing scheme. An assay is first synthesized as if on a direct-addressing DMFB. Then, special heuristics attempting to solve the clique partitioning problem (NP-Hard) are used to minimize the total number of control pins, based on which electrodes can be activated together without causing undesired droplet movement [32].

To summarize, pin-constrained designs offer minimal product costs, are inflexible and cannot be reprogrammed after being manufactured; cross-referencing DMFBs are reprogrammable, but add another layer of complexity that must be handled to serialize droplet motion [31].

Fortunately, active-matrix addressing designs are emerging which give independent control of $(M \times N)$ electrodes while using only $(M + N)$ control pins [17]. Active matrix addressing can scale without growing prohibitively expensive, while maintaining the maximum level of flexibility and control so that assays can be programmed with minimal levels of algorithmic complexity. The online synthesis flow presented in this paper is compatible with direct, cross-referencing, and active-matrix addressing DMFBs.

II. Related Work

Here, we highlight some of the previous work in DMFB synthesis for scheduling, placement and routing.

A. Scheduling

Su and Chakrabarty [26] present modified list scheduling (MLS) and genetic algorithm (GA) heuristics, as well as an optimal integer linear programming (ILP) model for scheduling microfluidic operations onto a DMFB. As expected, the ILP implementation consumes a large amount of time to compute optimal solutions. Although the GA finds optimal or near-optimal results in much less time than ILP, its iterative nature still results in large computation times. MLS produces schedules comparable to GA in much less time. Other scheduling algorithms such as Ricketts' hybrid genetic algorithm [21] and Maftei's tabu search scheduler [16] are iterative improvement algorithms which spend anywhere from four seconds to one hour computing schedules. We chose list scheduling as the base scheduler for our framework, but other fast schedulers being developed now [11], [18], or in the future could be used as well.

B. Placement

At the physical level, all electrodes are equally capable of performing the basic microfluidic operations (i.e., merging, mixing, splitting, transport, storage). Hence, basic operations can be performed anywhere on a DMFB array. The objective for most placers is to pack as many concurrent operations/modules into as little area as possible. Several direct-addressing placement and unified scheduling-placement algorithms [27], [28], [33], [35] use simulated annealing, which run in minutes or tens of seconds; in contrast, our online flow completes in tens of milliseconds.

Griffith *et al*. [8] place a virtual topology onto the DMFB, which dictates separate regions for assay operations and droplet routing. However, they only present results for one assay, and their implementation suffers from deadlocks during droplet routing. Our approach is similar, but does not suffer from deadlock; in the absolute worst case, our router will transport one droplet at a time; however, we include a compaction step to transport multiple droplets concurrently.

C. Routing

Böhringer [5] modeled droplet routing as an A* search, similar to path planning in robotics, achieving an optimal-length solution, when routable. Su *et al*. [29] route droplets sequentially and redo placement when routing fails. BioRoute [34] uses a min-cost max-flow algorithm to compute several routes

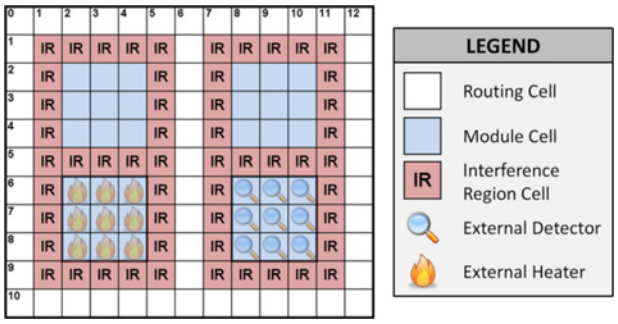

Fig. 5. Virtual topology imposed onto a DMFB.

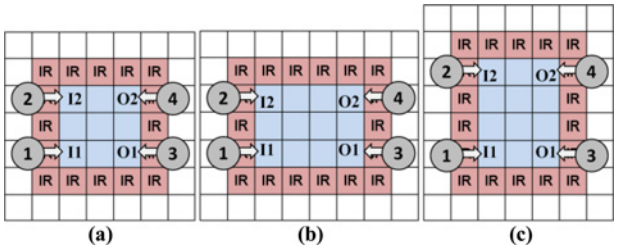

Fig. 6. Entrance cells (I1/I2) and exit cells (O1/O2). (a) 3x3 module. (b) 4x3 module. (c) 3x4 module.

at once, followed by negotiation-based detailed routing. Cho and Pan [6] route droplets one-by-one and sort them based on a by passability metric; if a deadlock occurs, droplets are moved to concession zones to break the deadlock. Huang and Ho [12] construct a system of global routing tracks, which are aligned in the same direction as the majority of droplets traveling on that tract. They use an entropy-based equation to determine the order in which droplets are routed, and finally, compact the routes using dynamic programming. Since the aforementioned methods were designed for offline routing, few mention runtimes [5], [6], [29]. Bio Route [34] and Huang's algorithm [12] both report runtimes below one second on a desktop PC. The router used in our online flow, a modified version of Roy's maze router [22], achieves comparable runtimes, while achieving deadlock freedom.

D. Combined Methods

Most work on synthesis has focused on the scheduling, placement or routing problems in isolation. Several papers, however, solve some of these problems together, using iterative improvement heuristics [15], [27], [28], [33], [35], whose runtime is prohibitive. These approaches address problems that can arise when one stage of synthesis does not consider the next. For instance, a placer can generate a valid placement that is unroutable. Our virtual topology ensures routability by leaving room for droplets to pass between adjacent modules where mixing, storage, and other assay operations are performed.

III. Virtual Topology

Our online interpreter utilizes a virtual topology, as seen in Fig. 5, and takes advantage of its order and structure to yield fast algorithmic runtimes for scheduling, placement and routing. First, we define a cell as the 2-D area covering an electrode. The virtual topology shows regularly spaced modules (3x3 squares of cells) where basic droplet operations (i.e., merge, mix, split, and store) are performed. If at least one of a module's cells is augmented with an external detector or heater, the module can also perform detect or heat operations, respectively. The white cells indicate the area of the DMFB array used explicitly for routing droplets between modules and I/O ports (not pictured); however, any cell can be used for routing if a module is not in use. Dedicated routing cells ensure there is a valid path between any source-destination pair. A perimeter of interference region (IR) cells surrounds each module [29], so that interference-free droplet routes can be computed easily. This topology ensures that there is at least one path between all DMFB inputs, modules, and outputs. The inputs and outputs (not shown in Fig. 5) are on the perimeter of the chip.

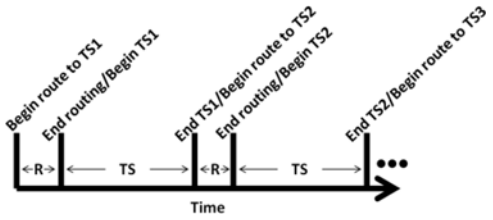

Fig. 7. Assay time-line showing that each fixed time-step (TS) is interleaved with a variable-length routing phase (R).

A. Module Topology and Synchronization

To help prevent droplet deadlock, droplets have well-defined module entrance and exit locations, as seen in the 3x3 module of Fig. 6(a). The two entrances are in the northwest and southwest corners, while the exits are in the northeast and southeast corners. By providing distinct entrances and exits, we prevent droplet deadlock by allowing droplets leaving a module to wait safely in their exit cells as long as necessary to avoid deadlock in the routing cells. Fig. 6(b) and (c) show that modules can be elongated along the x or y-axis to accommodate larger 2x4 mixers, often used in literature [19], [21].

As seen in Fig. 7, time-step stages of assay operations are interleaved with routing stages until the entire schedule has been processed. A time-step is the basic, minimum-resolution unit of time used to schedule microfluidic operations. Time-steps usually last one or two seconds, and are fixed in length for the duration of the assay. The routing stages are variable in length, depending on the routes that are generated, and can even be instantaneous if no droplets are being routed between time-steps.

Droplets are required to enter/exit a module at one of the two entrances/exits. When a droplet travels to a new module, it must enter the module during the routing phase at one of the module-entrance cells and wait until the time-step officially begins. The droplet is then processed (e.g., split, mixed, stored) during the time-step phase. If a droplet leaves the module after the current time-step, it must position itself at one of the module-exit cells before the end of the time-step. In Fig. 6(a) droplets 1 and 2 (D1/D2) enter a module to be processed while droplets 3 and 4 (D3/D4) exit to be processed elsewhere. If

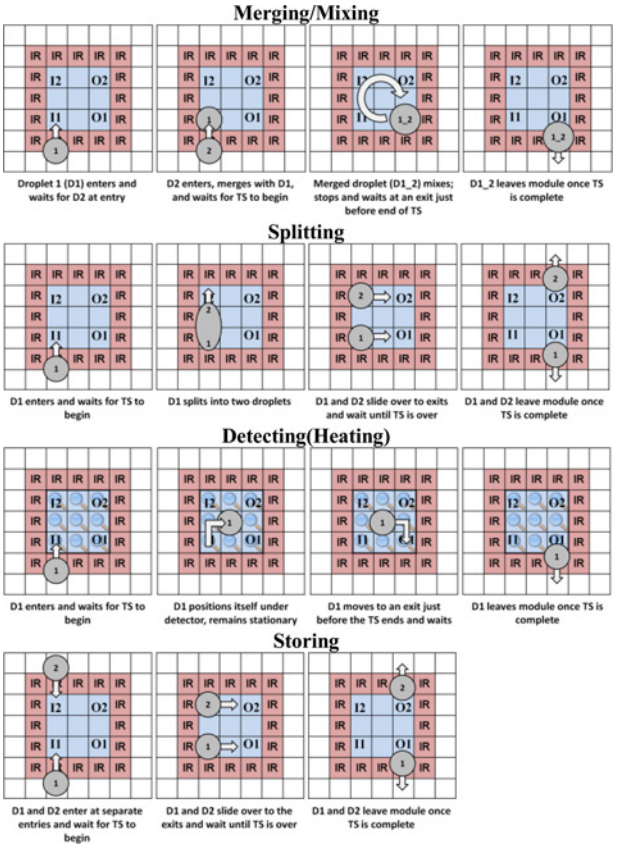

Fig. 8. Intramodule droplet processing/routing for microfluidic operations.

D1 and D2 arrive before D3 and/or D4 exit, there will be no conflict since the entrance and exit cells are sufficiently spaced to avoid droplet interference. When the time-step begins, D1 and D2 can move freely within the module, as D3 and D4 are at their respective destinations. This synchronization scheme prevents intermodule deadlocks because there is always an open spot at the destination module's entrances for every incoming droplet at every module.

Fig. 8 shows how a module can perform each assay operation. For each operation, the droplet(s) enters at one of the entrance cells and then waits for the time-step to begin. When the time-step begins, any droplets that were waiting in the exit cells are now gone, and thus, any remaining droplets in the module are free to move about the entire module to perform an operation. If the droplet(s) leaves the module at the end of the time-step, it moves to an exit cell before the time-step ends. Once the time-step is complete, during the subsequent routing stage, the droplet(s) exits the module. If a droplet is scheduled to begin a new operation in the same module at the next time-step, it maneuvers itself to an entrance cell before the time-step ends (not shown in Fig. 8); this eliminates the need for a droplet to exit and then reenter the same module.

IV. Fast Online Synthesis

In this section, we show how the virtual topology presented in Section III can be leveraged to create fast online synthesis methods for scheduling, placement and routing.

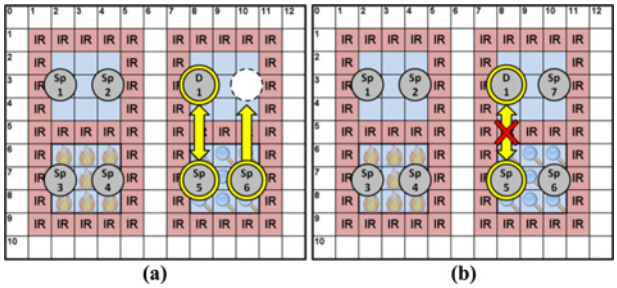

Fig. 9. Two DMFB scenarios with droplets that are going to be split (Sp) or detected (D) during the next time-step. (a) Sp6 can move and occupy the open space in another module, allowing D1 and Sp5 to swap so D1 can be detected in the detect-module. (b) There is no way to isolate a single droplet and since no droplets will be mixed next time-step, the assay cannot continue.

A. Scheduling

In this section we describe the definitions and constraints that must be observed during the scheduling phase. For online scheduling, a number of fast schedulers can be used with our topology that maintain the constraints defined in this section.

An assay is given to the scheduler in the form of a DAG, $G = (V, E)$, where the vertices (V) and edges (E) represent assay operations and operation dependencies, respectively. If the given DMFB is an $a_x \times a_y$ array of cells and each module is $m_x \times m_y$ cells, then the total number of modules, N_m, can be calculated as seen in (1). We add cells to the module dimensions to encapsulate the IR cells and the routing cells to the right (for the X dimension) of each module (Fig. 5)

$$\left\lfloor \frac{(a_x - 1)}{(m_x + 3)} \right\rfloor \times \left\lfloor \frac{(a_y - 3)}{(m_y + 1)} \right\rfloor = N_m. \tag{1}$$

Once the virtual topology is placed, modules with external devices above their cells are considered to be special modules (e.g., detect module and heat module). All other modules are considered to be basic modules. The array is initially populated based on the virtual topology. An array called $availMods[]$ contains the number of modules of each module-type (e.g., basic module, detect module, and so on), and satisfies the following condition:

$$\sum_{i=1}^{numModTypes} availMods[i] = N_m. \tag{2}$$

We define s_m to be the number of droplets a module can store and d_{max} to be the maximum number of droplets we allow on the DMFB during any time-step. Since each module has two entrance and two exit cells, a module can store two droplets during a time-step (i.e., $s_m = 2$). Consider Fig. 9(a) in which all but one of the modules is at maximum capacity. Since the northeast module has room for one droplet, droplets can be shuffled around so that any single droplet on the array can be isolated in any chamber, allowing the assay to continue. However, if all modules are at maximum capacity [Fig. 9(b)], then deadlock may arise because it is impossible to process more operations unless some of the droplets are scheduled to output or mix with each other next time-step. To reduce the likelihood of scheduling deadlock, we set the maximum

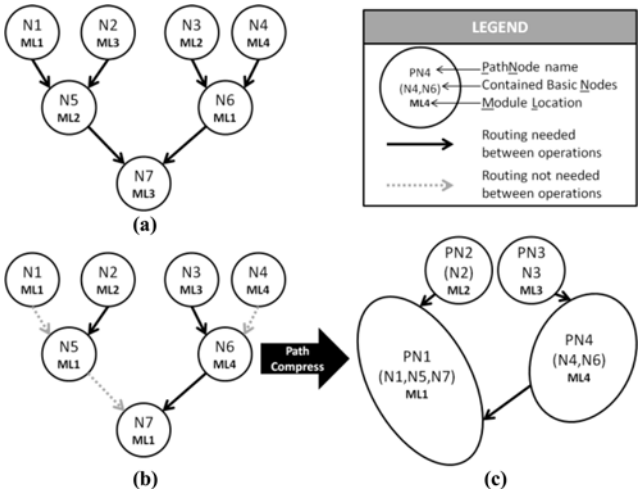

Fig. 10. (a) Randomly-bound sequencing graph for a simple assay requiring six droplet routes. (b) Sequencing graph with intelligent module selection requiring only three droplet routes. (c) Compressed sequencing graph.

```
1  // Initializations for graph variables
2  Given a scheduled sequencing graph of nodes: G = (V, E)
3  Storage operations: storageOps = V.storageOps
4  Input operations: inputOps = V.inputOps
5  Output operations: outputOps = V.outputOps
6
7  // Initializations for path-based variables
8  New sequencing graph of PathNodes: G_p = (V_p, E_p) = ∅
9  New List of PathNodes: pathLeaders = ∅
10 Operations by module-type: pathOpsByModType[numModTypes] = ∅
11
12 // Setting up path-based graph
13 pathLeaders = GetPathLeaders(G)
14 G_p = GeneratePathCompressedGraph(pathLeaders)
15
16 // Sort variables into bins and sort by start time
17 pathOpsByModType[ ] = SortOpsByModType(G_p)
18 Sort inputOps, outputOps, storageOps, all lists in pathOpsByModType[ ]
19                                                  ;ascending by start time
20 // Do binding of all nodes
21 LeftEdgeBind(pathOpsByModType[ ])
22 LeftEdgeBind(inputOps)
23 LeftEdgeBind(outputOps)
24 Storage by module location: storageByModLoc[numMods] = ∅;
25 storageByModLoc[ ] = SelectModuleLocations(storageOps) ;
26 BindStorageToHolders(storageByModLoc[ ]);
```

Fig. 11. Pseudocode for our path-based binder.

number of droplets permitted on the DMFB during any time-step (d_{max}) as follows:

$$d_{max} = (N_m \times s_m) - 1. \tag{3}$$

In our experimental section, we have successfully applied these constrains to two fast schedulers: list scheduling (LS) [26] and path scheduling (PS) [11]. LS is a greedy, constructive algorithm in which each operation (node) in an assay (DAG) is scheduled exactly once. LS is much faster than iterative improvement algorithms, which randomly compute numerous schedules [16], [21], [26] or optimal algorithms based on integer linear programming [26]; however, these approaches generally do produce higher quality schedules than LS. PS is another scheduler that attempts to schedule DAGs one path at a time (as opposed to a single node at a time). PS's runtimes have been found competitive to LS and produces superior schedules for assays with high fan-out. These schedulers were used because their fast runtimes allow them to be used in the context of online synthesis.

B. Placement

Microfluidic placement is NP-complete [27]; the virtual topology limits the reconfigurable capabilities of the DMFB by preplacing the location of modules. In our framework, operations are bound to preplaced modules in accordance with the schedule that has been computed *a priori*. The scheduler assigns operations to module-types (e.g., basic or specialized), but does not select a specific module for each operation; this is the job of the binder.

1) *Path-Based Binding Algorithm:* In this section, we present a more-intelligent, path-based binding algorithm that is inspired by Tseng's binding procedure for flow-based microfluidic biochips in which continuous operations are bound to the same component to reduce the amount of valve switching and overall assay completion time [30]. Tseng's algorithm was used for flow-based microfluidic devices, which are fundamentally different than DMFBs, and thus, is not directly applicable to DMFBs; however, the key principle that binding contiguously scheduled operations to the same component will reduce fluid transfers (droplet routes in the case of DMFBs) can be applied to both classes of microfluidic devices. This principle of spatial locality for contiguous operations was applied to path binder as described in the following sections to reduce droplet routing times.

In a previous work, Grissom and Brisk [10] present a fast binding solution based on the left-edge algorithm. When compared to the left-edge binder, the path-based binder is faster and performs binding in such a way that reduces route lengths. The left-edge binder does not take into account module-types or the locations of parent/child modules, instead, binds each operation to the first available module it finds with a matching module-type. Pathbinder takes parent/child module locations into consideration (reducing routing distances) and although it does use left-edge binding, performs preprocessing to reduce the graph, which eases algorithmic runtimes.

Fig. 10(a) shows a simple sequencing graph with seven nodes (for clarity, we will call these basic nodes for the remainder of this section). The edges denote operation precedence (e.g., N5 can only begin after N1 and N2 have completed); since each successive basic node has a different module location than its parent, the edges in Fig. 10(a) also denote droplets needing to be routed. Fig. 10(b) shows that certain routes can be eliminated if the binder selects the same module location for successive basic nodes(a key idea for path binder), allowing the router to sometimes produce shorter droplet routes because it will have less droplets to route. Furthermore, Fig. 10(c) shows that if successive nodes have the same module-type and location, they can be combined into path nodes, which contain a contiguous sub-set of basic nodes from the original sequencing graph (e.g. PN1 contains the sub-path N1, N5 and N7). When compared to the simple left-edge binder, the use of path nodes reduces the overall number of nodes in the sequencing graph, reducing the size of the problem and allowing for shorter algorithmic runtimes.

```
1  Given a list of path leaders: pathLeaders
2  Sequencing graph containing path leader PathNodes: G_p = pathLeaders
3
4  for (∀p : p ∈ pathLeaders)
5      Node n = p.nodes.front;
6      if (Type(n) == storage)
7          Create new PathNode, npn, containing Node n.children.front;
8          Add npn to pathLeaders, insert into G_p;
9      else // Traverse path
10         List uvc = GetUnVisitedChildren(n);
11         while (uvc.size > 0)
12             List ec = GetEligibleNodes(uvc); // n's eligible Children
13             List uec = GetIneligibleNodes(uvc); // n's ineligible Children
14             if (n.children.size > 1) // Split
15                 if (ec.size > 0)
16                     Add ec.front to p.nodes, add (ec − ec.front) to uec;
17                 for (∀uec_i : uec_i ∈ uec)
18                     Create new PathNode, npn, containing Node uec_i;
19                     Add npn to pathLeaders, insert into G_p;
20             else if (n.children.size == 1)
21                 if (ec.size > 0)
22                     Add ec.front to p.nodes, add (ec − ec.front) to uec;
23                 else
24                     Create new PathNode, npn, containing Node uec.front;
25                     Add npn to pathLeaders, insert into G_p;
26             end n.children.size if
27             if (ec.size > 0)
28                 n = ec.front;
29                 uvc = GetUnVisitedChildren(n);
30             else
31                 uvc = ∅;
32         end while
33     end Type(n) if
34 end ∀pl for
35 return G_p
```

Fig. 12. Pseudocode for the GeneratePathCompressedGraph() function.

```
1  Given a list of storage operations: storageOps
2
3  while (storageOps.size > 0)
4      Node sn = storageOps.RemoveFront( );
5      New List of ModuleLocations: potentialMods = ∅;
6      potentialModLocs = GetLongestFreeModLocs(sn);
7      ModuleLocation ml = GetClosestModLoc(potentialModLocs);
8      if (ml.end < sn.end)
9          Node snEnd = split(sn, ml.end);
10         Add snEnd to storageOps;
11     end if
12 end while
```

Fig. 13. Pseudocode for the SelectModuleLocations() function.

Fig. 11 presents high-level pseudocode for path binder. The binder is given a scheduled sequencing graph (line 2); at this point, each basic node/vertex,V, has a start time-step, stop time-step and module-type (e.g. mix module, detect module, etc.), but has not been bound to a particular module location. lines 3–5 obtain lists of important operation types (inputs, outputs and storage nodes); lines 7–10 initialize path-based variables.

Lines 12–14 construct the path-compressed graph, G_p. First, the initial path leaders are found, which are nodes whose parent nodes consist exclusively of input/dispense nodes (line 13); nodes with no parents (i.e., dispense nodes) are not included in this list. In line 14, the path leaders are passed to the GeneratePathCompressedGraph() function, which, at a high level, combines as many successive basic nodes as possible into larger path nodes, resulting in a new graph of path nodes. This function does not bind nodes to a particular module. We provide more details of this function in the following sub-section.

Lines 20–26 carry out all the binding. Line 22 performs a simple left-edge bind on the non-storage path nodes in G_p as performed in Grissom and Brisk's previous implementation of left-edge binding (except it is binding path nodes, instead of basic nodes) [10]. When a path node is bound to a particular module location, each basic node the path node contains is bound to that same module location. Inputs and outputs are bound (lines 22–23) as in the left-edge binder. Finally, lines 24–26 bind the storage nodes and complete the path binding algorithm; these functions are detailed in later sub-sections.

a) *Generating Path-Compressed Graph:* In order to reduce the work load of the binder and eliminate droplet routes for the subsequent routing stage, the sequencing graph is compressed such that a single path node contains one or more basic nodes, as demonstrated in Fig. 10(b) and (c). Eligible basic nodes can be compressed into a single path node if they form a path through the original sequencing graph (no gaps of time between basic nodes); an eligible node is a basic node that has not already been added to a path node in G_p, has the same resource-type as its path-parent, and is not an I/O or storage node. Ineligible nodes cannot be compressed into the current path node because, although they have not been added to G_p, they are of a different resource-type than their path-parent or are storage nodes. During the scheduling phase, non-storage operations are assigned a specific resource-type; since storage is extremely flexible, it is scheduled based on examining the number of free resources, but it is not assigned a specific resource-type. Thus, storage nodes are not path-compressed at this point because they will be broken up at a later stage to fit into any available resources.

Fig. 12 presents pseudocode for the path compression algorithm (Fig. 11, line 14). The resultant graph, G_p, is composed of a number of path nodes which each contain one or more basic nodes that can be bound to the same module location. Lines 4–34 show that each path leader is iterated through until there are no more path leaders, at which point the entire assay will be compressed. Each path leader (a path node) will initially contain exactly one basic node, which is examined in lines 5 and 6. Since storage is the most flexible operation and is designed to fit wherever other operations are not located, they are added to a new path node and added to the graph with no compression (lines 6–8).

Lines 9–33 attempt to traverse a path and compress eligible basic nodes into a single path node; lines 11–32 show that a path can be traversed while there are unvisited basic nodes (i.e., basic nodes not yet added to a path node in G_p) in the most-recently-added node's (n's) children. If n has multiple children (e.g., split operation), then only the first eligible child (randomly selected) is added to the current path node, p; a new path node is created for each remaining eligible and ineligible child and inserted into G_p and the path leaders list (lines 14–19). Similarly, if n only has one unvisited child, the child is either added to the current path node, p (if eligible), or used to create a new path node (if ineligible), as seen in lines 20–25. This loop (lines 11–32) continues until there are no more eligible children on the path.

b) *Selecting Storage Module Location:* Fig. 13 presents pseudocode to show how module locations are selected for storage nodes. Given a list of storage operations, lines 3–12

```
1  Given lists of storage nodes, sorted by module location: storageByModLoc[ ]
2  List of holder nodes, sorted by module location: holdersByModLoc[ ] = Ø
3
4  for each (ModuleLocation ml)
5      List holders = holdersByModLoc.at(ml)
6      List stores = storageByModLoc.at(ml)
7      sortByStartThenEnd(stores)
8
9      for each (s in stores)
10         if (no holders in holders)
11             (a)
12         else // holders already exist
13             int rStart = s.start // Running start for s
14             while (rStart < s.end)
15                 Node h = holders.getNext();
16                 if (rStart < h.start) // Starts before h
17                     if (s.end ≤ h.start) // s & h do not overlap
18                         (b)
19                     else // s & h overlap
20                         (c)
21                 else if (rStart == h.start) // Starts at same time as h
22                     if (s.end < h.end)  // s ends in middle of h
23                         (d)
24                     else // s ends at h
25                         (e)
26                 else // s ends after h
27                     (f)
28                 else if (h.start < rStart < h.end) // Starts in middle of h
29                     if (s.end > h.end) // s extends past h
30                         (g)
31                     else if (s.end == h.end) // s ends with h
32                         (h)
33                     else if (s.end < h.end) // s encompassed by h
34                         (i)
35                 else if (rStart ≥ h.end AND holders.hasNext() == false)
36                 // rStart  starts as or after h ends AND no more holders
37                     if (rStart  > s.start) // Part of s already bound
38                         (j)
39                     else
40                         (k)
41                 end rStart if
42             end while
43         end holders if
44     end stores for
45 end ModuleLocation for
```

Fig. 14. Pseudocode for the BindStorageToHolders() function (Fig. 11, line 26) with references (a)–(k) to pictorial pseudocode transformations in Fig. 15.

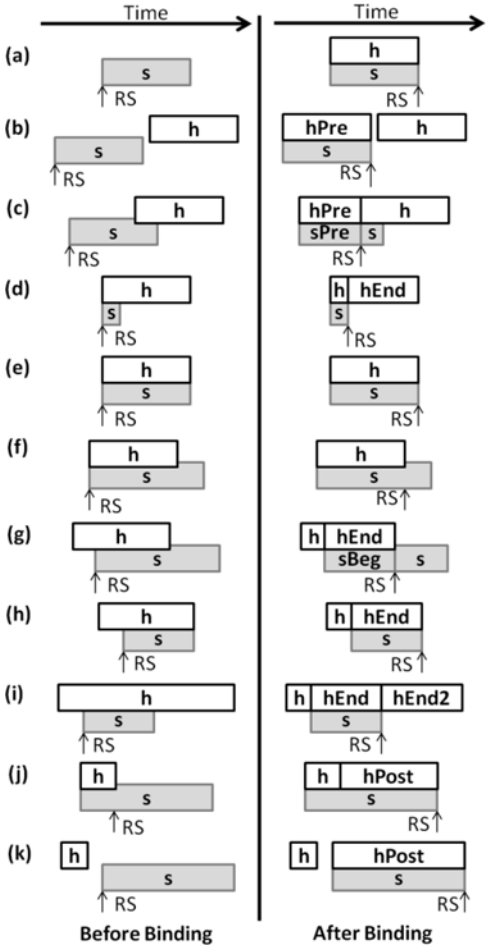

Fig. 15. (a)–(k) Transformations that take place at the corresponding times in the pseudocode in Fig. 14. RS denotes the rStart (running start) variable. An alignment of a storage (gray box) and holder (white box) node indicate that the storage node is bound to the overlapping holder node (after binding).

loop through and choose a module location for each storage operation, *sn*. In line 6, GetLongestFreeModLocs() examines all of the module locations and returns a list of one or more module locations with the longest uninterrupted availability, starting at *sn*'s starting time (*sn*.*start*). The main idea is to keep a droplet stored in a single location as long as possible since this minimizes the number of times a droplet needs to be routed. Next, in line 7, if there is more than one potential module location to choose from, GetClosestModLoc() selects the module location with the minimum distance to the storage node's already-bound parent or child, reducing any necessary routing lengths. Distance is computed as the Manhattan Distance between the top-left corners of the potential module location and the parent's/child's module location. Finally, if the selected module location was not free long enough to cover the entire length of *sn*, it is split and the second half is added to *storageOps* to be bound later (lines 8–10).

c) *Binding Storage To Holders:* Binding of storage nodes into storage holders is performed differently than in the left-edge binder. In the left edge binder, the minimal number of storage holders is created each time-step and each bound to a free module location (first free location in the list is selected if there is more than one available location); storage nodes (droplets) are then bound to a storage holder's location with no concern to the droplet's location. Path binder differs in that it first binds each storage node to a particular module location, and then creates storage-holders to accommodate these storage nodes. Thus, if resources permit, it is possible to have more than one module location being used to store less than s_m droplets. This uses more space (which would otherwise be unused) in exchange for spatial locality, which results in shorter droplet routes in the next stage.

Fig. 14 presents pseudocode for the BindStorageToHolders() function (Fig. 11, line 26). Instead of adding further pseudocode, Fig. 14 contains links (a)–(k) to pictorial transformations (Fig. 15) to more clearly explain the binding algorithm. Storage nodes are passed in and are already sorted by module location (line 1); holders are created by examining the storage nodes in one location at a time (line 4).

The algorithm attempts to bind each storage node, *s*, to any of the already-existing holders, *h* (lines 9–44), already created for that location (initially there are none). It does this by examining each holder's position relative to the storage node currently being examined. For example, case (a) shows the

case where there are no holders for that module location; thus, Fig. 15 illustrates that a new holder, h, is created to contain s. Lines 12–43 detail how storage is handled when there are holder nodes in existence. In these cases, s may overlap portions of one or more holders, and thus, the algorithm binds portions of s, from $s.start$ to $s.end$, until the entire storage node is bound (possibly being split in the process) to some number of storage holders. Fig. 14 shows that we hold a running-start variable ($rStart$) to denote that any portion of a storage node before $rStart$ has already been bound. Fig. 15 shows how much of the storage node is bound in each case by examining the before/after positions of the running-start (RS).

Examining Figs. 14 and 15, (b) and (c) handle the cases when a storage node's unbound portion begins before a holder; (d)–(f) handle the cases when a storage node's unbound portion begins at the same time as a holder; (g)–(i) handle the cases when a storage node's unbound portion begins in the middle of a holder. Finally, (j)–(k) handle the cases when the storage node's unbound portion starts after the last holder. If none of these cases apply, s is compared against the next holder until a case does apply.

In Fig. 15, storage and holder nodes named with suffixes (Pre, Beg, End and Post) show that new nodes were created during the binding process. In these cases, the original nodes in question (s or h) may have been shortened in length. A node's suffix (e.g., *"Pre"* in $hPre$) describes its position in relation to the original node with the name of the prefix ("h" in $hPre$). For example, as seen in Fig. 15(b) [Fig. 15(j)], after binding, a new node called $hPre(hPost)$ is created and exists entirely before (after) h's original position before binding; likewise, a node called $hBeg(hEnd)$ is one that spans a time-range, after binding, which was originally spanned by the beginning (end) of the prebound h.

C. Routing

To complete the synthesis flow, we use a simplified version of an existing droplet router by Roy *et al.* [22]. We created a number of routing methods that restricted routes to the cells in between modules, but found that Roy's maze-routing approach produced shorter routes in only a few more milliseconds of computation time compared to the alternatives. As in Roy's router, we use Soukup's fast maze router [23] to produce sequential routes for droplets and then compact the routes together, adding stalls in the middle of the routes to avoid droplet interference.

The routing algorithm by Roy *et al.* that we have implemented here, does not support rip-up and reroute. We chose this algorithm because it offers a good trade-off between runtime and route quality. Roy's algorithm works in two phases: 1) compute routes for all droplets using a variation of Soukup's VLSI routing algorithm (initially assuming that droplets are routed one-by-one) and 2) use a greedy algorithm to compact the droplets so that they can be routed concurrently without interfering. The routes are compacted in time, not space, and the pathways chosen in step 1) are never changed.

In principle, step 2) could be improved by adding the capability to rip-up and reroute certain droplet pathways, but that would require a longer runtime. Since our focus is online synthesis, where a premium is placed on runtime, we determine Roy's algorithm to be a reasonable solution. In the online context, the extra time spent performing these computations would be greater than the savings in execution time that is obtained from shorter routes.

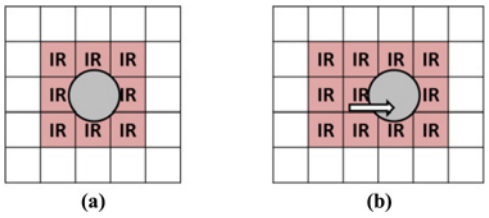

Fig. 16. (a) Interference region (IR) for a droplet at the beginning of a droplet-actuation cycle. (b) IR at the end of a cycle.

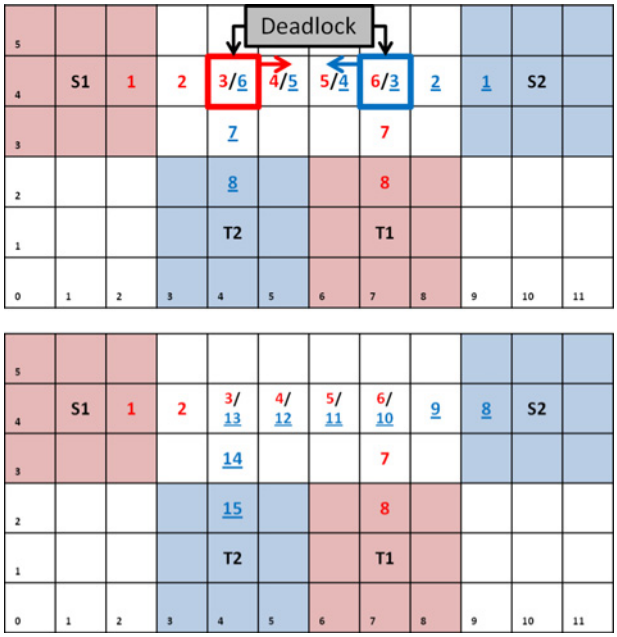

Fig. 17. Droplets 1 and 2 are traveling from source 1 and 2 (S1/S2) to target 1 and 2 (T1/T2), respectively. The red and blue (blue also underlined for clarity) numbers are time-stamps for droplets 1 and 2, respectively). The top scenario shows that deadlock can occur when routes 1 and 2 are compacted and stalls are added mid-route. The bottom scenario shows that both routes are safely completed if droplet 2 stalls at its source location until droplet 1 is safely out of the way.

The router receives a scheduled and placed DAG, from the placer. Throughout the routing process, all droplets in motion must maintain static and dynamic spacing constraints to prevent interference, as shown in Fig. 16. Droplet routes are computed one routing sub-problem at a time. As seen in Fig. 17, a routing sub-problem (or phase) is the problem of routing a number of droplets from their source (input or module) to their destination (module or output); routing sub-problems occur between the end of one time-step and the beginning of the subsequent time-step.

During a routing sub-problem, blockages are created and must be avoided. For a particular routing sub-problem t, any persisting module, m, that is performing operations (i.e., $m.Start < t < m.End$) is considered a blockage (including its interference region). In addition, for each droplet d_i,

the source and target (including their interference regions, a 3x3 blockage) for any droplet d_j also being routed during the same sub-problem are considered as blockages for d_i. Due to the virtual topology, the sources/targets for all d_j will never interfere with d_i, and thus, deadlock-freedom is guaranteed.

For a specific sub-problem, individual routes are first computed for each droplet using Soukup's fast maze routing algorithm [23]. Soukup's maze router works by routing around blockages; it routes straight to its destination until it hits a blockage (e.g., existing module or droplet), at which point it attempts to route around it.

We do modify Roy's router, however, taking advantage of the virtual topology to avoid deadlock (i.e., when droplets form a dependency cycle and cannot move forward until one of the droplets in the dependency cycle concedes).

Route compaction is the process of taking a number of sequential routes and causing the droplets to move in parallel at the same time. However, the original routes are not created with concern to other droplet routes and caution must be taken when compacting to prevent routes from intersecting in time and space. When compacting routes, droplets may not enter any cell that is adjacent to any other droplet or the droplets will interfere (merge) with one another. To prevent this, a static interference region (IR) is created around each droplet at the beginning of each droplet-actuation cycle, as seen in Fig. 16(a). As a droplet moves from one cell to the next, the IR is stretched dynamically to include the union of the static IRs of the beginning and end cells [see Fig. 16(b)]. In general, a droplet may not enter any other droplet's IR while routing. Static and dynamic droplet interference rules are formally defined in [29].

It is possible that deadlock may occur during compaction if two (or more) droplets are waiting for each other to move. In this case, stalling cannot resolve the deadlock (e.g., consider the case where two droplets are attempting to enter the same cell but cannot because it would cause a head-on collision).

Roy's router attempts to recover from deadlock by moving one of the droplets backward [22]. We simplify the process by taking advantage of our virtual topology. With our module synchronization, described in Section III-A, droplets have designated sources (module exits) and destinations (module entries) that do not interfere with any other sources and destinations in a given time-step (i.e., a droplet source will never interfere with another droplet's destination). Thus, a droplet can stay at its source as long as necessary, until all other droplets are safely off its path, and then commence its route. By employing this method, we are guaranteed to avoid deadlock.

With this in mind, the router keeps track of the number of stalls added to any route r. If the number of stalls added to route r reaches some threshold, *stallThresh*, all of the stalls added to any route thus far in the sub-problem are removed. Then, the entire sub-problem is compacted again, except this time, stalls are added to the beginning of the routes instead of the middle. In this case, droplets do not leave the safety of their source cell until they are guaranteed an unobstructed path in space and time to their destination.

Consider Fig. 17 in which droplets 1 and 2 are being routed from their sources (S1 and S2) to their target cells (T1 and T2).

TABLE I
ASSAY BENCHMARK TABLE

Benchmark	Number of Operations				Dispense Time
	Inputs	Outputs	Detects	Mix/Split	
PCR	8	1	0	7	2
InVitro1	8	4	4	4	2
InVitro2	12	6	6	6	2
InVitro3	18	9	9	9	2
InVitro4	24	12	12	12	2
InVitro5	32	16	16	16	2
ProteinSplit1	12	2	2	12	2
ProteinSplit2	24	4	4	26	2
ProteinSplit3	48	8	8	54	2
ProteinSplit4	96	16	16	110	2
ProteinSplit5	192	32	32	222	2
ProteinSplit6	384	64	64	446	2
ProteinSplit7	768	128	128	894	2

Table of Benchmarks Showing the Number of Different Operation Types and Dispense Times

As seen in the top scenario, if the routes start at the same time, deadlock will occur at cycle 3 as droplet 1, at cell (4, 4), and droplet 2, at cell (7, 4), cannot move forward without merging. No amount of mid-route stalls will resolve this deadlock since they are heading straight toward each other; it is not a matter of allowing one droplet to pass. The bottom scenario shows that if droplet 2 is allowed to stay in its source until droplet 1 is safely off its route, droplet 1 can reach its target. Since the cells around S2 are considered as blockages to droplet 1, droplet 2 is safe to wait at S2 as long as necessary because droplet 1 will never attempt to pass through that area, even if its destination is to the east of S2.

Adding stalls to the beginning of a path will always work and will never result in deadlock, as can occur when inserting stalls mid-route; however, we discovered empirically that inserting stalls mid-route tends to yield shorter routes, and rarely results in deadlock. Thus, we employ the mid-route-stall compaction method first and revert to the preroute-stall compaction method only when a deadlock occurs.

V. EXPERIMENTS

A. Benchmarks

We used three benchmark families: PCR, in-vitro diagnostics, and a protein assay (see Table I), whose base DAGs have been made publicly available by researchers at Duke University [25]; we also used the provided module libraries to obtain operation timings. We used a 4×2 mixing times (3s) for all PCR mixing operations. In-vitro diagnostics is a family of assays that mixes and detects up to four samples with four reagents (e.g., up to 16 mix-and-detect operations). We use the five in-vitro assays, along with mixing/detection times, as listed in [24, Table I].

We also use the protein-split benchmark, described in [11], which represents the traditional protein assay with varying levels of splitting from one to seven (the traditional protein has three levels, with $2^3 = 8$ output droplets); all operation timings are the same as the protein assay. These assays are used as large problem instances to push the synthesis flow's capabilities. For the protein assay, we used 4×2 dilution times

(5 s) and 4×2 mixing times (3 s) for all dilute and mixing operations, respectively; all 2-input, 2-output dilute operations in the protein assay were implemented using a mix operation, followed by a split operation, which took 5 s in total.

We assume a droplet actuation frequency of 100 Hz [34] and all droplet input times are assumed to be two seconds in length. The Protein Split assays in Experiments 1 and 2 were all scheduled using path scheduler [11]. All assays in Experiment 3 were scheduled with list scheduler [26]. Furthermore, although the virtual topology uses 4×2 mix and dilution times, it still uses 4×3 modules for module synchronization purposes; the 4×2 module was the largest/fastest module described in the Duke benchmark document that would fit inside our standard 4×3 module [25]. The free placer in Experiment 3 uses 4×2 mixing times in a 4×2 module since it does not need the extra space for module synchronization.

B. Implementation Details

All code was implemented in C++ using the University of California, Riverside's (UCRs) DMFB Synthesis Framework [9]. We evaluated performance on a 64-bit Windows 7 desktop PC, with 4GB of RAM and an Intel Core i7^{TM} CPU operating at 2.8 GHz. This platform represents a typical use case for a controlled laboratory setting.

C. Experiment 1: Left-Edge Binding Versus Path Binding

We first compared the left-edge binder with the path binder, both described in Section IV-B, on a 15W $\times$ 19H DMFB with the basic topology described in Fig. 5 with 4×3 modules such that 6 modules could safely be placed onto the DMFB. The objective was to experimentally verify that the use of path binder leads to shorter routes and algorithmic times, despite the seemingly-added complexity of path binder and its preprocessing computations. We evaluate the two algorithms on the ProteinSplit family of assays, as they provide increasingly-larger problem instances as the number of split-levels is increased from 1–7 (14 nodes to 1022 nodes). Table II shows the results for left-edge and path binding. For ProteinSplit 1–5, the problem instances are too small to really see a difference in computation time. However, as the assays grow larger (ProteinSplit 6 and 7) path binder's improvements are clearly seen since it produces a valid binding $10 \times$ faster than the left-edge binder.

Table II also shows the total length of the droplet routes generated by the router stage (described in Section IV-C) when given the bindings for each benchmark. The results show that the routing lengths are shorter for all but the smallest benchmark (ProteinSplit1), saving up to 3.8 s on the largest assay. It should also be noted that, although not seen in Table II, the computation time for routing is decreased due to the spatial enhancements of path binder; from ProteinSplit 1–7, the router saves from 2 ms to 6.4 s, respectively, further adding to the time savings when using pathbinder. For the remainder of this paper, we use path binder as our binder of choice.

D. Experiment 2: Topology Exploration

Here, we explore several topological configurations and the effects on routing. Fig. 18 shows three different configurations

TABLE II
LEFT-EDGE BINDING VERSUS PATH BINDING

Benchmark	Left-Edge Binding		Path Binding	
	Comp. Time (ms)	RL (s)	Comp. Time (ms)	RL (s)
ProteinSplit1	0	1.10	0	1.22
ProteinSplit2	0	3.42	0	3.18
ProteinSplit3	0	7.57	0	6.83
ProteinSplit4	1	17.02	0	14.23
ProteinSplit5	3	36.99	1	29.68
ProteinSplit6	21	73.49	2	57.43
ProteinSplit7	99	147.39	9	108.76

Results showing the route lengths (RL) and computation times for left-edge and path binding performed on seven ProteinSplit (PS) benchmarks on a 15W×19H DMFB.

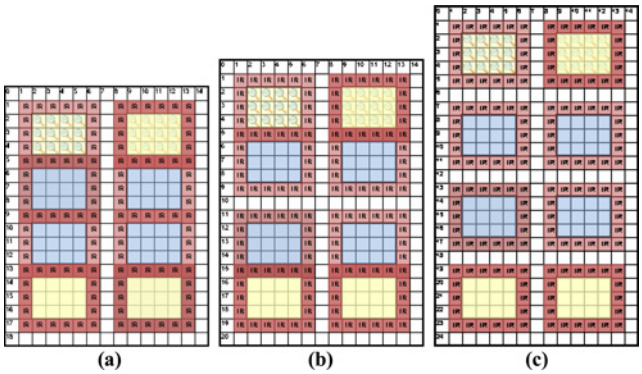

Fig. 18. Three different topologies showing modules stacked vertically (2W×4H). (a) No horizontal routing channels (HRC, the white cells between modules) between modules. (b) One HRC between every two modules. (c) One HRC between every module.

with horizontal routing channels (HRCs) interspersed at varying regularities between vertical groups of modules. An HRC is a group of contiguous horizontal cells that extends from side to side and will never be occupied by a module or its interference region. Fig. 18(a) shows the tightest configuration, which is the case where there are no HRCs. Fig. 18(b) and (c) illustrate the cases where there is a single HRC between every two modules and every module, respectively. The design seen in Fig. 18(c) allows for maximum routability and provides the fewest blockages (at the cost of using more space). Fig. 18(a) is the tightest design (with the most blockages for routing); Fig. 18(b) presents a compromise between the two.

In Tables III and IV, we show how the topologies affect schedule length and routing times. Table III presents results for the ProteinSplit assays when the DMFB size is fixed. This shows that certain topologies, which make less room for routing, can fit more modules in some instances. For example, as seen in Table III, the tight topology with no HRCs [similar to that seen in Fig. 18(a)] could fit ten modules on a 15W $\times$ 23H DMFB, while the topology with one HRC between each module [similar to that seen in Fig. 18(c)] could only fit six modules. The results are clearly seen in that, as the number of modules increases, the schedule lengths are reduced.

Table IV gives results for the ProteinSplit assays when the number of resources are fixed (eight modules), in order to show the results on routing. In this case, the DMFB topologies

TABLE III
SCHEDULE LENGTH (S) FOR FIXED-SIZE DMFB (15W × 23H)

HRC Spacing	# Mods	ProteinSplit (PS) Benchmark						
		PS1	PS2	PS3	PS4	PS5	PS6	PS7
None	10	55	70	95	155	270	505	987
Every 2 Mods	8	55	70	108	175	317	609	1,213
Every Mod	6	55	70	119	218	418	864	1,796

Results showing the number of modules that can fit and the resultant schedule lengths of three topologies with different horizontal routing channel (HRC) spacing; each topology is placed onto a 15W×23L DMFB.

TABLE IV
TOTAL ROUTE LENGTH (S) FOR FIXED-MODULE-COUNT (EIGHT MODS)

Benchmark	Sched. Length (s)	HRC Spacing		
		None	Every 2 Mods	Every Mod
ProteinSplit1	55	1	1	1
ProteinSplit2	70	3	3	3
ProteinSplit3	108	7	8	9
ProteinSplit4	175	15	16	17
ProteinSplit5	317	29	30	33
ProteinSplit6	609	61	62	69
ProteinSplit7	1213	123	124	136
		15W×19H (285)	15W×21H (315)	15W×25H (375)
		DMFB Dimensions (# Electrodes)		

Results showing the sizes of the DMFBs and resultant route lengths for three topologies with different horizontal routing channel (HRC) spacing; each DMFB is sized to fit eight modules with the given topology.

and sizes are exactly those seen in Fig. 18. The purpose of the HRCs is to create shortcuts for droplets that must otherwise travel all the way to the north/south border and around the entire stack of modules to get to its destination (the extreme cases) if all modules are busy. As seen in Table IV, the most compact topology (with no HRCs) produces the shortest overall routes for every benchmark. Thus, these results show that the elimination of occasional worst-case routing situations does not offset the constantly shorter distances droplets travel between modules in the most compact topology with no HRCs. Furthermore, as stated in Section III, droplets can cut through inactive modules (essentially creating a temporary HRC) to reduce routing times. Hence, Tables III and IV show that the topology with no HRCs [Fig. 18(a)] uses the least space (which can lead to greater utilization and shorter schedules) and yields the shortest routing lengths.

E. Experiment 3: Comparison To Fast Free Placer

In this section, we highlight the key benefits of the virtual topology and binder by comparing Path Binding to a fast free placement algorithm known as Keep All Maximal Empty Rectangles (KAMER) placement [4], [16]. The KAMER placer works by quickly computing all the maximal empty rectangles (MERs) (i.e., the empty rectangles that cannot be contained within another empty rectangle) and then placing a module within one of the MERs. We chose KAMER placement as a fair comparison because it is very fast and used in other online synthesis works [3].

We compare the KAMER placer (KP) to path binding (PB) with the virtual topology seen in Fig. 18(a) (no HRCs), both on an identical 15W×19H DMFB, such that eight mixers could be accommodated. Both synthesis flows used list scheduling [26] and Roy's maze router [22], described in Sections IV-A and IV-C, respectively. The schedules, computed as input to the KAMER placer and path binder, were identical.

We experimented with two and three storage droplets (PB_2/KP_2 and PB_3/KP_3) per module for the two methods/flows. For PB_2, storage is handled as described in Fig. 8 (storage enters via I1/I2 and leaves via O1/O2). For PB_3, when three droplets were allowed to be stored per module, we allowed the router to break the module I/O synchronization rules by allowing the third droplet to enter via O1; the two droplets that entered via I1 and I2 remained there and also exited via I1 and I2. All modules used by PB were 4×3 cells; KP was able to use smaller 4×2 modules since it does not need to enforce droplet synchronization rules. For storage, KP_2/KP_3 places two/three single-cell (1×1) modules (which is the common storage-module size for free placement [26]) instead of storing two/three droplets in a larger 4×2 mixing module.

Table V shows the results for ten runs of PCR, InVitro1-5 and ProteinSplit 1–6 for PB and KP for two and three storage droplets per module; PB_2 is the solution presented in this paper. The first section shows that, in 10 runs, PB_2 has no failures until ProteinSplit 6, when list scheduling fails because there are not enough resources for it to schedule such a large assay. PB_3 fails completely on routing on ProteinSplit 4–6. The third section shows the schedule length and total assay time (which includes routing); this section shows that PB_2 and PB_3s schedules did not differ until ProteinSplit 4–6. Thus, the scheduler did not need 3 droplets per module until ProteinSplit 4, showing that routing failed for PB_3 as soon as the system attempted to actually bind three droplets to a single module.

KP_2 shows that, even with only two storage droplets per mix module being scheduled, placement and routing errors occur often; KP_3 shows similar results. Thus, although allowing for three droplets per module produces better schedules, it is clear from the results that doing so yields more congestion, making it difficult to produce valid routing solutions for both binding and free placement. This suggests that it is unwise to attempt scheduling more than two droplets per (4×2/4×3) mix module.

The middle section of Table V shows the synthesis times for placement and routing of the first successful run of the ten runs, if any existed. The results show that both placers are extremely fast (milliseconds), with PB being slightly faster or equal to KP in all comparable instances, making it suitable for online synthesis. Finally, as seen in the third section of Table V, when comparing PB_2 versus KP_2 and PB_3 versus KP_3 (since both pairs have the same schedule), PB produces overall shorter routing times than KP since it reduces the number of droplets that need to be routed by binding contiguous operations to the same module (location) when possible.

Overall, Table V supports our decision to limit storage to two droplets per module and shows that, even though more droplets could be placed in our modules, they cannot be reliably routed. The results also demonstrate that, although KP

TABLE V
PATH BINDING (PB) WITH VIRTUAL TOPOLOGY VERSUS KAMER PLACER (KP)

Assay	# Scheduling/Placement/Routing Failures in 10 Runs				Place/Route Comp. Time (ms) (First Success)				Schedule/Assay Length (s) (First Success)			
	PB_2	PB_3	KP_2	KP_3	PB_2	PB_3	KP_2	KP_3	PB_2	PB_3	KP_2	KP_3
PCR	-	-	-	-	0 / 0	0 / 0	0 / 0	0 / 0	12 / 12.44	12 / 12.44	12 / 12.74	12 / 12.79
InVitro1	-	-	-	-	0 / 0	0 / 0	0 / 0	0 / 0	15 / 15.64	15 / 15.64	15 / 16.49	15 / 16.48
InVitro2	-	-	-	-	0 / 1	0 / 1	0 / 1	0 / 1	19 / 20.28	19 / 20.28	19 / 20.79	19 / 20.93
InVitro3	-	-	-	-	0 / 1	0 / 1	0 / 2	0 / 2	19 / 20.49	19 / 20.49	19 / 21.76	19 / 21.9
InVitro4	-	-	1 RF	1 RF	0 / 2	0 / 2	0 / 2	0 / 2	23 / 25.01	23 / 25.01	23 / 26.35	23 / 26.31
InVitro5	-	-	1 RF	1 RF	0 / 3	0 / 3	0 / 4	0 / 4	29 / 31.48	29 / 31.48	29 / 33.54	29 / 32.84
ProteinSplit1	-	-	-	-	0 / 1	0 / 1	0 / 2	0 / 2	53 / 53.73	53 / 53.73	53 / 54.78	53 / 54.76
ProteinSplit2	-	-	-	-	0 / 4	0 / 3	0 / 4	0 / 5	63 / 64.83	63 / 64.85	63 / 67.95	63 / 67.89
ProteinSplit3	-	-	3 PF, 6 RF	4 PF, 5 RF	0 / 8	0 / 8	1 / 11	1 / 10	84 / 88.85	84 / 88.81	84 / 94.12	84 / 94.33
ProteinSplit4	-	10 RF	10 PF	10 PF	1 / 27	-	-	-	215 / 223.21	175 / -	215 / -	175 / -
ProteinSplit5	-	10 RF	10 PF	10 PF	2 / 76	-	-	-	486 / 513.08	363 / -	486 / -	363 / -
ProteinSplit6	10 SF	10 RF	10 SF	10 PF	-	-	-	-	- / -	725 / -	- / -	725 / -

Results showing path binding (PB) Versus KAMER placement (KP) with two and three storage droplets per mixing module on a 15W×19L DMFB. The first section shows the number of scheduling/placement/routing failures (SF/PF/RF) in ten runs ('-' means no failures). The second section shows the computation time of placement and routing for the first successful run ('-' means all ten runs were failures and no timing was measured) of each flow. The third section shows the schedule length and the total length of the assay (which includes the routing time).

uses less space for modules, the chaotic and super-compact placements make it difficult-to-impossible for routing. We should note that we also tried a version of KP which left additional space around modules to improve routability; however, this configuration of KP performed worse than the version presented in Table V, often failing on placement because there was not enough room to randomly place the modules freely with the extra space.

VI. CONCLUSION

The online synthesis flow introduced in this paper can run in real-time on a typical laboratory desktop system, as typified by the Intel i7 processor used in our experiments. Empirically, this paper has shown that a virtual topology coupled with a binding algorithm can greatly simplify the placement problem, ease the router's job and lead to better droplet routes. We present a basic left-edge binder and a more-intelligent path-based binder which bind assay operations to module locations. The first simply computes a valid binding solution, while the latter takes spatial and temporal locality into account to produce better solutions.

The topology is designed to facilitate basic microfluidic operations and ensure that any droplet's source-destination pair can be quickly computed without fail on the first try. These features are vital in an online environment where recomputing synthesis stages will be felt by the user as he or she waits. We demonstrate that a compact topology produces better results both in scheduling and routing than sparse topologies designed to allow more room for routing. We also show tiling modules vertically, with a width-to-height ratio slightly below zero, yields the best routing results.

Our future work will extend the online synthesis flow to account for control flow operations that cannot be predicted at compile-time, including variable-latency assay operations, and runtime fault detection and recovery; of particular interest is the ability to dynamically reconfigure the virtual topology when permanent errors are detected; when this occurs, the online flow will be invoked to resynthesize the assay at runtime.

REFERENCES

[1] D. Grissom and P. Brisk. (2012). *UCR Microfluidics lab* [Online]. Available: http://www.microfluidics.cs.ucr.edu

[2] M. Alistar, E. Maftei, P. Pop, and J. Madsen, "Synthesis of biochemical applications on digital microfluidic biochips with operation variability," in *Proc. DTIP MEMS/MOEMS*, 2010, pp. 350–357.

[3] M. Alistar, P. Pop, and J. Madsen, "Online synthesis for error recovery in digital microfluidic biochips with operation variability," in *Proc. DTIP MEMS/MOEMS*, 2010, pp. 350–357.

[4] K. Bazargan, R. Kastner, and M. Sarrafzadeh, "Fast template placement for reconfigurable computing," *IEEE Design and Test of Computers*, vol. 17, no. 1, pp. 68–83, Jan.–Mar. 2000.

[5] K. F. Böhringer, "Modeling and controlling parallel tasks in droplet-based microfluidic systems," *IEEE Trans. Computer-Aided Design Integr. Circuits Syst.*, vol. 25, no. 2, pp. 334–344, Feb. 2006.

[6] M. Cho and D. Z. Pan, "A high-performance droplet routing algorithm for digital microfluidic biochips," *IEEE Trans. Computer-Aided Design Integr. Circuits Syst.*, vol. 27, no. 10, pp. 1714–1724, Oct. 2008.

[7] S.-K. Fan, C. Hashi, and C.-J. Kim, "Manipulation of multiple droplets on NxM grid by cross-reference EWOD driving scheme and pressure-contact packaging," in *Proc. IEEE MEMS*, 2003, pp. 694–697.

[8] E. J. Griffith, S. Akella, and M. K. Goldberg, "Performance characterization of a reconfigurable planar-array digital microfluidic system," *IEEE Trans. Computer-Aided Design Integr. Circuits Syst.*, vol. 25, no. 2, pp. 345–357, Feb. 2006.

[9] D. Grissom, K. O'Neal, B. Preciado, H. Patel, R. Doherty, N. Liao, *et al.*, "A digital microfluidic biochip synthesis framework," in *Proc. VLSI-SoC*, 2012, pp. 177–182.

[10] D. Grissom and P. Brisk, "Fast online synthesis of generally programmable digital microfluidic biochips," in *Proc. CODES + ISSS*, 2012, pp. 413–422.

[11] D. Grissom and P. Brisk, "Path scheduling on digital microfluidic biochips," in *Proc. DAC*, 2012, pp. 26–35.

[12] T. Huang and T. Ho, "A fast routability- and performance-driven droplet routing algorithm for digital microfluidic biochips," in *Proc. IEEE ICCD*, 2009, pp. 445–450.

[13] T. Ho, K. Chakrabarty, and P. Pop, "Digital microfluidic biochips: Recent research and emerging challenges," in *Proc. CODES + ISSS*, 2011, pp. 335–343.

[14] C. Liao and S. Hu, "Multiscale variation-aware techniques for high-performance digital microfluidic lab-on-a-chip component placement," *IEEE Trans. NanoBiosci.*, vol. 10, no. 1, pp. 51–58, Mar. 2011.

[15] Y. Luo, K. Chakrabarty, and T. Ho, "A cyberphysical synthesis approach for error recovery in digital microfluidic biochips," in *Proc. DATE*, 2012, pp. 1239–1244.

[16] E. Maftei, P. Pop, and J. Madsen, "Tabu search-based synthesis of dynamically reconfigurable digital microfluidic biochips," in *Proc. CASES*, 2009, pp. 195–203.

[17] J. H. Noh, J. Noh, E. Kreit, J. Heikenfeld, and P. Rack, "Toward active-matrix lab-on-a-chip: Programmable electrofluidic control enabled by arrayed oxide thin film transistors," *Lab Chip*, vol. 12, no. 2, pp. 353–360, Dec. 2011.
[18] K. O'Neal, D. Grissom, and P. Brisk, "Force-directed list scheduling for digital microfluidic biochips," in *Proc. VLSI-SoC*, 2012, pp. 177–182.
[19] P. Paik, V. Pamula, and R. Fair, "Rapid droplet mixers for digital microfluidic systems," *Lab Chip*, vol. 3, no. 4, pp. 253–259, Sep. 2003.
[20] M. G. Pollack, A. D. Shenderov, and R. B. Fair, "Electrowetting-based actuation of droplets for integrated microfluidics," *Lab Chip*, vol. 2, no. 2, pp. 96–101, 2002.
[21] A. J. Ricketts, K. Irick, N. Vijaykrishnan, and M. J. Irwin, "Priority scheduling in digital microfluidics-based biochips," in *Proc. DATE*, 2006, pp. 329–334.
[22] P. Roy, H. Rahaman, and P. Dasgupta, "A novel droplet routing algorithm for digital microfluidic biochips," in *Proc. GLSVLSI*, 2010, pp. 441–446.
[23] J. Soukup, "Fast maze router," in *Proc. DAC*, 1978, pp. 100–102.
[24] F. Su and K. Chakrabarty, "Architectural-level synthesis of digital microfluidics-based biochips," in *Proc. ICCAD*, 2004, pp. 223–228.
[25] F. Su and K. Chakrabarty. (2006) *Benchmarks for digital microfluidic biochip design and synthesis* [Online]. Available: http://www.ee.duke.edu/fs/Benchmark.pdf
[26] F. Su and K. Chakrabarty, "High-level synthesis of digital microfluidic biochips," *ACM J. Emerg. Technol. Comput. Syst.*, vol. 3, no. 4, pp. 16.1–16.32, Jan. 2008.
[27] F. Su and K. Chakrabarty, "Module placement for fault-tolerant microfluidics-based biochips," *ACM Trans. Design Autom Electron Syst*, vol. 11, no. 3, pp. 682–710, Jul. 2006.
[28] F. Su and K. Chakrabarty, "Unified high-level synthesis and module placement for defect-tolerant microfluidic biochips," in *Proc. DAC*, 2005, pp. 825–830.
[29] F. Su, W. Hwang, and K. Chakrabarty, "Droplet routing in the synthesis of digital microfluidic biochips," in *Proc. DATE*, 2006, pp. 323–328.
[30] K.-H. Tseng, S.-C. You, W. H. Minhass, T.-Y. Ho, and P. Pop, "A network-flow based valve-switching aware binding algorithm for flow-based microfluidic biochips," in *Proc. ASP-DAC*, 2013, pp. 213–218.
[31] Z. Xiao and E. F. Y. Young, "Placement and routing for cross-referencing digital microfluidic biochips," *IEEE Trans. Computer-Aided Design Integr. Circuits Syst.*, vol. 30, no. 7, pp. 1000–1010, Jul. 2011.
[32] T. Xu and K. Chakrabarty, "Broadcast electrode-addressing for pin-constrained multi-functional digital microfluidic biochips," in *Proc. DAC*, 2008, pp. 173–178.
[33] T. Xu and K. Chakrabarty, "Integrated droplet routing in the synthesis of microfluidic biochips," in *Proc. DAC*, 2007, pp. 948–953.
[34] P.-H. Yuh, C.-L. Yang, and Y.-W. Chang, "BioRoute: A network-flow-based routing algorithm for the synthesis of digital microfluidic biochips," *IEEE Trans. Computer-Aided Design Integr. Circuits Syst.*, vol. 27, no. 11, pp. 1928–1941, Nov. 2008.
[35] P.-H. Yuh, C.-L. Yang, and Y.-W. Chang, "Placement of defect-tolerant digital microfluidic biochips using the T-tree formulation," *ACM J. Emerg. Technol. Comput. Syst*., vol. 3, no. 3, pp. 13.1–13.32, Nov. 2007.

Daniel T. Grissom received the B.S. degree in computer engineering from the University of Cincinnati, Cincinnati, OH, USA, in 2008, and the M.S. degree in computer science from the University of California (UCR), Riverside, CA, USA, in 2011. He is currently pursuing the Ph.D. degree in computer science at UCR.

He has published seven papers, one journal, and filed one provisional patent. His current research interests include languages, synthesis, and hardware interfacing for digital microfluidic biochips.

Mr. Grissom was the recipient of the National Science Foundation's Graduate Research Fellowship Program Award in 2010, and is a member of the ACM and ACM SIGBED.

Philip Brisk (M'09) received the B.S., M.S., and the Ph.D. degrees, all in computer science, from University of California, Los Angeles, LA, USA, in 2002, 2003, and 2006, respectively.

From 2006–2009, he was a Post-Doctoral Scholar in the Processor Architecture Laboratory in the School of Computer and Communication Sciences, École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland. He is currently an Assistant Professor with the Department of Computer Science and Engineering, Bourns College of Engineering, University of California, Riverside, CA, USA. His current research interests include FPGAs, compilers, and design automation and architecture for application-specific processors.

Dr. Brisk was a recipient of the Best Paper Award at the International Conference on Compilers, Architecture, and Synthesis, in 2007, and the International Conference on Field Programmable Logic and Applications, in 2009. He is a member of the program committees of several international conferences and workshops, including Design Automation and Test in Europe, the IEEE Symposium on Application-Specific Processors, the International Workshop on Software and Compilers for Embedded Systems, and the Reconfigurable Architecture Workshop. He was also the General Chair of the 4th IEEE Symposium on Industrial Embedded Systems, in 2009, and the General Co-Chair of the 8th IEEE Symposium on Application Specific Processors, in 2010.