UNIVERSITY OF CALIFORNIA
RIVERSIDE

Design of Topologies for Interpreting Assays on Digital Microfluidic Biochips

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Daniel Thomas Grissom

June 2014

Dissertation Committee:
Dr. Philip Brisk, Chairperson
Dr. Frank Vahid
Dr. Qi Zhu
Dr. Walid Najjar

The Dissertation of Daniel Thomas Grissom is approved:

Committee Chairperson

University of California, Riverside

ACKNOWLEDGMENTS

I would like to thank Dr. Philip Brisk for his continual dedication to, not only our microfluidic research, but also to my personal growth and well-being. From the day I met Dr. Brisk, he has shown a true desire to challenge me, while ensuring that I am enjoying my work and keeping my sanity. His guidance and countless hours of mentoring have been foundational to my success at the University of California, Riverside (UCR) and has undoubtedly helped shape my career aspirations.

Much of the work in this dissertation has already been published and I thank the Institute of Electrical and Electronics Engineers (IEEE) and the Association for Computing Machinery (ACM) for allowing me to include significant portions of my work in this dissertation from the following published works:

- ACM Journal on Emerging Technologies in Computing Systems [28] (**CHAPTER 2**)
- ACM Proceedings of the Great Lake Symposium on VLSI [30] (**CHAPTER 3**)
- IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems [31] (**CHAPTER 3**)
- ACM Proceedings of the Design Automation Conference [29] (**CHAPTER 4**)

I would like to thank the following entities for their generous awards, fellowships and stipends which made it possible for me to focus on research during my time at UCR:

- The National Science Foundation (NSF) for the Graduate Research Fellowship
- UCR for their initial fellowship aid and Dissertation Year Fellowship

DEDICATION

This dissertation is dedicated to:

- Jesus Christ, my personal savior, who gave me the idea and strength to become a graduate student
- Maritza Grissom, my beautiful wife, who has learned to say “duh” when I talk about microfluidics so it feels like it should all actually make sense to everyone listening
- William & Sandra Grissom, my amazing parents, who paid my way through undergrad and have loved and supported me through it all
- Joanna, Michael, Rebecca, Jonathan & Jeremiah, my older siblings, who have always been there for me
- The orphans, widows and all the beautiful people of Sub-Saharan Africa, who have been my inspiration for advancing the field of microfluidics

ABSTRACT OF THE DISSERTATION

Design of Topologies for Interpreting Assays on Digital Microfluidic Biochips

by

Daniel Thomas Grissom

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, June 2014
Dr. Philip Brisk, Chairperson

In the last decade, digital microfluidic biochips have emerged as a viable candidate for the automation and miniaturization of biochemistry; however, digital microfluidic designs in previous works typically suffer from two short-comings: 1.) they are unable to respond to live feedback and errors; 2.) they are application-specific, rather than programmable. In the early years of digital microfluidic research, the synthesis problems of scheduling, placement and routing were performed offline (before runtime) due to their algorithmic complexity, typically yielding application-specific devices that could perform only one type of biochemical reaction and could not respond to live feedback in a timely manner, due to the complexity of their algorithms.

This dissertation offers topological solutions toward realizing digital microfluidic biochips that are both dynamic and programmable in nature. We begin by presenting interpretation, which is a form of dynamic synthesis. Instead of static compilation which generates a deterministic electrode activation sequence, interpretation acts like an operating system that manages resources as an assay is being executed, allowing for resources to be allocated and dispatched dynamically in response to live feedback from integrated sensors and video monitoring. We also present new language constructs necessary to incorporate control flow into digital microfluidic biochips (DMFBs). We then introduce virtual topologies, a virtual organization of electrodes into "city streets and blocks," which help simplify dynamic synthesis flow algorithms; we show two new virtual topologies and describe scheduling, placement and routing algorithms to accompany them, yielding fast, reliable, dynamic, programmable DMFBs. Finally, we present a field-programmable, pin-constrained (FPPC), topology which, for the first time, offers a solution to reduce the cost of a DMFB while maintaining programmability. We include results which show our FPPC design to be the least expensive when compared to prior pin-constrained and direct addressing DMFBs, while offering unmatched flexibility next to its closest competitors in price. We conclude with the first detailed cost analysis and shed light on the relationship between PCB layer count, pin count and cost. Our results reveal that the minimization of pin-count, if not done carefully, can necessitate additional PCB layers and yield a more expensive DMFB.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF APPENDIX FIGURES

LIST OF TABLES

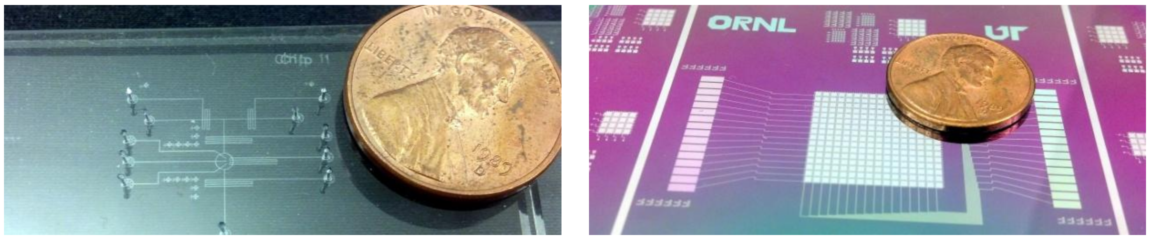
CHAPTER 1 INTRODUCTION

Over the last decade, microfluidics has emerged as a viable technology for automating and miniaturizing biochemical reactions. Instead of mixing fluids together on the order of milliliters and liters in test tubes and beakers, microfluidic devices can perform many of the same reactions by manipulating nanoliter-sized quantities of fluid on a small lab-on-chip (LoC) device. Microfluidic LoCs have been designed to execute a multitude of different biochemical applications including in-vitro diagnostics and immunoassays used in clinical pathology [75], DNA polymerase chain reaction (PCR) mixing stages used to amplify DNA [50] and protein crystallizations [81].

Figure 1-1 shows the two major types of microfluidic LoCs. *Continuous-flow microfluidic devices* (**Figure 1-1(a)**), also called *analog microfluidic biochips*, perform biochemical reactions, known as *assays*, by actuating pumps to open and close micro-valves [57]. On these devices, fluid flows through tiny channels in a continuous fashion similar to how water flows through the pipes in a house. In contrast, a newer LoC platform known as a digital microfluidic biochip (DMFB) performs assays by manipulating discrete droplets of fluid around a 2D-array of electrodes (see **Figure 1-1(b)**).

Although continuous-flow microfluidic devices are more mature and offer a solid platform for executing microfluidic assays, their micro-channel and micro-valve locations are permanently etched into the device, as seen in **Figure 1-1(a)**, which limits the LoC's capabilities since fluids can only be mixed and transported via the pre-planned layout.

Typically, continuous-flow devices are designed and fabricated to perform a single assay. DMFBs, however, offer a generic 2D-array of electrodes on which various microfluidic operations can be performed with much less restriction (see **‘Section 1.1 - DMFB ’**). Since DMFBs typically have no permanent regions for mixing and transportation, they are inherently reconfigurable in nature, which make them an excellent candidate for a general-purpose microfluidic device that can perform a wide assortment of assays.

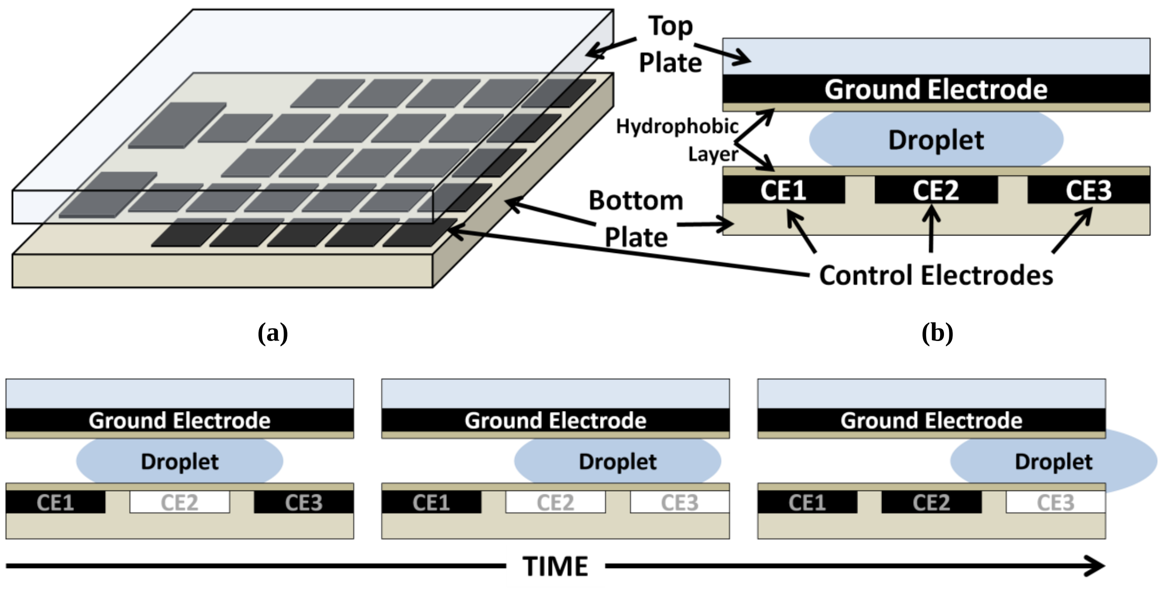


(a) **(b)**

Figure 1-1: (a) A continuous-flow microfluidic device operates on continuous flows of fluid that flow through micro-channels; (b) A digital microfluidic biochip (DMFB) operates on discrete droplets that move around a 2D-array of square electrodes.

1.1 - DMFB Droplet Manipulation

DMFBs execute assays by manipulating nanoliter-sized droplets of fluid on a 2D-array of electrodes, as seen in **Figure 1-2(a),** and are typically based on a phenomenon known as electrowetting on dielectric (EWOD) [63]. An EWOD-based DMFB, as seen in **Figure 1-2(b)**, consists of a top and bottom plate coated with a hydrophobic layer (to prevent fluids from sticking to the DMFB). The bottom plate has an array of droplet-sized control electrodes, while the top plate has a single conducting electrode that spans the entire array (**Figure 1-2(a)**) of control electrodes. Each droplet is sandwiched between the bottom and top plates and will hold its place if its underlying electrode remains activated.



(c)

Figure 1-2: (a) A DMFB is a planar array of electrodes; (b) Cross-sectional view of electrode array; (c) A droplet is transported from control electrode 2 (CE2) to CE3 by activating (white) CE3, while deactivating (black) CE2, allowing for droplets to be transported around the DMFB.

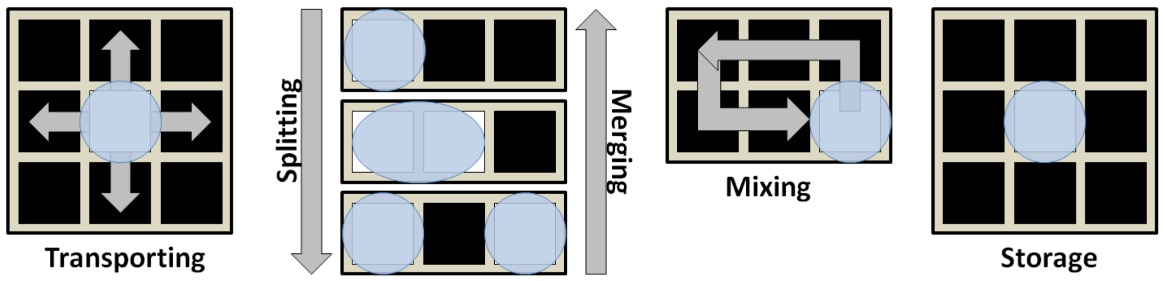


Figure 1-3: Basic microfluidic operations form the building blocks for assays to be executed.

In **Figure 1-2(b)**, a droplet is seen to overlap neighboring electrodes; when a neighboring electrode is activated, electrowetting causes an electric field change, inducing increased wetting and surface tension on that side of the droplet, which causes it to flow toward the newly activated electrodes [63]. Thus, **Figure 1-2(c)** shows that if CE3 is activated while CE2 is being deactivated, the entire droplet will move to cover CE3. As seen in **Figure 1-3**, with the proper sequence of electrode activations, several fundamental microfluidic operations can be performed: droplet transportation, splitting, merging, mixing and storage. Sensor-based detection operations execute by moving a droplet to a detector (placed above an electrode) and storing the droplet there; among the most popular sensors beginning to be employed in microfluidic systems, optical sensors can be used to measure wavelength/color (e.g., a certain reaction may be considered complete when the solution turns a certain wavelength/color) [49][65][71], while capacitive sensors can be used to measure the volume and validate the presence of a droplet [64]. Dispense and output operations are performed by I/O reservoirs on the perimeter of the DMFB.

If a droplet is not centered over or adjacent to any activated electrodes, it will *drift* across the DMFB in an undetermined and unpredictable manner. In contrast, activating an electrode underneath a droplet will hold it in place as long as the electrode remains activated, inducing the storage operation seen in **Figure 1-3**.

1.2 - DMFB DEVICE TECHNOLOGY OVERVIEW

There are several classes of DMFBs that provide varying levels of droplet control. Typical *direct-addressing* (*individually-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 [82].

Cross-referencing DMFBs use $(M + N)$ control pins to control an $(M \times N)$ array of electrodes [23]. 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 [80]. 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 [80].

Active-matrix addressing designs are emerging which give independent control of $(M \times N)$ electrodes while using only $(M + N)$ control pins [60]. 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; however, the fabrication process of these devices is difficult and has not yet been mastered.

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 [80]. Individually-addressable and active-matrix DMFBs provide the most programmability and flexibility, in terms of droplet control; however, individually-addressable designs are expensive and the active-matrix fabrication process is not stable enough to yield reliable devices.

1.3 - HIGH-LEVEL ASSAY SYNTHESIS OVERVIEW

A digital microfluidic system typically consists of three parts, as seen in **Figure 1-4**: a PC controller, a micro-controller and a "wet" DMFB which contains the droplets where the microfluidic reaction is performed. The PC controller performs the computations which map a microfluidic reaction to a DMFB; this process is known as *synthesis* and yields an electrode activation sequence (see **'Section 1.3.3 - Droplet Routing'**) which is sent to the microcontroller. The microcontroller and its accompanying circuitry amplifies and "plays" this sequence of signals to the DMFB to activate the DMFB's electrodes, in turn performing the microfluidic reaction. Today, modern DMFBs have optical and capacitive sensors which can communicate back to the PC and microcontroller to help ensure proper execution of the microfluidic reaction and provide feedback for error recovery.

Figure 1-5 details the general synthesis flow used for DMFBs. An assay can be represented as a directed acyclic graph (DAG), where the nodes represent microfluidic operations (e.g. mix, split, etc.) and the edges represent precedence (i.e., a partial order of operations). The left side of **Figure 1-5** depicts a simple assay in which two droplets are input, mixed, and finally, output from the DMFB.

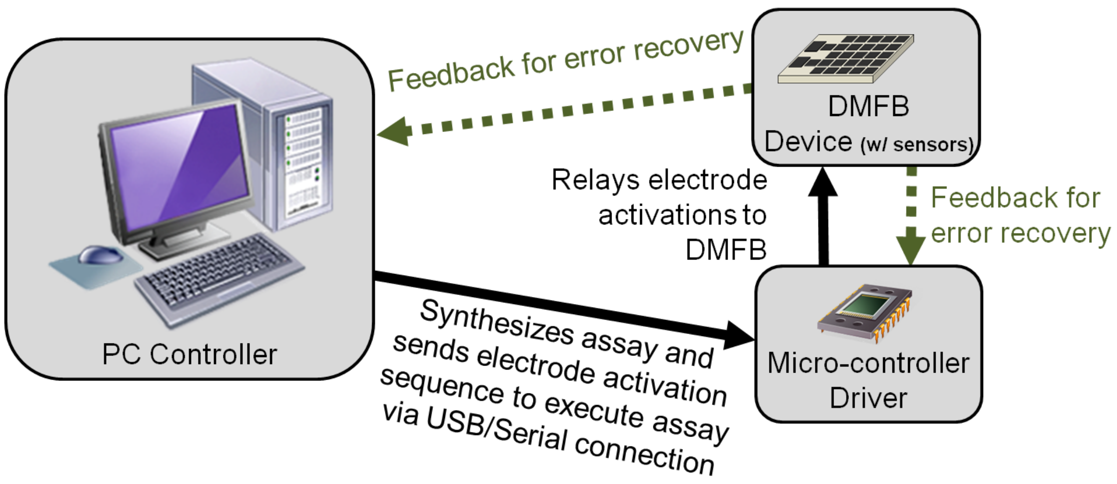

Feedback for error recovery
DMFB
Device (w/ sensors)
Relays electrode activations to DMFB
Feedback for error recovery
Micro-controller Driver
PC Controller
Synthesizes assay and sends electrode activation sequence to execute assay via USB/Serial connection

Figure 1-4: A microfluidic system typically consists of three major parts: a PC controller, a microcontroller and a DMFB.

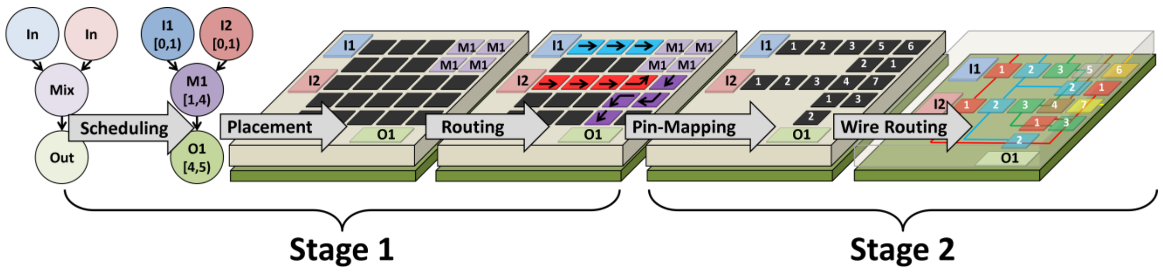


Figure 1-5: A typical microfluidic synthesis flow dictates that a microfluidic assay is represented in the form of a DAG; in Stage 1, its operations are scheduled and placed onto the DMFB array and droplets are routed between operation locations. In Stage 2, pin-mapping and wire routing are performed to eliminate unused electrodes and connect the electrodes to an external edge of the device to be driven by a microcontroller.

1.3.1 - SCHEDULING

The first step of synthesis is scheduling. In this step the scheduler assigns an absolute start and stop time to each operation (e.g., the mix operation, *M1*, will execute from time 1 to 4). The scheduler must ensure that no operation starts before any of its parent operations end and that there are enough resources to simultaneously perform any concurrently scheduled operations (e.g., the scheduler must guarantee that two detection operations are not scheduled at the same time if there is only one available detection module on the DMFB).

1.3.2 - Placement

Once a schedule has been computed, placement is performed. The placer decides where on the DMFB to perform each operation. For input operations, an input reservoir containing the appropriate fluid is selected. However, operations such as mixing and splitting can be performed at a variety of different locations on the DMFB. For example, in **Figure 1-5**, the mix operation *M1* is placed in the 2×2 array of cells in the top-right corner of the DMFB. However, *M1* could be placed in any unoccupied 2×2 array of cells on the DMFB. This array of cells used to denote the location of an operation is temporary and is known as a *module*. As seen in **Figure 1-6**, a mix operation may be placed into a variety of different module sizes, although the time it takes to complete the mix operation may vary based on the module size. This allows the placer flexibility when placing operations. It should be noted, however, that if the placer changes the module size, the schedule may need to be recomputed because of the new module's required change in operation length.

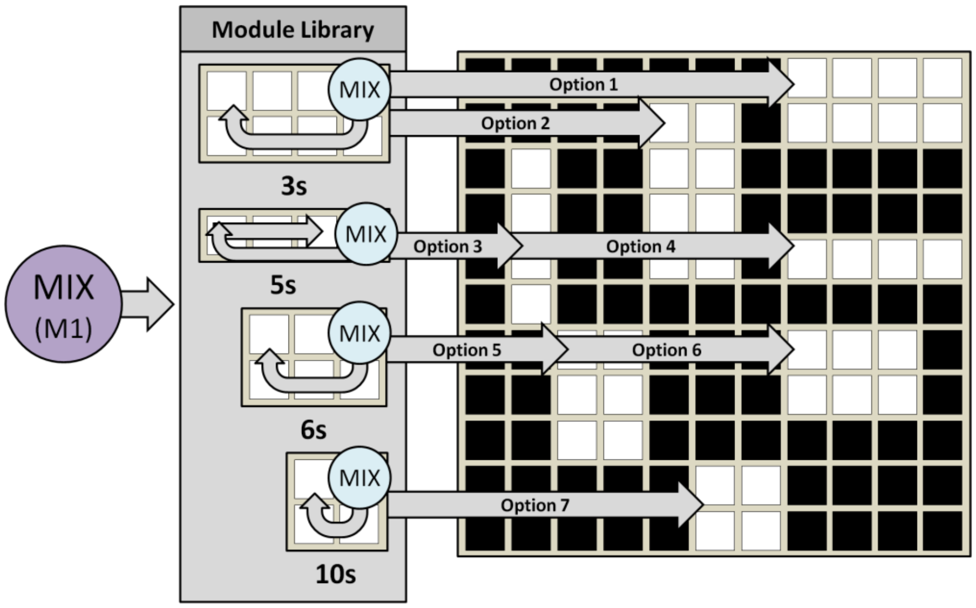
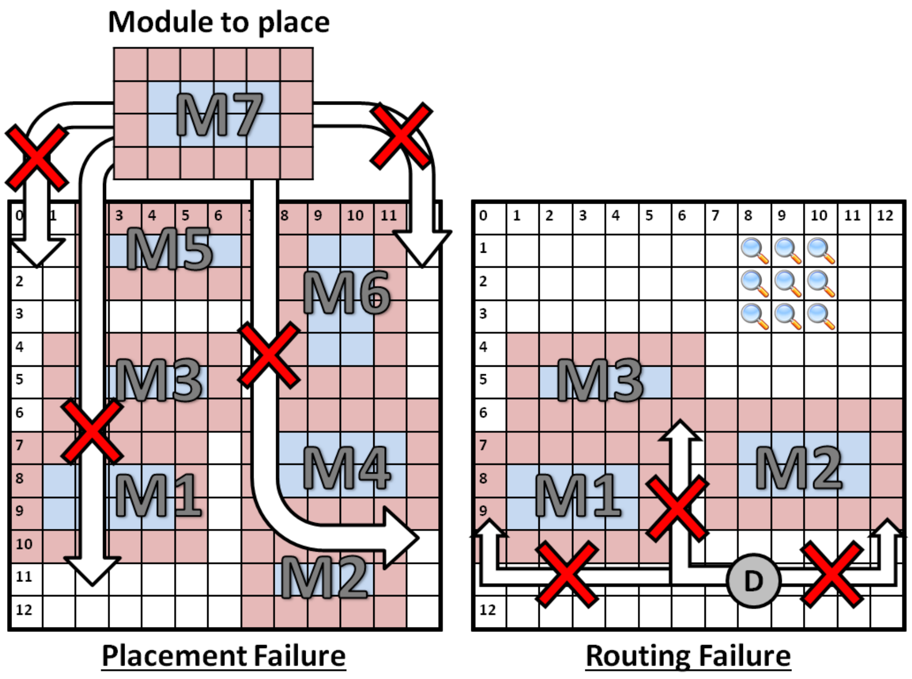


Figure 1-6: An operation, such as the mix operation from Figure 1-5 (M1), leverages a module library to select its duration and module shape (7 distinct horizontal/vertical orientation options shown given the 4 module shapes) and is then placed anywhere onto an empty DMFB.

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., **Figure 1-7(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 [76]. In addition, module placement should leave enough space around modules to prevent droplet routing failure; in the case of a poor placement, a droplet may not be able to reach its destination because every possible path is being blocked by an existing module (as seen in **Figure 1-7(b)**), causing the assay to fail.



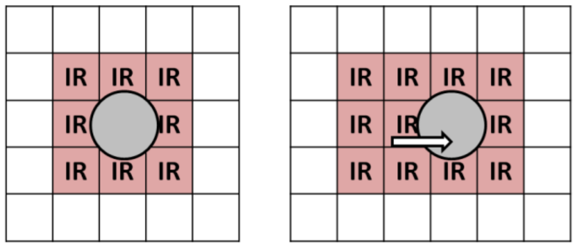
(a) (b)

Figure 1-7: (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.

1.3.3 - Droplet Routing

Next, once operations have been scheduled and their corresponding modules have been placed onto the DMFB, droplet routing is performed to compute paths between the modules where each operation is performed. Operations can be routed from input reservoirs to modules, from modules to modules, and from modules to output reservoirs. Referring back to the scheduled DAG in **Figure 1-5**, the edges also represent the points in the assay when a droplet may need to be routed to a new location on the DMFB.

The router must ensure that a droplet reaches its destination and that it does not collide or interfere with any other droplets currently being routed or performing an operation (e.g., a newly routed droplet could interfere with an ongoing mix). To avoid droplet interference, a set of simple rules is followed. An *interference region* is defined as the cells directly adjacent to a droplet, as seen in **Figure 1-8(a)**. A *droplet actuation cycle* (or *cycle*) is the discrete time-step that it takes for a droplet to move from the center of one electrode to the center of an adjacent electrode; the cycle length is determined by the electrode size, applied voltage and fluidic properties, and thus, can differ from system to system. At the beginning of each cycle, an interference region is set around each droplet at its current location. As a droplet moves, its interference region expands to include the entire region around the two cells the droplet occupied during that cycle (see **Figure 1-8(b)**). No droplet can move into a cell containing another droplet or into the interference region of any other droplet at any time, unless those droplets are part of the same merge or split operation. This set of rules keeps droplets safe by prohibiting any other droplets to move into potentially adjacent cells at any time during a cycle.



(a) **(b)**

Figure 1-8: (a) The interference region ('IR') of a droplet at the beginning of a cycle represents its static constraints; (b) The interference region at the end of a cycle demonstrate its dynamic constraints.

$$\left|X_i(c+1) - X_j(c+1) \geq 2\right| \quad and \quad \left|Y_i(c+1) - Y_j(c+1) \geq 2\right| \qquad (1.1)$$

$$\left|X_i(c+1) - X_j(c) \geq 2\right| \quad and \quad \left|Y_i(c+1) - Y_j(c) \geq 2\right| \qquad (1.2)$$

$$\left|X_i(c) - X_j(c+1) \geq 2\right| \quad and \quad \left|Y_i(c) - Y_j(c+1) \geq 2\right| \qquad (1.3)$$

For the sake of completeness, a formal description of the fluidic constraints, as detailed in ref. [78], is included in **Equations 1.1-1.3**. Let n be the number of droplets in the system, and $0 \leq i, j < n$. Then, the constraints seen in **Equations 1.1-1.3** must hold for all pairs of droplets D_i and D_j, $i \neq j$. The 2D microfluidic array is represented by the coordinates (X, Y), where $(X_i(c), Y_i(c))$ is used to represent the location of droplet D_i at the beginning of cycle c. We assume that no droplet D_i is initialized inside the interference region of any other droplet D_j. **Equation 1.1** ensures that the locations of D_i and D_j are not adjacent to each other at the end of cycle c. **Equation 1.2** and **Equation 1.3** guarantee that D_i and D_j never enter each other's interference regions (the extended region seen in **Figure 1-8(b)**) at any time during cycle c.

The output of the droplet router is a list of electrodes to activate each cycle; a *cycle* is the time it takes to move a droplet from one electrode to the next. In **Figure 1-9**, a "dry" controller (e.g., a PC and/or microcontroller) sends signals to activate electrodes during each cycle on the "wet" DMFB.

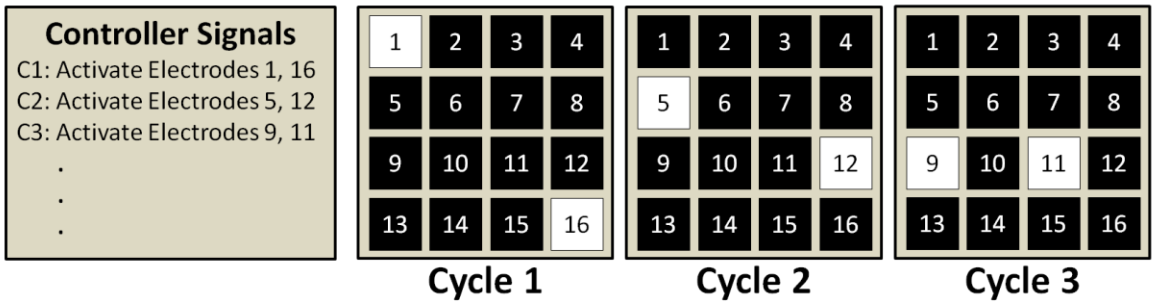

Controller Signals
C1: Activate Electrodes 1, 16
C2: Activate Electrodes 5, 12
C3: Activate Electrodes 9, 11
.
.
.
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
Cycle 1
Cycle 2
Cycle 3

Figure 1-9: The droplet router produces an electrode activation sequence, driven by a microcontroller (left), which corresponds to droplet movement on the DMFB.

1.3.4 - PIN-MAPPING

Once droplet routing is complete, an optional step called pin-mapping can be performed when DMFB manufacturers want to reduce the cost of the device. As mentioned in **'Section 1.2 - DMFB Device Technology Overview'**, in an individually addressable DMFB, each electrode is wired to an external electrical pin on the edge of the DMFB; in turn, each pin is connected to and driven by the microcontroller such that each electrode can be independently controlled (see **Figure 1-10(a)**). Individually addressable DMFBs offer the most flexibility in terms of droplet coordination; however, they are expensive to fabricate because the number and complexity of wire routes that must exist to connect each electrode to an external pin on the DMFB can require an increasing number of printed circuit board (PCB) layers (**Figure 1-11(a)**), which significantly adds to the cost of a DMFB.

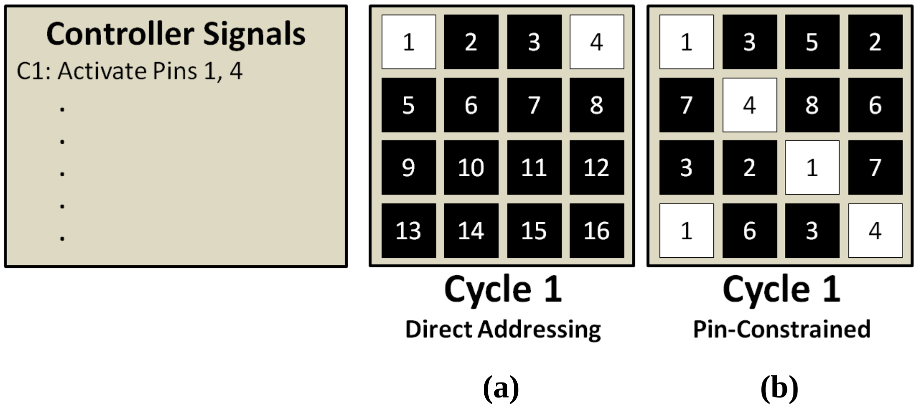
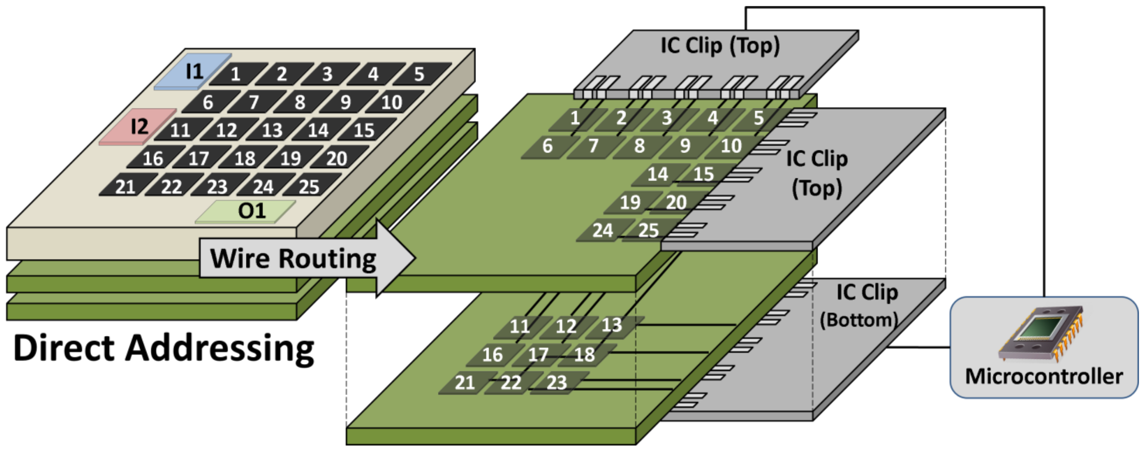
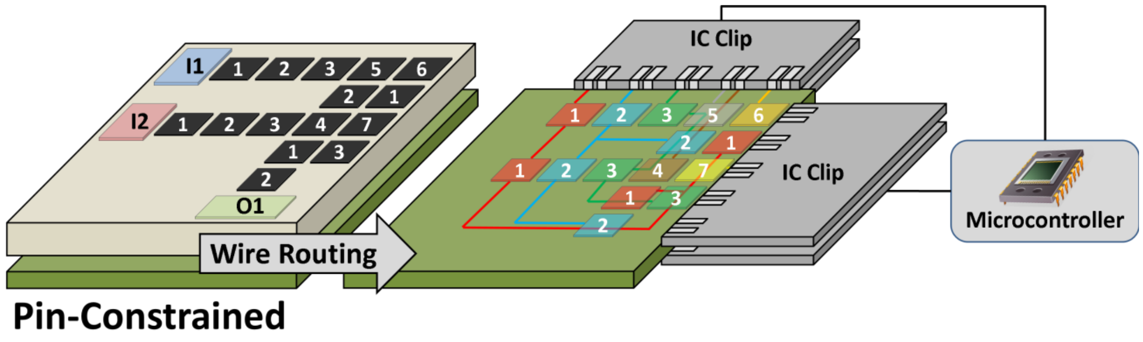


Figure 1-10: Activating a pin on a (a) direct-addressing DMFB activates (white) exactly 1 electrode per pin; (b) a pin on a pin-constrained DMFB activates 1+ electrodes per pin, depending on the pin layout.



(a)



(b)

Figure 1-11: A DMFB has printed circuit boards (PCBs, green layers) underneath the substrate containing the control electrodes that serve as the medium for wire-routing. A microcontroller sends signals to and interfaces with the DMFB via one or more integrated circuit (IC) clips. (a) A direct addressing DMFB, may require many PCB layers, while (b) a pin-constrained DMFB is designed to perform wire routing with fewer PCB layers.

Pin-constrained DMFBs employ pin-mapping techniques to reduce the wire-routing complexity by connecting multiple electrodes together in such a way that activating a single external pin (via a single signal from the microcontroller) can activate multiple electrodes (**Figure 1-10 (b)**). In **Figure 1-5**, the pin-mapping step removes the 10 non-used electrodes and connects the remaining electrodes in such a way that reduces the number of pins from 15 to 7.

The scheduling, placement and routing steps of synthesis are performed as on a direct addressing DMFB for a particular assay (as seen in Stage 1 of **Figure 1-5**); then the resultant electrode activation sequence is examined to compute a pin-mapping that will successfully execute the synthesized assay using fewer pins [82]. The drawback of this approach is that the DMFB is now physically tailored to execute a single assay.

1.3.5 - WIRE ROUTING

The last stage of synthesis is wire routing. The control electrodes reside on the lower substrate of the DMFB (seen in **Figure 1-2(b)**), while the wire-routing is performed below this substrate on one or more printed circuit board (PCB) layers (the green layers in **Figure 1-5** and **Figure 1-11**). In an individually addressable DMFB, this stage computes wire-routes to connect each electrode to its own pin on the peripheral of the DMFB, allowing the DMFB to be controlled by the microcontroller signals (**Figure 1-11(a)**).

In a pin-constrained DMFB, the wire router must first connect all of the electrodes together with the same pin-number. For example, in **Figure 1-11(b)**, all the electrodes with a "1" must be connected together (done so with the red wires), all the electrodes with a "2" must be connected with each other (blue), and so on and so forth. Each network of wires connecting a group of electrodes with the same pin-number is known as a *net*. Once the nets are connected together, a wire must be routed to connect each net to its own pin on the peripheral of the DMFB. Because of the reduction of electrodes and combination of pin signals, pin-constrained devices typically have fewer and smaller PCB layers, making them much more cost-effective than direct addressing DMFBs.

1.4 - DMFB Design Objectives

This dissertation addresses two major objectives and goals in the design of DMFBs:

1. DMFBs that are dynamic in nature
2. DMFBs that are programmable

The following sub-sections provide a brief background and explain why these objectives are important and why they are not currently being addressed.

1.4.1 - Dynamic DMFBs

High-level synthesis for DMFBs has been studied for over a decade and began by employing a static compilation flow [73]. In a static compilation flow, scheduling, placement and routing are performed in their entirety and the output is compiled into a deterministic electrode activation sequence, as described in **'Sections 1.3.1 - Scheduling'** and **'Section 1.3.3 - Droplet Routing'**; however, a number of errors can occur during the execution of an assay, such as:

- A droplet might not move as intended from one electrode to the next
- A mix operation might not reach a uniform concentration in its pre-allotted time
- A droplet might not split into two droplets of perfectly equal size, as expected

The deterministic nature of static compilation assumes that there are no errors during assay execution; thus, without live feedback from the DMFB, it is very difficult to know if an assay run by a deterministic state machine (i.e., static compilation) has executed properly since there is no way to detect and respond to an error.

Figure 1-13(a) shows the basic order of a static synthesis flow (i.e., static compilation) and that scheduling, placement and routing are performed exactly once; the resultant electrode activation sequence attempts to execute the assay in its entirety. In fact, static synthesis is typically performed for a particular assay during the DMFB design process and the electrode activation sequence is compiled as a bit-vector (containing the *On*/*Off* signals for each electrode at each cycle) onto the DMFB long before an end-user ever receives the final biochip; the resultant electrode activation sequence mimics droplet movement and is the driver for droplet actuation, as seen in **Figure 1-12**. Since an assay is compiled once during the design process, complex, long-running algorithms (e.g., it could take hours to compute synthesis steps) are typically used to yield highly-optimized results (e.g., shortest overall execution time; fewest number of electrodes utilized) [73][75][76].

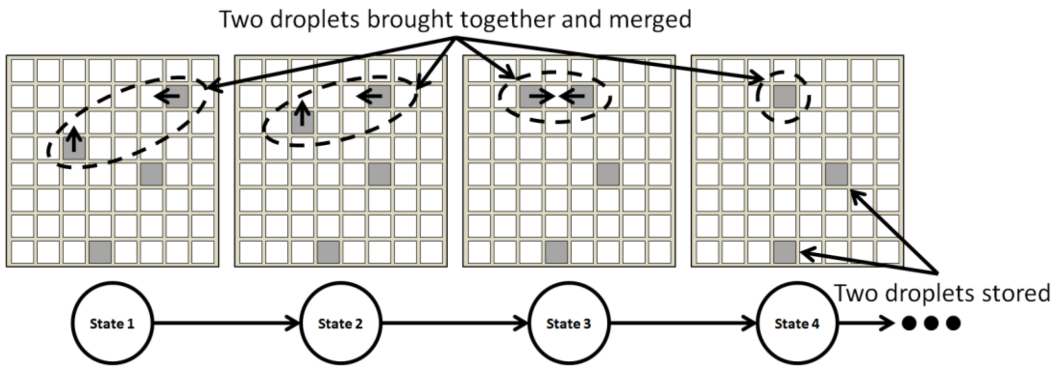
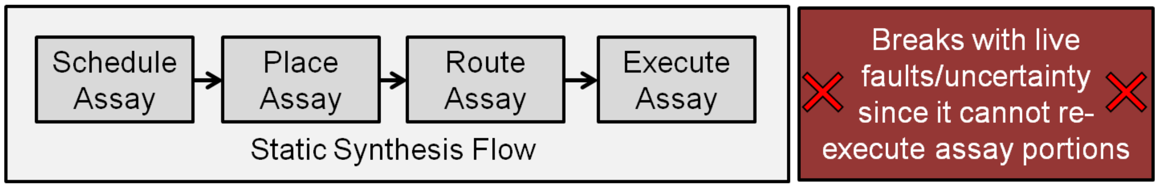
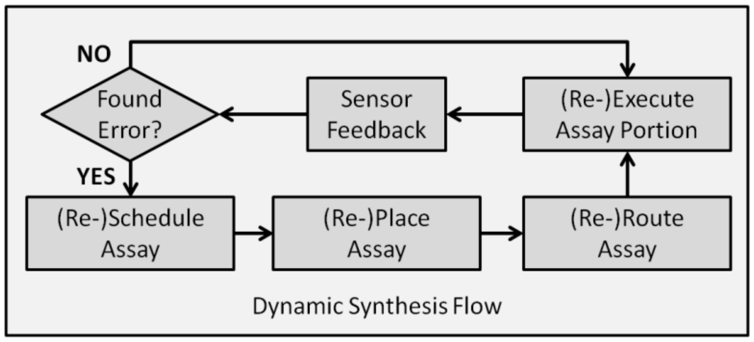


Figure 1-12: Example of the linear state machine model of DMFB control. The output of each state is the subset of electrodes in the DMFB that will be activated during each time step (shown in gray). The state machine is timed, based on the activation frequency, typically 100Hz. In this example, two droplets are transported to a common location so that they can be merged, and two droplets are stored in place.

Today's DMFBs can now integrate optical and capacitive sensors for detecting live errors during an assay's execution [21][64][71]. Although many of these sensors were developed during the same time-frame as the initial DMFB synthesis publications [65][73][75], synthesis works have not attempted to dynamically integrate live feedback from sensors until more recently [6][7][36][52][53][54]. A dynamic synthesis flow, as seen in **Figure 1-13(b)**, may perform an initial synthesis for an entire assay, but executes smaller portions in between sensor-feedback checkpoints. A dynamic synthesis flow can continue executing the assay as originally synthesized until an error is detected, at which point it will re-synthesize the assay to account for the unexpected events.


Schedule Assay
Place Assay
Route Assay
Execute Assay
Static Synthesis Flow
Breaks with live faults/uncertainty since it cannot re-execute assay portions

(a)



(b)

Figure 1-13: (a) A static synthesis flow performs scheduling, placement and routing exactly once, and will break in the presence of errors since it cannot adjust to live feedback; (b) a dynamic synthesis flow computes an initial schedule, placement and routing, but then executes portions of the assay while checking sensor feedback. If an error is found, portions of the assay can be re-synthesized.

Error-recovery is not the only reason for a dynamic synthesis flow. In addition, live-sensor feedback allows dynamic decisions to be made about intermediate results. For example, given the completion of a clinical diagnostic on a DMFB, one might want to continue by running a particular sub-assay based on whether the initial result was positive or negative. Thus, dynamic synthesis also allows the introduction of control flow into DMFB research.

1.4.1.1 - Barriers to Dynamic DMFBs

There are two major problems with current synthesis methods that make them poor dynamic synthesis algorithm candidates for DMFBs:

1. Current synthesis algorithms take too long
2. Current synthesis algorithms do not guarantee legal solutions

As mentioned in the **'Section 1.4.1 - Dynamic DMFBs'**, static synthesis methods typically employ long-running algorithms that can take hours or days to yield results. In contrast, many assays only take seconds or minutes to perform in their entirety; thus, in a dynamic environment where synthesis methods are called to re-map an assay in response to runtime errors or control flow, it is not feasible to wait hours or even minutes to re-synthesize portions of the assay. Synthesis methods are needed that can dynamically interpret the assay (in response to live feedback) in milliseconds (or less) such that the overall execution time of the assay is not impacted in any significant way.

In contrast to static offline compilers, which synthesize assays as deterministic state-machines, a *dynamic online interpreter* will act more like a virtual machine which manages the DMFB's resources and interprets assays on-the-fly. **Figure 1-14** 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 [52].

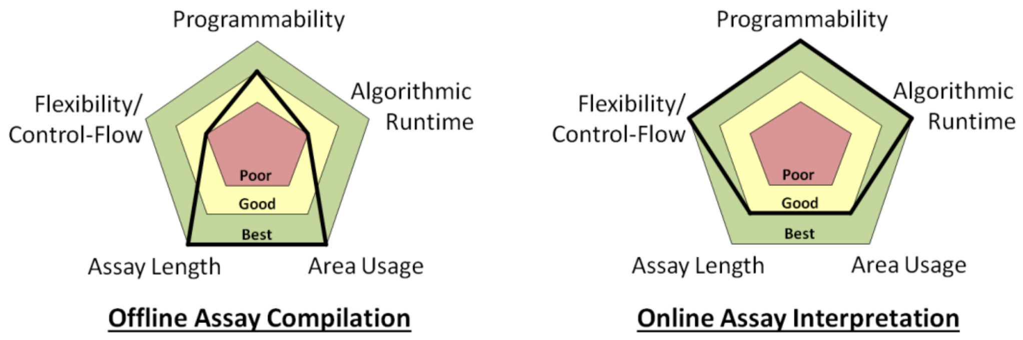


Figure 1-14: Offline vs. online synthesis tradeoffs.

Secondly, current synthesis algorithms do not guarantee legal solutions. For example, a droplet routing algorithm has the task to compute routes from any source to any destination, as dictated by the placement stage; however, all previous droplet routing algorithms found cases that proved to be unroutable (no path existed from source to destination) or yielded deadlock (i.e., so much congestion existed that droplets would get stuck in an irresolvable “traffic jam” and be unable to reach their destinations) [11][16][26][88]. In these cases, it may be possible to re-perform the scheduling and placement stages in an attempt to provide routable problems; however, these operations are computationally expensive in the context of dynamic synthesis and can cause the end-user to wait for significant periods of time, while his or her assay is being executed, in an attempt to find a new synthesis solution. *Thus, it is imperative that dynamic synthesis methods guarantee a valid solution on the first attempt to successfully execute the assay to completion.*

1.4.2 - Programmable DMFBs

It is crucial that DMFBs remain programmable. In an effort to optimize and reduce the cost of DMFBs (see **'Section 1.3.4 - Pin-Mapping** and **'Section 1.3.5 - Wire Routing'**), many papers have been published in the microfluidics community detailing algorithms to design assay-specific DMFBs [39][40][41][50][51][82][86]. Although there is a valuable place for this type of research, this means that only a few, select applications will make their way to DMFB technologies, as seen fit and profitable by the handful of DMFB vendors currently in existence. In contrast, we believe that the continued development of programmable DMFBs could enable researchers around to globe to specify and execute their own experimental assays, taking advantage of real-time sensory feedback to perform biochemical reactions with greater accuracy and speed than ever. Ultimately, programmable DMFBs will ease and encourage the development of new applications, allowing this emerging technology the exposure and traction that it needs to fully catch on in the scientific community and general populous.

Just as the iPhone is made more useful by its robust app market, we believe that DMFBs will become ubiquitous as independent researchers are able to contribute their own DMFB assay protocols to the scientific community for fast, concise, and reliable reproduction of scientific experiments and clinical tests. Imagine an assay is developed to detect a new strain of flu. Instead of having to develop a unique DMFB hardware solution, an end-user (presumably a physician's office, but, perhaps, someday a patient him/herself) can simply download and install the new assay protocol onto their low-cost, programmable DMFB and purchase a "fluids kit" that has the necessary reagents to perform the assay (see **Figure 1-15**). This overall usage flow promises to be much more efficient concerning both time and money.

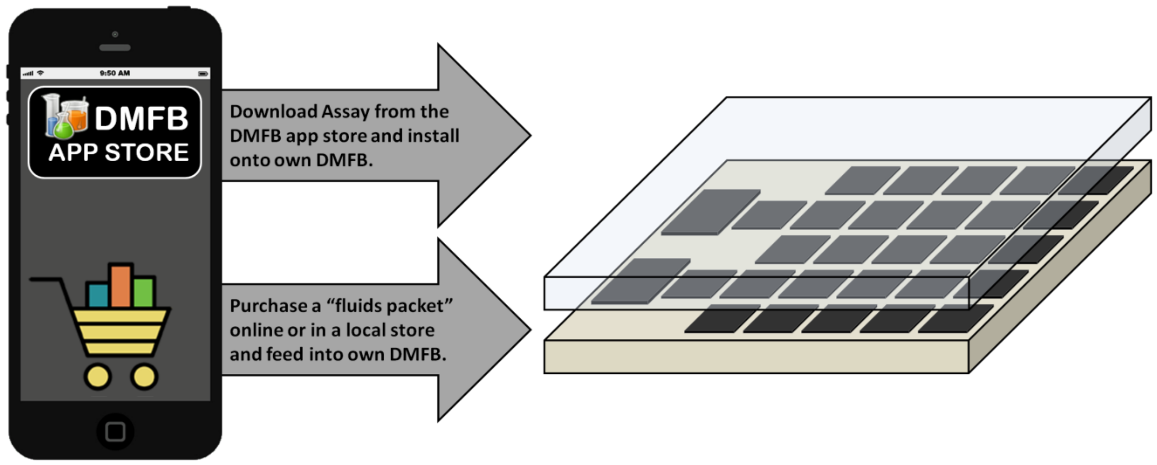

9:50 AM
DMFB
APP STORE
Download Assay from the DMFB app store and install onto own DMFB.
Purchase a "fluids packet" online or in a local store and feed into own DMFB.

Figure 1-15: A DMFB "App Store" would allow a user to download an assay specification to run on their own DMFB; separate fluid kits could be developed and purchased, as well.

1.4.2.1 - Barriers to Programmable DMFBs

As mentioned in **'Section 1.3.4 - Pin-Mapping'**, **'Section 1.3.5 - Wire Routing'**, and **'Section 1.4.2 - Programmable DMFBs'**, many synthesis flows map assays to highly-optimized pin-constrained DMFBs that, although cost effective, are severely limited in application. Typically, these DMFBs are designed to perform a small handful of assays (e.g., 3), at most, and are not able to perform much else. Work exists which takes an already-existing DMFB design and attempts to map two independent droplet movements onto it to obtain a general-purpose DMFB [51]; although this may lead to some limited opportunities for error-recovery and very basic assays, it does not allow for more-complex, generic assays to be synthesized on low-cost DMFBs. *Thus, it is imperative that synthesis algorithms and DMFBs are designed to not only be inexpensive, but also to perform any sequence of basic operations, instead of specific assays.*

1.5 - CONTRIBUTIONS

This dissertation details significant contributions to the primary design objectives, detailed in **‘Section 1.4 - DMFB Design Objectives’**, of achieving programmable and dynamic DMFBs. In **CHAPTER 2**, we present interpretation, which is a form of dynamic synthesis. Instead of static compilation which generates a deterministic electrode activation sequence, interpretation acts like an operating system that manages resources as an assay is being executed, allowing for resources to be allocated and dispatched dynamically in response to live-feedback. We also provide new language constructs necessary to incorporate control flow into DMFBs. **CHAPTER 3** presents the idea of a virtual topology, which is an abstract organization of electrodes into “city streets and blocks,” which help simplify dynamic synthesis flow algorithms; we introduce several variations of a virtual topology and detail scheduling, placement and routing algorithms that accompany them to yield fast, reliable, dynamic, programmable DMFBs. **CHAPTER 4** details a pin-constrained physical topology which offers a solution to reduce the cost of a DMFB while maintaining programmability. Our design is pin-constrained, and thus, very inexpensive, but also designed to perform basic microfluidic operations, making it general-purpose in nature. In this chapter, we provide the first detailed cost analysis for DMFB PCB layers and shed light on the relationship between, PCB layer count, I/O pin count and cost. Finally, **CHAPTER 5** concludes by summarizing the general findings of this dissertation.

CHAPTER 2 INTERPRETATION

2.1 - INTRODUCTION

At present, LoC programming is either done at the machine level (i.e., manually choosing a sequence of actuation signals to send to the device over time), or is highly restricted, e.g., to assays that can be represented as DAGs without control flow and without the ability to take action based on feedback provided by the device. This chapter introduces a software interpreter that performs online execution of assays featuring control flow, allowing them to be scheduled and executed immediately in response to live, sensor-based feedback. To allow the programmer to express control flow operations, language extensions to a high-level biochemical programming language are introduced as well. The long-term objective of this research is to open the door for new microfluidic capabilities and applications.

Figure 2-1 motivates the need for control-flow and online interpretation with a drug discovery application. The assay performs a test (Test 1), detects the result, and then automatically responds by determining that it has found a valid solution, or continues exploring the solution space by executing new assays (Test 2 and 3) with varying concentrations. **Figure 2-1** shows the first few tests, but this procedure could be extended and repeated hundreds or thousands of times, adjusting various parameters along the way, until a valid solution is found.

Without control-flow, this application is intractable for all but the smallest examples because the designer must create a single DAG offline that describes and handles each possible path through the application [6]. Instead, an online interpreter could leverage control-flow to instantly schedule and dispatch new assays (in **Figure 2-1**, the boxes labeled Test 1-3) upon detection of any terminating, dependent assays.

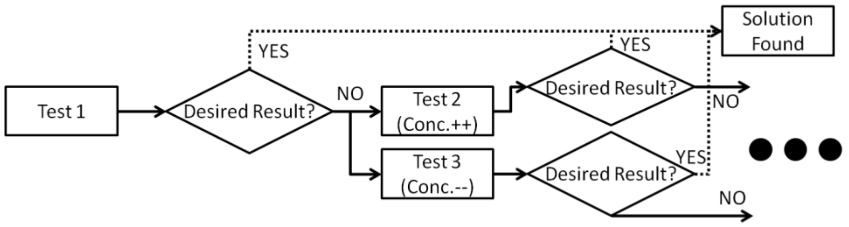


Figure 2-1: A control-flow graph for a simple drug-discovery assay that increases (++) or decreases (--) concentrations based on the results of previous tests.

Although synthesis has been performed entirely offline up to this point, Ho, Chakrabarty and Pop suggest that online systems are forthcoming with the development of "specialized heuristics" which can perform synthesis in milliseconds [36]; Luo, Chakrabarty, and Ho [52] 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 re-synthesized 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.

2.1.1 - INTERPRETATION VS. COMPILATION

Historically, assays have been specified as DAGs, without control flow. A typical compilation sequence is shown in **Figure 1-5**. The DAG is first scheduled [20][33][61][65][75]; dimensions for each operational module are selected [77][83]; scheduled operations are then placed onto the 2D grid, ensuring that no concurrently executing operations overlap to prevent interference [45][76][87]; lastly, non-interfering droplet routes are computed to deliver droplets to the appropriate DMFB locations at appropriate times [11][16][38][48][66][67][69][78][88]; in some cases, all of these problems can be solved in conjunction with one another [56].

The control program that is generated by the compiler is a linear state machine, as shown in **Figure 2-2**. Each state specifies a subset of electrodes that will be activated; it typically takes 10 ms to transport a droplet to an adjacent cell, and mixing/dilution times during assay execution are on the order of seconds or tens-of-seconds. The linear state machine control model is wholly deterministic, which is acceptable for a scheduled DAG with no operation variability. To cope with bounded variability, it is possible to enumerate schedules for all possible combinations of operation times, which is exponential in the general case [6]. An alternative approach, which can accommodate assay operations that fail and require partial re-computation, is to pause assay execution temporarily and recompile the assay on-the-fly [7][52][90]. Although this overall approach results in the execution of a non-linear state machine, it uses dynamic recompilation to replace one linear state machine with another, as shown in **Figure 2-3**.

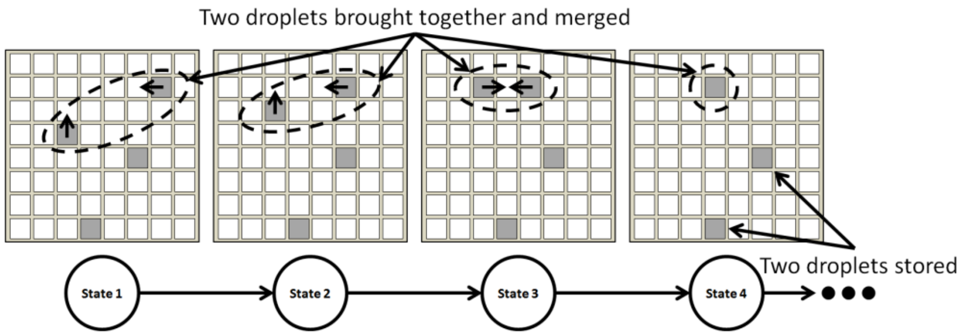


Figure 2-2: **Example of the linear state machine model of DMFB control. The output of each state is the subset of electrodes in the DMFB that will be activated during each time step (shown in gray). The state machine is timed, based on the activation frequency, typically 100 Hz [88]. In this example, two droplets are transported to a common location so that they can be merged, and two droplets are stored in-place.**

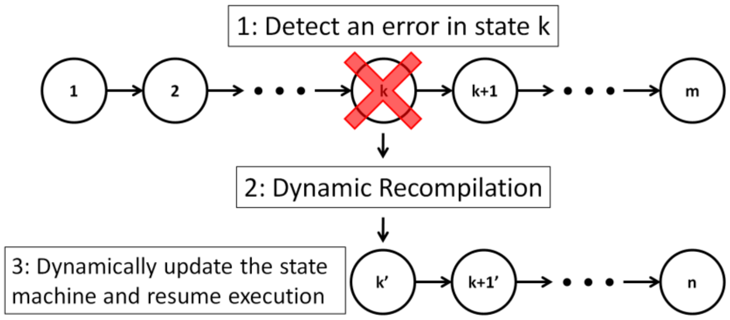

1: Detect an error in state k
1
2
k
k+1
m
2: Dynamic Recompilation
3: Dynamically update the state machine and resume execution
k'
k+1'
n

Figure 2-3: **In response to an error detected in state k, the assay is paused and recompiled, which includes the insertion of new states to recompute fluids that have been lost due to erroneous processing, which may execute concurrently with other ongoing assay operations that were not adversely affected. The output is a new linear state machine that compensates for and corrects the errors that occurred.**

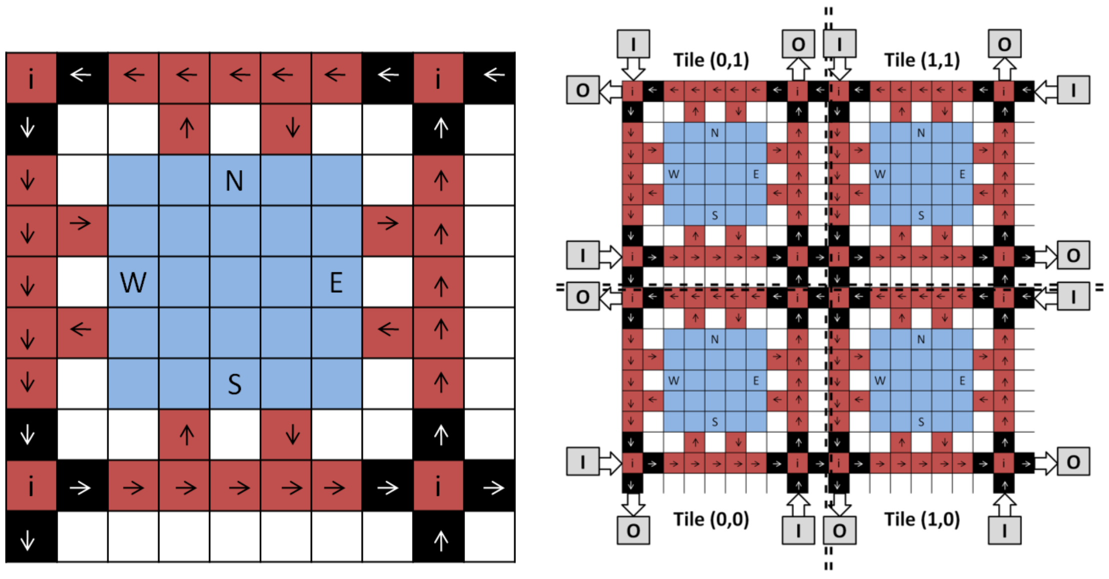
2.1.2 - CONTRIBUTION

We have developed a compiler and runtime system to translate assays specified using a high-level language into an executable form appropriate for a DMFB. Assays are specified using BioCoder [9], a C++ library for biological protocol specification developed at Microsoft Research, India. We present new language extensions to BioCoder that facilitate user-specified control-flow operations (e.g., conditionals, loops, droplet-transfer mechanisms). The interpreter can execute assays with control flow, use feedback from the DMFB to make control flow decisions, and does not rely on complex re-synthesis methods that dynamically recompile the assay when control flow becomes unpredictable [6][7][52][90]. To date, prior compilers targeting DMFBs are limited to assays specified as DAGs and cannot handle arbitrary control flow or feedback from sensors integrated onto the DMFB. The framework presented herein addresses these challenges through dynamic interpretation, thereby enlarging the space of assays that can be compiled onto EWoD devices.

2.2 - VIRTUAL TOPOLOGY

The system outlined in this chapter interprets assays dynamically, rather than compiling them statically and then (possibly) recompiling dynamically. Similar to prior work by Griffith et al. [26], the approach taken here imposes a *virtual topology* (**Figure 2-4)** on the DMFB that restricts the functions that different cells can perform; the interpreter exploits the restrictive structure to achieve fast algorithmic runtimes. Input and output reservoirs are placed on the perimeter of the DMFB, as seen in **Figure 2-4(b)**. The city blocks are referred to as *(work) modules*, because all non-I/O assay operations occur there. Each module can perform one operation (e.g., merging, mixing or splitting) or can store up to four droplets. External devices such as heaters or optical detectors can be affixed to the DMFB above or below a module. All streets are 1-way; eight 1-way streets meet together at rotaries, which offer an abstraction like a network router. Droplets travel clockwise through these rotaries.

Without loss of generality, a droplet traveling north that enters a rotary could continue straight, or turn left (west) or right (east); our routing algorithms do not allow droplets to reverse directions, so a droplet would not enter a rotary traveling north and then exit traveling south. A *tile* consists of a module and the four adjacent streets surrounding it, as shown in **Figure 2-4(a)**. If the module is 5x5, then a tile requires a 10x10 array of cells. Tiles are then repeated in two dimensions to form the virtual topology. For example, **Figure 2-4(b)** shows a virtual topology that is a 2x2 array of tiles.



(a) **(b)**

Figure 2-4: (a) A tile is the fundamental building block of the virtual topology and contains a work module (blue) where operations are performed and a network of one-way "streets" (red/black) for droplet transportation; (b) the virtual topology is imposed onto a DMFB by tiling the fundamental building blocks to create a 2D array of tiles.

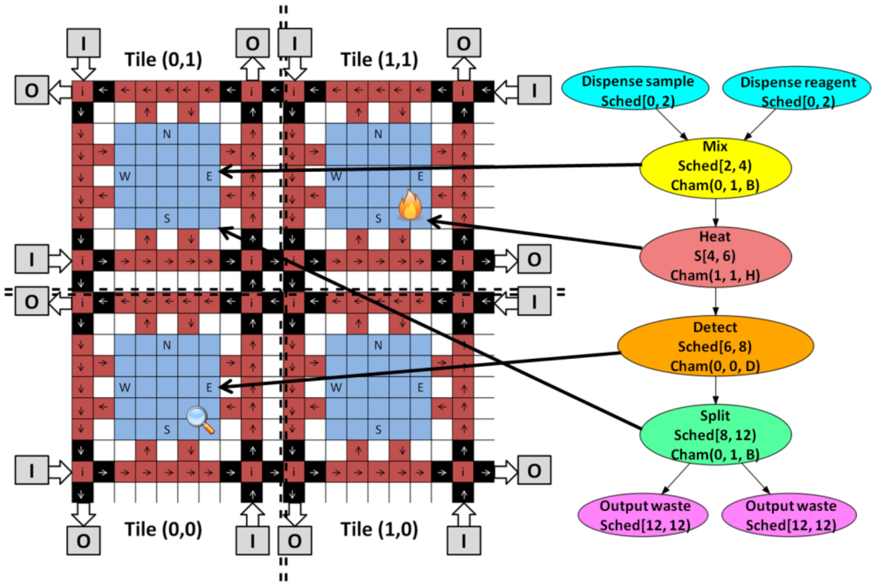
2.2.1 - SYNTHESIS SIMPLIFICATIONS

The virtual topology simplifies the problems of scheduling, placement and routing, which facilitates low-overhead dynamic interpretation and responsiveness to control flow:

Scheduling: With no virtual topology, the scheduler estimates the number of modules it can support from the dimensions of the microfluidic array. Ambitious estimations overestimate the number of concurrent operations that can occur on the DMFB surface and often leads to placement and routing errors; in these, cases, resource estimation and scheduling will need to be performed again. In contrast, a virtual topology provides a concise number of resources that are guaranteed to fit in the array, ensuring that placement and routing errors do not occur; thus, scheduling only needs to be performed once to compute an initial schedule.

Placement: Rather than placing operations at any location on the DMFB, the interpreter dynamically *binds* operations to modules, as shown in **Figure 2-5**. In principle, any available work module can be chosen; when multiple modules are available, the best choice is generally the one that is closest to the sources of the droplet(s) that are the operation's inputs; this minimizes droplet transportation latency.

Routing: The traditional approach to droplet routing is chaotic and disorderly, in part, driven by the fact that the placement of assay operations is likewise. The virtual topology, in contrast, imposes an orderly network of city streets that all droplets must follow. This limits the number of legal routes between each source-destination pair, which simplifies the process by which routes are computed. The approach taken here is to adapt deadlock-free 2D mesh network routing algorithms [18] to be compatible with DMFB technology. Thus, droplet routing follows a simple *protocol*, rather than solving a challenging constrained-optimization problem.



Tile (0,1)
Tile (1,1)
Tile (0,0)
Tile (1,0)
I
O
N
W
E
S
Dispense sample Sched[0, 2)
Dispense reagent Sched[0, 2)
Mix Sched[2, 4) Cham(0, 1, B)
Heat S[4, 6) Cham(1, 1, H)
Detect Sched[6, 8) Cham(0, 0, D)
Split Sched[8, 12) Cham(0, 1, B)
Output waste Sched[12, 12]
Output waste Sched[12, 12]

Figure 2-5: The operations of a scheduled DAG are bound to the modules indicated during the specified time steps.

2.2.2 - Deadlock-Free 2D-Mesh Routing

The virtual topology organizes the DMFB as a 2D-mesh network of work modules, where the rotaries play a role akin to network routers. One contribution of this chapter is to adapt deadlock-free 2D-mesh routing algorithms [18] to DMFBs using this virtual topology. Deadlock-free routing is important in an online system where assays are scheduled and executed on-the-fly because it is imperative to guarantee droplets can reach their destination.

To motivate the design for our virtual topology, we highlight the similarities and differences between our virtual topology and a 2D-mesh network in the following sub-sections. Later on, **'Section 2.5.2.3 - Droplet Transportation Protocol (DTP)'** shows how 2D-mesh routing algorithms are leveraged to achieve deadlock-free routing in our system. Griffith et al. [26], it should be noted, also achieved deadlock-freedom in their virtual topology, but did so by limiting the injection rate of droplets into the system.

2.2.2.1 - ANALOGUE TO 2D-MESH TOPOLOGY

As shown in **Figure 2-4**, a tile contains a module surrounded by one-way streets on each side. The module has an entry and exit on each side and each corner of the tile contains an intersection in which droplets can choose to stay in the current tile or travel to a neighbor (**Figure 2-6(a)**). The four streets and intersections surrounding the module form a counter-clockwise traffic circle called the *module rotary*. In **Figure 2-6(c)**, *exchange rotaries*, which allow droplets to move from one tile to its neighbors, are formed between tiles.

The virtual topology presented here shares many similarities with 2D-mesh networks:

Modules and I/O to Processors: In a computer network, processors send packets to one another. Likewise, a DMFB can route droplets from one module to another, or between modules and I/O reservoirs.

Module Rotaries to Routers: Figure 2-6(a) depicts a module rotary, which is similar to a traffic circle. The four intersections marking its corners are the entry points of the tile. The four streets (which are unidirectional) are similar to the buffers in a network router, shown in **Figure 2-6(b)**.

Exchange Rotaries to Wires: Figure 2-6(c) depicts an exchange rotary. The cells extending from the module rotary (the inputs and outputs in **Figure 2-6(a)**) are similar to wires in a 2D-mesh (**Figure 2-6(d)**). The exchange rotary connects four adjacent tiles, which form an inner cycle, similar to cycles formed among adjacent routers in a 2D-mesh, e.g., *Proc(0, 1)* in **Figure 2-6(d)**.

Streets to Buffers: Streets hold droplets, similar to input buffers of routers (see **Figure 2-6(a-b)**).

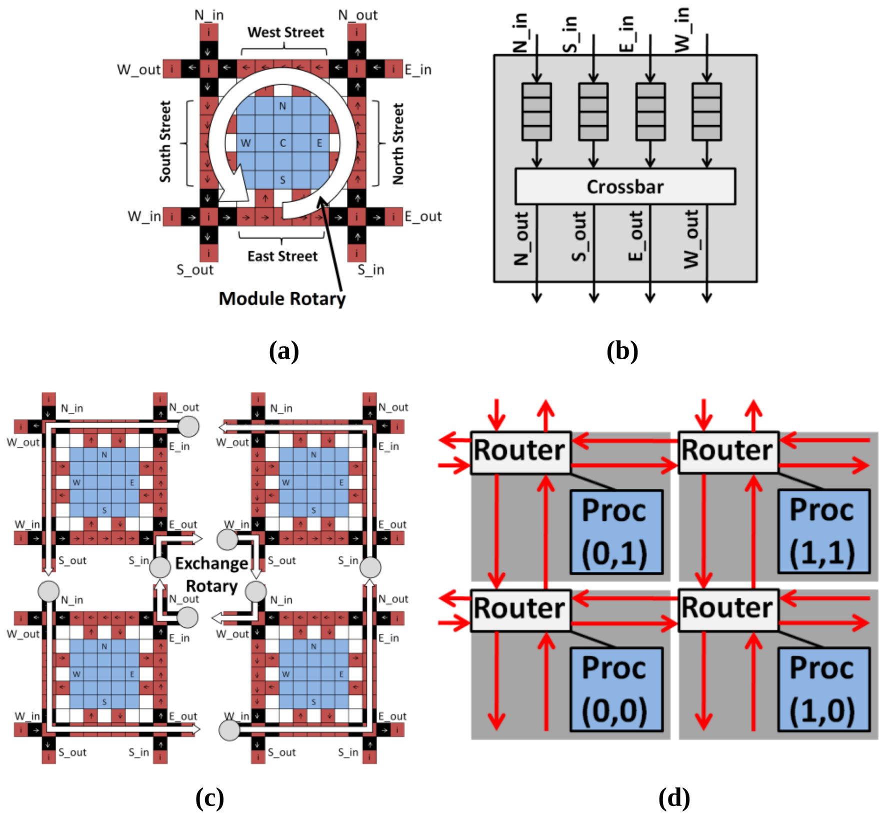


Figure 2-6: (a) A module rotary (the cycle formed by four streets surrounding a module and their intersections) is similar to (b) a 2D-mesh network router; (c) an exchange rotary (the clockwise inner loop) and a long counterclockwise cycle (outer loop) of a tiled virtual topology form equivalent connections to a (d) 2D-mesh network.

2.2.2.2 - Differences from 2D-Mesh Topology

Integrated circuits use wires to propagate signals, which are stored in buffers (flip-flops); in general, there is a clear separation between logic, storage, and interconnect. In contrast, DMFBs use cells for droplet transportation and storage. Also, the 2D-mesh router (**Figure 2-6(d)**) employs a crossbar (**Figure 2-6(b)**), which allows up to four signals to pass through concurrently. Exchange rotaries within DMFBs cannot employ crossbars, as droplets passing through the crossbar would inadvertently mix with one another.

2.3 - INTERPRETATION

The ability to perform deadlock-free droplet routing enables abstraction layers that share some principle similarities with the TCP/IP stack used in computer networks. This, in turn, facilitates an *intermediate bytecode format* (motivated by virtual machines, such as the JVM), which simplifies the design of the interpreter. Without loss of generality, suppose that we want to dynamically issue the command: "Mix droplets *x* and *y*." Under the recompilation paradigm described previously, the placement must be updated to make room for the new mixing operation (which, literally, could be anywhere on the chip, especially if other ongoing operations need to be moved), and then routes to deliver the two droplets must be computed algorithmically on-the-fly. In contrast, our interpreter could select *any* available work module, knowing that the *droplet transportation protocol* will deliver the two droplets (unless the device suffers from a physical failure). Similarly, this capability facilitates the interpretation of assays that feature control flow operations, as runtime decisions (i.e., which module executes each assay operation) can be made dynamically with minimal overhead.

2.4 - BIOCODER LANGUAGE AND EXTENSIONS

BioCoder [9] is a C++ library developed at Microsoft Research, India, for specifying biological protocols in an unambiguous fashion. BioCoder's compiler converts the assay specification into an English language description that is similar to a recipe in a cookbook. BioCoder's original purpose was to eliminate ambiguities that often occur when biological protocols are disseminated in peer-reviewed literature. The authors of the paper that introduced BioCoder suggested that it could be used as an input language to program an LoC; however, their initial work did not attempt to do so.

2.4.1 - LACK OF UNIVERSALITY IN LOC COMPILATION

BioCoder was designed to specify a wide variety of assays including many that are not compatible with the DMFBs that we target in this chapter. For example, BioCoder supports solid chemical data types and centrifugation; DMFBs cannot manipulate solids, and do not generally have integrated centrifuges; therefore, they cannot perform these operations. Unlike computer hardware and software, biochemistry has no theoretical notion akin to Turing completeness that can bound the capabilities of LoCs [8]. Similarly, there is no "universal" set of components akin to "universal" logic gates (e.g., **NAND**, or **AND-OR-INV**) that can provably implement any combinational logic function.

On the one hand, any language or library for specifying biological protocols must evolve as new components are developed for use: new operators (languages) or functions (libraries) to specify the usage of these components must be added; otherwise, the language or library itself will become stale over time. On the other hand, a compiler targeting a specific LoC technology is likely to support only a subset of the language; for example, any attempt to compile an assay that includes a centrifugation operation targeting a DMFB *must* fail, due to the lack of a centrifuge. This generally does not occur in software compilation. For example, many microcontrollers do not contain hardware multipliers, dividers, or floating-point units, but can still support these operations in software; compilation only fails when the device has insufficient memory. As biochemistry has no notion of a universal operator, compilation fails when the assay specification does not match the physical resources of the target device.

2.4.2 - OBJECT-ORIENTED ORGANIZATION

As mentioned earlier, BioCoder is a C++ library containing a variety of **structs**, global variables, and **static** functions. BioCoder's compiler creates an internal data structure that represents an assay as a DAG. The DAG is traversed to convert the assay into English language output. Assay information is not saved, and the data structure is deallocated during the traversal. We discovered that this library format was incompatible with the instantiation and maintenance of multiple assays at the same time.

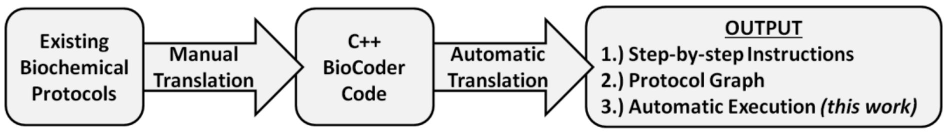
One of our goals was to introduce control flow into biochemical specifications in order to support assays where decisions are taken based on feedback from the LoC. This requires a control flow graph (CFG) where each basic block is represented as a BioCoder assay (i.e., a DAG). Naturally, any CFG containing control flow requires multiple assays.

BioCoder was converted to several C++ classes that enabled the construction of protocols comprised of multiple assays that no longer mangled one another when constructed. To specify an assay, the user instantiates an instance of the **BioCoder** class; assay operations are specified as method calls. The compiler converts the program into a graph-based intermediate representation using a new class that we introduced called **AssayProtocol**, which effectively represents the CFG. **AssayProtocol** enables the protocol to be saved, copied, and executed multiple times (if desired).

BioCoder Functions Supported by DMFBs	
Microfluidic Operations	**BioCoder Function**
Dispense	void measure_fluid (Fluid f, Volume v, Container c)
Output	void drain (Container c, string outputSinkName)
Mix/Merge	void vortex (Container c, Time t)
Split	void measure_fluid (Container c1, Volume v, Container c2)
Heat	void store_for (Container c, float temp, Time t)
Detect	string measure_fluorescence (Container c, Time t)

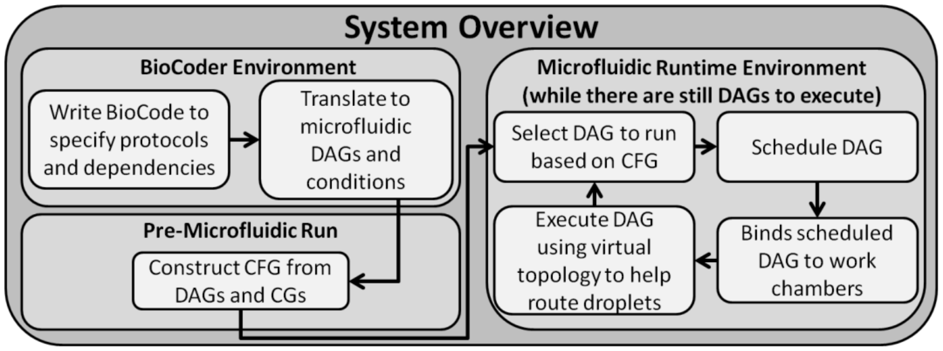
Table 2-1: Prototypes for six BioCoder functions that are supported by EWoD-based LoCs.

All original BioCoder functionality was left intact, so it remains possible to convert the assay to an English-language description or graphical representation if desired. **Figure 2-7(a)** depicts the new capabilities within the original BioCoder flow.



Existing Biochemical Protocols
Manual Translation
C++ BioCoder Code
Automatic Translation
OUTPUT
1.) Step-by-step Instructions
2.) Protocol Graph
3.) Automatic Execution *(this work)*

(a)



System Overview
BioCoder Environment
Write BioCode to specify protocols and dependencies
Translate to microfluidic DAGs and conditions
Pre-Microfluidic Run
Construct CFG from DAGs and CGs
Microfluidic Runtime Environment (while there are still DAGs to execute)
Select DAG to run based on CFG
Schedule DAG
Binds scheduled DAG to work chambers
Execute DAG using virtual topology to help route droplets

(b)

Figure 2-7: (a) Overview of the BioCoder system and output, highlighting the addition in this work; (b) System overview, showing the BioCoder environment, the runtime environment, and the interface between them.

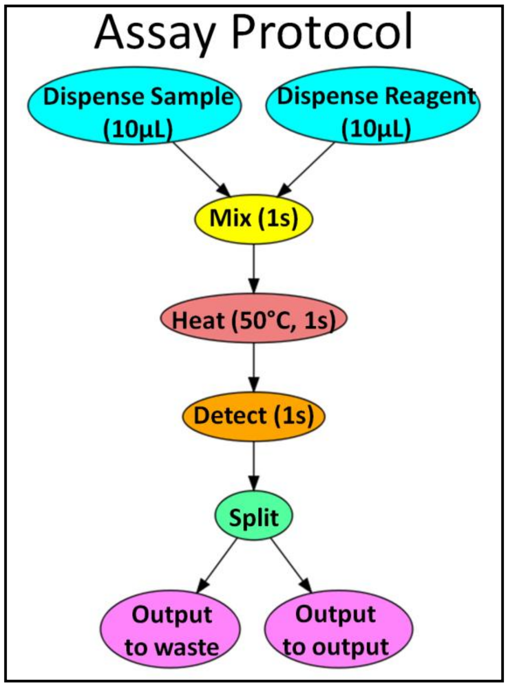
Table 2-1 lists six BioCoder functions that are compatible with the capabilities of EWoD-based LoCs. In our new implementation, these functions are methods of the **BioCoder** class. The data types **Fluid**, **Volume**, **Time** and **Container** used in **Table 2-1** are part of the original BioCoder specification. In traditional benchtop chemistry, the meaning of container is literal—e.g., it could be a test tube, beaker, or flask that *contains* fluid; in the case of our LoC, a **Container** is effectively used as a proxy for a droplet, which represents *embodiment*, rather than containment, of fluid.

2.4.2.1 - EXAMPLE

Figure 2-8 illustrates a simple protocol built using BioCoder. **Containers** represent droplets that carry fluids from one step to the next, while instances of the **Fluid** class act as input reservoirs. The protocol dispenses and mixes 10µl of a sample and reagent for 1s, heats the mixture for 1s at 50°C, detects the fluorescence for 1s, splits and outputs the two resultant droplets to the "output" and "waste" reservoirs. Each edge in the assay protocol graph represents a droplet flowing from one operation to the next, i.e., the fluidic analogue of a data dependency.

```
BioCoder * bio = BioSys->addBioCoder();
Volume reservoirVol = bio->vol(100,ML);
Volume dropVol = bio->vol(10, UL);
Time time = bio->time(1, SECS);
Container tube1 = bio->
      new_container(STERILE_MICROFUGE_TUBE2ML);
Container tube2 = bio->
      new_container(STERILE_MICROFUGE_TUBE2ML);
Fluid reagent = bio->new_fluid("reagent", reservoirVol);
Fluid sample = bio->new_fluid("sample", reservoirVol);

bio->first_step("Dispense Fluids");
bio->measure_fluid(sample, dropVol, tube1);
bio->measure_fluid(reagent, dropVol, tube1);

bio->next_step("Mix Fluids");
bio->vortex(tube1, time);

bio->next_step("Heat Fluids at 50 C");
bio->store_for(tube1, 50, time);

bio->next_step("Detect Fluorescence");
bio->measure_fluorescence(tube1, time);

bio->next_step("Split Fluid");
bio->measure_fluid(tube1, dropVol, tube2, false);

bio->next_step("Output Fluid");
bio->drain(tube1, "waste");
bio->drain(tube2, "output");
```

(a)



(b)

Figure 2-8: (a) BioCoder code for a sample assay and (b) the representative DAG structure. This assay does not require control flow.

2.4.3 - Extensions for Feedback and Control Flow

To support feedback and control-flow constructs, several new BioCoder classes have been created: **BioSystem**, **BioConditionalGroup**, **BioCondition**, and **BioExpression**. A **BioSystem** contains a list of **BioCoder** protocols and **BioConditionalGroups** which dictate the order in which the **BioCoder** assays are executed at runtime. As seen in **Figure 2-9**, a **BioConditionalGroup** is an **IF/ELSE-IF/ELSE** statement, where each **IF**, **ELSE-IF** and **ELSE** in the **BioConditionalGroup** is a **BioCondition**. Each **BioCondition** contains a **BioExpression** which can be evaluated to true or false at runtime and is used to determine which assay protocols to execute next.

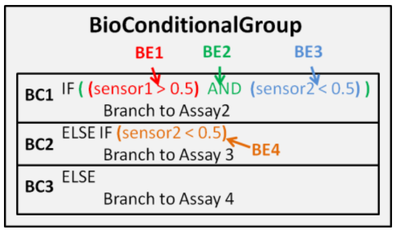


Figure 2-9: A BioConditionalGroup contains BioConditions (BC1-BC3). A BioCondition is evaluated by its BioExpressions (BE1-BE4).

Table 2-2 shows the five general types of **BioExpressions**. **BioExpressions** with operation types of AND, OR, and NOT are composed of one or more **BioExpressions**, which may be nested. **BioExpressions** are evaluated recursively at runtime to determine the result (True or False). The one- and two-sensor comparisons support decision-making based on feedback from sensors on the device. The functionality of the comparison depends on the data type returned by the sensors.

A **BioConditionalGroup** can have as many **ELSE-IF** statements as desired and is not ready to be processed until each of its **BioConditions**' dependent protocols have executed. Each **BioExpression** determines which **BioCoder** protocols its parent **BioCondition** is dependent upon. This value is passed in explicitly in the case of an unconditional expression. In the one- and two-sensor comparison expressions, the dependent **BioCoder** protocols are determined implicitly as the **BioCoder** protocols which contain sensor1 and sensor2 (for a two-sensor compare). The **measure_fluoresence( )** function (and other detection functions) return unique strings that act as tags for that specific reading and is the input for the one- and two-sensor compare expressions.

The control flow mechanism treats each protocol as a DAG, and uses control flow to determine which protocols to execute (in sequence) at runtime. In this respect, it is also necessary to transfer droplets from one protocol to the next, depending on which conditions are met. **Table 2-3** lists new operations and functions that we added to BioCoder to enable the transfer of droplets from one protocol to another.

BioExpression Types			
Expression Type	C++ Style Operator	Evaluation Type	BioCoder Construction
Unconditional	true, false	Direct	BioExpression (BioCoder *parent, bool unconditional)
One-Sensor Comparison	>, <, ≤, ≥, ==	Direct	BioExpression (string sensor1, OpType ot, double constant)
Two-Sensor Comparison	>, <, ≤, ≥, ==	Direct	BioExpression (string sensor1, OpType ot, string sensor2)
AND/OR	&& / ||	Nested	BioExpression (OpType andOr)
NOT	!	Nested	BioExpression (BioCoder *notExp)

Table 2-2: BioExpressions create simple or complex, nested expressions for branching functions.

New BioCoder Features for Control Flow	
New Microfluidic Operation	New BioCoder Function
Transfer_In	string reuse_fluid (Container con)
Transfer_Out	string save_fluid (Container con)

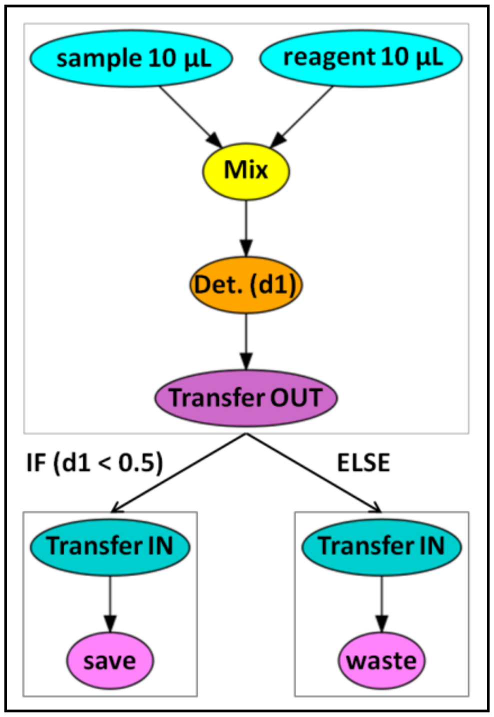
Table 2-3: Microfluidic operations and BioCoder functions that enable droplet transfers between protocols.

2.4.3.1 - EXAMPLE

Figure 2-10 shows an example BioCoder protocol that uses conditionals. This example illustrates the instantiation of a **BioConditionalGroup**, **BioExpression**, and **BioCondition** in sequence, followed by setting up the targets of the condition that form CFG edges (the **addNewCondition** method) and the mechanism to transfer droplets from one basic block to another (the **addTransferDroplet** method). Admittedly, this syntax is somewhat unwieldy compared to a more straightforward if-then-else statement. The fundamental challenge here is that BioCoder represents each assay as a DAG, and extending that representation to a model that includes control flow would break BioCoder's ability to output an English language description of each DAG—at present, BioCoder's cookbook-style output is linear and does not naturally support conditionals. This requirement forced us to create a syntax that is far from ideal. However, this syntax can easily be hidden by a simple graphical user interface (GUI) wrapper program that presents high-level if-else options while calling the underlying BioCoder functions. Nevertheless, we hope to switch to a more convenient syntax in the future and extend BioCoder's English language output capabilities to account for it.

```
// BioCoder protocol declarations & operations
BioCoder * bioMixDet = BioSys->addBioCoder();
bioMixDet->measure_fluid(sample, dropVol, tube);
bioMixDet->measure_fluid(reagent, dropVol, tube);
bioMixDet->vortex(tube, time);
d1 = bioMixDet->measure_fluorescence(tube, time);
to = bioMixDet->save_fluid(tube);

BioCoder * bioSave = BioSys->addBioCoder();
ti_save = bioSave->reuse_fluid(tube);
bioSave->drain(tube, "save");

BioCoder * bioWaste = BioSys->addBioCoder();
ti_waste = bioWaste->reuse_fluid(tube);
bioWaste->drain(tube, "waste");

// Conditional Groups
BioConditionalGroup *bcg = BioSys->addBioCondGroup();
BioExpression *be = new BioExpression(d1, OP_LT, 0.5);
BioCondition *bc = bcg->addNewCondition(be, bioSave);
bc->addTransferDroplet(to, ti_save);
bc = bcg->addNewCondition(NULL, bioWaste);
bc->addTransferDroplet(to, ti_waste);

bio->end_protocol();
```

(a)



(b)

Figure 2-10: (a) BioCoder code illustrating the use of conditionals; (b) the resultant CFG.

2.5 - SYSTEM OVERVIEW AND RUNTIME ENVIRONMENT

Figure 2-7(b) shows an overview of the BioCoder environment, including the interface between the compiler and the runtime system. A chemist (programmer) specifies the "BioSystem" of assay protocols and dependencies (control-flow and droplet-transfer). The compiler transfers its protocols and **BioConditionalGroups** into microfluidic DAGs and conditional groups (CGs), which are passed as a direct input to the runtime system.

The pre-runtime system constructs a CFG from the DAGs and CGs given as input from BioCoder. The runtime system then selects which basic block to execute based on the conditions that are evaluated at runtime. DAG operations are scheduled, bound to the "work module" areas of the LoC (**Figure 2-4**) at runtime, and executed dynamically; details are described in the following subsections.

The runtime system is a program that runs on a PC that sends signals to the LoC to actuate fluid motion; in our implementation, the LoC is simulated in software. The runtime system receives the **AssayProtocol** data structure produced by the BioCoder compiler and processes it to induce assay execution. This section describes the key algorithms that are used to determine how and where to apply the different assay operations shown in **Figure 1-3** (i.e., transportation, splitting, merging, mixing and storage).

2.5.1 - INTERMEDIATE BYTECODE FORMAT AND INTERPRETER

Conceptually, the set of signals sent to a DMFB during each cycle can be viewed as a machine language. If the DMFB is comprised of *N* cells, then *N* binary signals are sent to the device (e.g., a '1' activates an electrode, and a '0' leaves it off). This is a relatively low level of abstraction; however, this level is the target of the static compilation and re-compilation approaches shown in **Figure 1-5**. The virtual topology (**Figure 2-4(b)**) raises the level of abstraction at which the DMFB can be controlled. Recall that a device's control program executes on a PC or microcontroller that sends signals to the DMFB in order to activate the electrodes that induce droplet motion. Under static compilation, the device's control program is a realization of the linear state machine model (**Figure 2-2** and **Figure 2-3**).

The interpreter is a software application that accepts a partially compiled BioCoder assay (its CFG representation) and decomposes each operation into a short sequence of bytecode instructions that are executed dynamically. The bytecode format does not include timing information; the interpreter is responsible for keeping track of time, and its foremost responsibility is to issue and execute bytecode instructions at the correct time.

2.5.1.1 - BYTECODE INSTRUCTION FORMAT

The virtual topology enables the device control program to evolve from a state machine into a fully functional virtual machine with its own *intermediate bytecode language* that is simple, yet operates at a much higher level of abstraction. Bytecode instructions are categorized as operational *(O-type*) and transport *(T-type)*.

Each O-type instruction has the form *(opcode, module-id)*, where the opcode specifies the operation to perform, and the module-id specifies which module to perform the operation. All modules support four basic opcodes: *{start-mix, stop-mix, split, store}* (merging is viewed as a precursor to mixing). If a module has an external device affixed to the outside of the chip, such as a heater or detector, then it may support additional opcodes such as *{heater-on, heater-off, detector-on, detector-off}*. There are some straightforward restrictions, such as limitations on the number of droplets that a work module can store (four for this topology), and, other than multi-droplet storage, each work module can only perform one operation at a time.

Each T-Type instruction has the form *(src, dst)* or *(droplet-id, src, dst)*, which transports a droplet from the source (src) to a destination (dst). If the source contains a single droplet, then the identifier of the droplet is implicit and is not needed; if it contains multiple droplets, then the *droplet-id* field is required for disambiguation. This generally occurs in two situations: (1) a work module stores multiple droplets, only one of which will be transported; or (2) a work module performs a split operation, thereby creating multiple distinct droplets; the default behavior is then to store the two droplets that have been created. If a T-type instruction transports a droplet to a module, it is stored implicitly, until an O-type instruction initiates an operation.

2.5.1.2 - I/O OPERATIONS

An important issue with respect to the design of the virtual machine is whether I/O operations should be O-type or T-type instructions. DMFB I/O mechanisms presently lack standardization, and there exist several different ways to move droplets onto a chip [64]. One approach is to generate a droplet from a pressurized off-chip source, such as a pipette or needle; once on the chip, the droplet should be transported to an appropriate location for storage or other processing, unless it is deposited precisely onto the location where it will be used or stored. In this case, the amount of time required to input the droplet is non-negligible compared to the time required to transport a droplet.

An alternative approach is to store fluids in on-chip reservoirs; the amount of fluid stored in a reservoir is significantly larger than the size of a droplet. In this case, individual droplets for processing can be “split” from the reservoir, and then transported to their appropriate locations. In this case, the input process is simply a variation of droplet transport.

The first input approach naturally lends itself to an O-type operation, especially since it is timed. This approach is sufficient as long as there are known a-priori locations on the DMFB where droplets will be deposited; our convention is to place such locations on the perimeter of the chip. Thus, each location can have a unique identifier and can easily be specified as a source, even though it is not a work module and cannot perform other operations, such as mixing and splitting. Once a droplet has been input to the device, it can be transported to its location for processing using a T-type instruction. In contrast, the second approach naturally lends itself to a T-type operation, since the droplet is split and transported away from the reservoir over a relatively short sequence of time steps (see ref. [64], Fig. 9, for details).

The interpreter implements both options, in order to best support different types of input mechanisms. Output and disposal operations are represented as T-type instructions, as no meaningful “processing” is applied to a droplet in order to remove it from the chip.

2.5.1.3 - DROPLET IDENTIFICATION

Some additional internal bookkeeping is necessary to track the names and identification numbers of droplets. The discussion of this bookkeeping has been omitted, thus far, in order to simplify the discussion. In practice, droplet names are only needed to disambiguate the situation where multiple droplets reside in a module; this may occur as a result of a split operation, or if the module stores more than one droplet. In this case, a T-type operation of the form *(src, dst)* would be ambiguous, as it is unclear which droplet should be transported. Similarly, mixing operations merge two previously distinct droplets into one, thus an appropriate naming convention is required.

The interpreter adopts the following conventions regarding droplet numbering. Let *n* denote the next available identification number for a new droplet. Initially, $n = 0$.

Dispensing: Each droplet that is injected into the DMFB is assigned identification number *n*; *n* is then incremented.

Mixing: When droplets having identification numbers *i* and *j* are mixed, $i < j$, the resulting droplet receives identification number *i*; identification number *j* is no longer available for future droplets and cannot be reclaimed.

Splitting: When a droplet having identification number *i* is split, the resulting droplets are assigned identification numbers *i* and *n*; *n* is then incremented.

Disposal: When a droplet having identification number *i* is transported off-chip (e.g., to an output or waste reservoir) its identification number is no longer available for future droplets and cannot be reclaimed.

2.5.1.4 - KEEPING TRACK OF TIME

The bytecode format does not include timing information. The virtual machine, which interprets BioCoder assays, is responsible for keeping track of time between starting and stopping assay operations. It is important to understand that the bytecode format is never compiled directly to a human-readable text file (except internally for development and debugging purposes); instead, the virtual machine issues and executes commands in an on-the-fly fashion, using a work queue of operations that have been stamped with information regarding their execution time.

For example, to mix two droplets in work *module M* for 20 seconds, the virtual machine would immediately issue the bytecode instruction *(start-mix, M)* and, at the same time, add a command *(stop-mix, M)* to the work queue, with the time *t+20* seconds, where *t* is the present time. The work queue is implemented as a priority queue, where the highest priority entry is the next bytecode instruction (among all those in the queue) that needs to execute. This way, as time progresses, the interpreter only needs to compare the current time with the time at which the highest priority bytecode operation in the queue; once that operation executes, the priority queue is once again adjusted so the highest-priority operation sits in the queue. This ensures the correct execution of timing-driven operations.

T-type instructions are variable-latency operations, because the time required to transport a droplet from its source to its destination depends on the amount of congestion in the DMFB. The interpreter maintains a separate list of in-transit droplets. When a droplet completes its route, the corresponding T-type operation is removed from this list. **'Section 2.5.2.3 - Droplet Transportation Protocol (DTP)'** describes the algorithms used to perform on-line droplet routing.

For debugging purposes, the system can produce a time-stamped trace of bytecode instructions. Each instruction is time-stamped with its start and finish times. Certain O-type operations, such as splitting and merging, occur within a single time quantum, so their execution time is treated as zero. Mixing and transport operations, in contrast, require multiple time quanta, so their finish times are always later than their start times.

2.5.2 - Interpreting a DAG on the Virtual Topology

Three steps are required to compile a DAG onto the virtual topology: scheduling, operation binding, and droplet routing. Since the droplet routing protocol/algorithm we developed is specific to the virtual topology described in **Figure 2-4**, we describe it in detail in this sub-section. However, the scheduling and binding algorithms we use are applicable to virtual topologies, in general; thus, in an effort to keep the focus of this chapter on interpretation, we only present high-level details for scheduling and binding and save a more detailed presentation of these synthesis algorithms for **'Section 3.4 - Fast Online Synthesis'**.

2.5.2.1 - Scheduling

We use a straightforward list scheduling algorithm for droplet-based LoCs targeting the virtual topology [75]. The overall goal of the problem formulation is to find a legal schedule having minimum latency. Scheduling is NP-complete, and list scheduling is a naïve, but efficient, heuristic.

Each DAG in the CFG is scheduled separately; the interpreter manages control flow transitions at runtime. DAG operations are scheduled in discrete time steps. At each time step, the algorithm considers all sources (i.e., DAG vertices having no predecessors) for scheduling. It selects as many of these operations as possible, within the resource limits of the LoC and virtual topology. The scheduled operations are then removed from the DAG and the process repeats until all operations have been scheduled. Storage operations are inserted when gaps between dependent operations occur in the schedule. In our virtual topology, each module can store up to four droplets, however, it cannot perform any mixing operations if it is storing at least one droplet. The schedule computed by this algorithm is somewhat inexact, as it does not account for droplet routing times; however, this is not usually problematic, as droplet routing times are several orders of magnitude faster than assay operations (e.g., milliseconds to move a droplet from one cell to its neighbor, compared to seconds to perform an assay operation).

Suppose that the virtual topology provides M work modules, I input reservoirs, O output reservoirs, and W waste reservoirs. These resources limit the number of different compatible operations that can occur concurrently. During each time step t, m_t mixing operations, s_t storage operations, i_t input operations, o_t output operations, and w_t waste operations as long as the following equations are satisfied:

$$m_t + \left\lceil \frac{s_t}{4} \right\rceil \leq M,\ i_t \leq I,\ o_t \leq O,\ and\, w_t \leq W \tag{2.1}$$

Assay operations that require external components, such as sensors or heaters, must be scheduled onto modules that provide those elements. Without loss of generality, the number of heating operations scheduled concurrently cannot exceed the number of modules on-chip to which heaters are affixed, etc.

Similarly, there may be some constraints imposed on the fluidic input process. For example, if the LoC has two input reservoirs supplying fluid A, then it is impossible to schedule two input operations for fluid type A concurrently.

In some cases, resource constraints cannot be satisfied based on an assay’s demand for operational and storage resources; when this occurs, the only option is to switch to a larger LoC device that provides more modules.

2.5.2.2 - BINDING

After scheduling, each DAG operation is annotated with the time-step and resource type to which it is bound: the three operational resource types are general modules, heating modules, and detecting modules; the three I/O resource types are input, output, and waste reservoirs.

The binder selects appropriate resources for each operation that has been scheduled. For illustrative purposes, we briefly describe a simple, greedy, efficient binding algorithm adapted from the Left Edge Algorithm used for dogleg routing in VLSI [35] and register allocation in high-level synthesis [44]; **Figure 2-11** shows an example. Operations are first separated into bins based on the module-type they were assigned during the scheduling phase. The bins are then sorted by their starting time-step. Finally, each resource (e.g., module or I/O) is paired with a bin of operations that matches its resource type and non-overlapping operations are bound to that resource until it reaches the end of the bin. Since a legal schedule satisfying resource constraints has been established, operations will be bound to a compatible resource at the end of the algorithm.

The focus of this chapter is interpretation and language constructs, and thus, we do not go into great detail about synthesis algorithms here. For pseudocode and a detailed explanation of our left-edge binding algorithm, please see **'Section 3.4.2.1 - Left-Edge Binding Algorithm'**. In general, it is a good strategy to bind each operation to a resource nearby the resources that supply its inputs (left-edge binding does not do this). For example, if we want to mix fluid inputs A and B, it may be a good idea to choose a module that is relatively close in proximity to their two input reservoirs. Thus, we also present a path-based binding algorithm, in detail, which takes spatial locality into account, in **‘Section 3.4.2.2 - Path-Based Binding Algorithm’**.

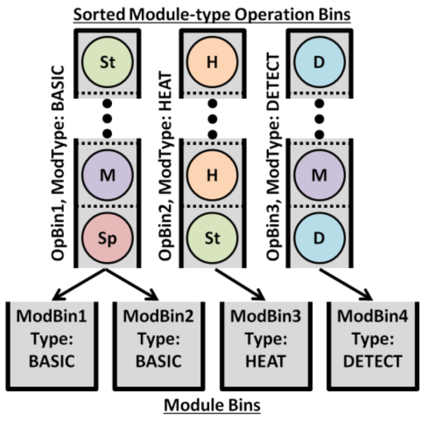


Figure 2-11: Illustration of the left-edge binding solution.

2.5.2.3 - Droplet Transportation Protocol (DTP)

The DTP chooses a path in the virtual topology for each droplet, and then routes the droplets along their respective paths while satisfying all droplet interference constraints, as described in **‘Section 1.3.3 - Droplet Routing’**. Deadlock prevention is the foremost responsibility of the DTP. In this respect, the rotaries in the virtual topology take on a role similar in principle to network routers; however, there are several important differences that we mention here. Firstly, most network routers contain an internal crossbar that connects input ports to output ports; packets or flits are selected from one (or more) input buffers and are transmitted to the corresponding output buffers. This type of router topology is not feasible in DMFB technology, as the droplets in transit across a crossbar would necessarily collide with and contaminate one another. A second difference, which generalizes from the first, is that digital logic has an implicit separation of logic, storage, and routing resources; with respect to routing networks, this means that buffers, crossbars, and wires are distinct. In contrast, the DMFB has a single resource (a cell) that performs both storage and transport in the context of the rotary; this has significant implications for protocol design which are discussed in detail in the subsequent paragraphs. Lastly, certain mechanisms, such as virtual channels [17], which can ensure deadlock freedom in computer networks, cannot be applied to droplets due to the lack of a crossbar in a rotary; thus, more restrictive mechanisms are needed to prevent deadlock from occurring.

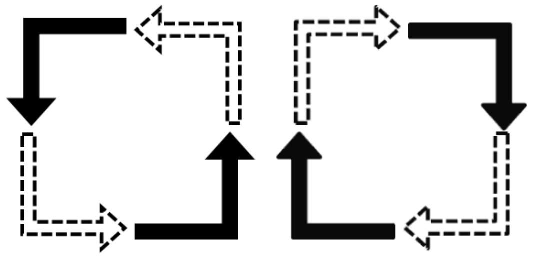


Figure 2-12: Legal turns (black) and prohibited turns (dashed outline) in XY routing.

The DTP adapts dimension-ordered routing (e.g., XY or YX routing in 2D) for the DMFB virtual topology. Conceptually, XY routing moves each droplet from its source position *(x1, y1)* to its destination position *(x2, y2)*, by first traveling along the *x*-axis to *(x2, y1)* and then traveling along the *y*-axis to complete the route. XY routing is deterministic and non-adaptive, but worked well enough for our purposes.

XY and YX work by preventing two turns in each cycle, as seen in **Figure 2-12**. Another class of routing algorithms that can be used with the virtual topology, based on the turn model, prevent deadlock by carefully selecting and prohibiting a single turn from each cycle. Negative-first (NF), north-last (NL) and west-first (WF) routing algorithms all prohibit one turn in each cycle to eliminate deadlock [25]. Odd-even (OE) routing is another adaptive algorithm that prevents deadlock by prohibiting some types of turns in certain tile columns [15]. For brevity and scope, we omit further discussion of network routing algorithms and assume XY is used for the remainder of this work.

XY routing requires several modifications to account for rotaries, the I/O reservoirs on the perimeter of the DMFB, and the process by which droplets enter and exit a module. The four streets and intersections surrounding a module form a counter-clockwise traffic circle called a *module rotary*, shown in **Figure 2-6(a)**. The term *exchange rotary* is introduced to refer to the original rotaries which allow droplets to move from a tile to one of its neighbors. **Figure 2-6(c)** shows that droplets travel clockwise through an exchange rotary, and counterclockwise through one or more module rotaries. Groups of droplets traveling along XY paths can form cycles in module and exchange rotaries, and therefore, preventing the formation of these cycles prevents deadlock.

Four specific rules are required:

1.) Module Entries and Exits: Droplets may not make prohibited turns (**Figure 2-12**) when leaving source and entering destination modules. To ensure routability in light of prohibited turns, entries and exits are placed on all four sides of the module.

2.) Droplet I/O: To prevent forbidden turns, input, output, and waste reservoirs are placed on the DMFB perimeter and the allowable turns that a droplet may make at an entry point are limited.

3.) Exchange Rotaries: In **Figure 2-13(a)**, a droplet *clips* an exchange rotary if it touches one intersection before leaving. In **Figure 2-13(b) and (c)**, a droplet *passes* *through* an exchange rotary if it touches at least two intersections. As droplets move clockwise within an exchange rotary, a clip implies a left turn, and passing through implies that the droplet continues traveling straight or turns right. **Figure 2-13(d)** depicts exchange rotary deadlock when four droplets attempt to pass through; no droplet can progress without maintaining spacing constraints (**Figure 1-8**). In **Figure 2-13(e)**, deadlock is eliminated if at least one droplet clips the exchange rotary. *To prevent deadlock in an exchange rotary, at most three droplets that wish to pass through may enter concurrently.*

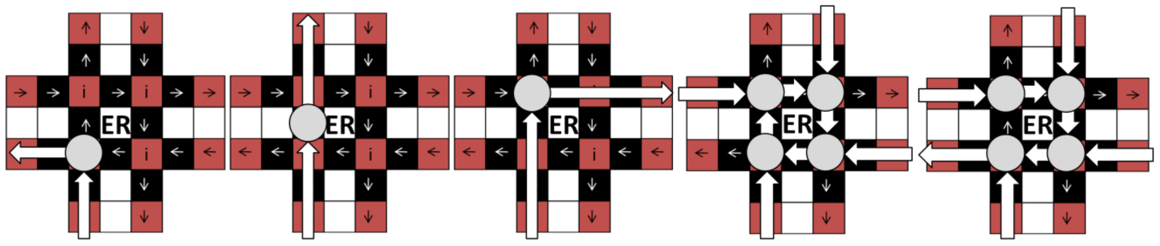


Figure 2-13: (a) Clipping an exchange rotary ('ER'); (b) passing through an ER while traveling straight; (c) passing through an ER while turning right; (d) deadlocked ER; (e) non-deadlocked ER.

4.) Module Rotaries: Figure 2-14(a) illustrates module rotary deadlock. Droplet 16 creates a dependency chain which causes deadlock; however, if it does not enter the module rotary, then a bubble is created which ensures that the sequence of droplets can proceed, starting with Droplet 1. *To prevent deadlock in a module rotary, no droplet may enter an exchange rotary unless the system can guarantee that there is space for it to exit into the next street; if the street is full, then the droplet must wait for space to become available prior to entering the exchange rotary.* Droplets attempting to enter a street from an adjacent module or input reservoir must also wait until that street has room; in **Figure 2-14(b)**, Droplets 1, 2, 3, 5 and 6 must wait for this reason.

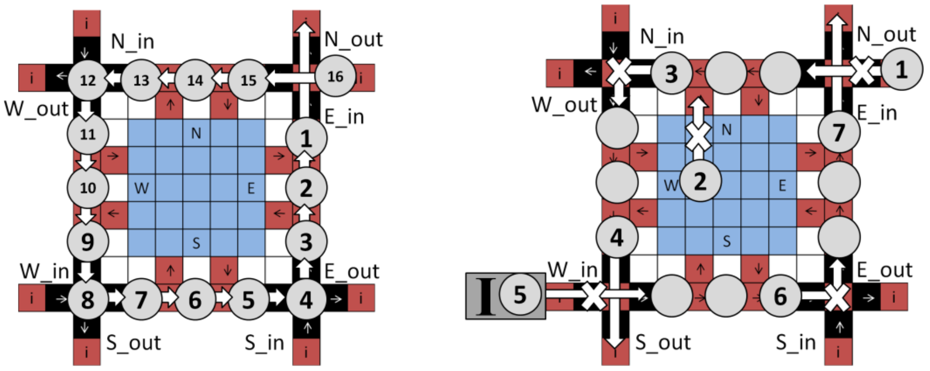


Figure 2-14: (a) Deadlock in a module rotary; (b) Module rotary with street capacity rules being enforced to prevent deadlock.

2.5.3 - CFG Execution

Given the ability to execute a DAG, generalizing the runtime system to execute a CFG is straightforward. Before executing a CFG, the DAG corresponding to each basic block is scheduled to determine whether the LoC can meet its resource demand. If all DAGs can be scheduled, then the CFG can execute. Much like a software program, CFG execution proceeds one basic block at a time. Each DAG can be partially compiled in isolation; however, the transfer of droplets from one DAG to another in response to a conditional evaluation occurs dynamically. Referring to the right of **Figure 2-7(b)**, the entry and exit nodes of the CFG are known. Starting with the entry node, the system dynamically schedules, binds, and executes each DAG. When the DAG finishes its execution, its conditions are checked and the next DAG to execute is chosen. This process repeats until the CFG exit node completes its execution.

2.6 - Simulation Results

We developed a software simulator for EWoD-based LoCs in C++, and our compiler and runtime system currently interface with it. At each time step, the runtime system sends signals to the simulator indicating which electrodes to activate, based on the assay operations that are currently executing and the droplets that are currently undergoing transport. The simulator estimates the execution time of an assay based on operation latencies provided in the **Appendix**.

We chose a low-end embedded processor to evaluate these benchmarks because DMFBs have the potential to be used in battery-powered point-of-care diagnostic devices [22] that can be deployed in remote rural areas, possibly in third-world countries; in such a context, it would be a significant burden to transport a modern desktop or laptop PC to control the DMFB, and then power it up. Initially, we considered a low-end microcontroller implementation, but our code base was too large to fit into the limited memory of the device. Instead, we compared the interpreter with the static compiler using an Inforce SYS9402-01 development board, which features a 1GHz Intel Atom^{TM} E638 processor with 512MB RAM, running TimeSys 11 Linux; memory constraints were not a concern using this more powerful platform.

2.6.1 - EXPERIMENT #1: FAULT-TOLERANT SPLITTING

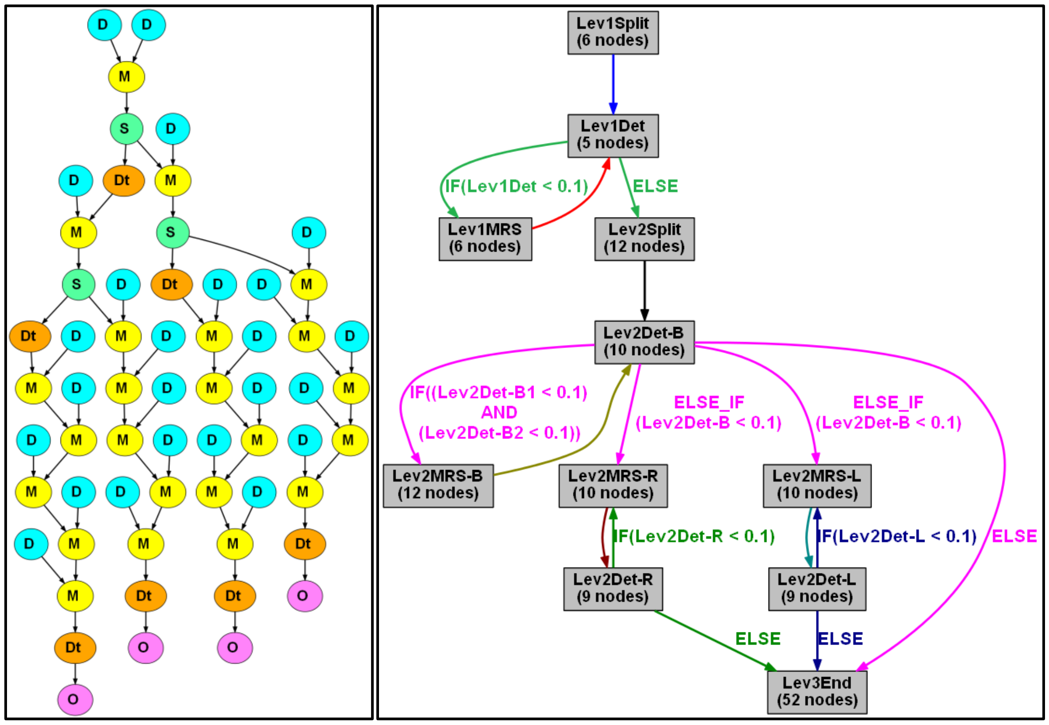
A common assumption made when compiling assays onto EWoD-based DMFBs is that every split operation is perfect, i.e., the two split droplets have equal volumes; however, this may not be the case. Some assays require an exact volume and demand a certain amount of precision when working to obtain a specific concentration. As droplet volumes decrease, an uneven split may significantly bias assay results. After the split, the volume of one of the two resulting droplets is measured. If the volume obtained from the measurement is sufficiently close to half of the pre-split volume, then the split is deemed to be successful; otherwise, it fails, and the two split droplets are re-merged and the split operation repeats [7]. This process, which is naturally probabilistic, can be expressed as a loop using the extensions to BioCoder.

Alistar et al. offer two solutions to this problem. The first modifies the DAG to form a fault tolerant sequencing graph (FTSG) where each split is attempted a fixed number of times [6]. This changes the assay scheduling problem formulation, because splits are now variable-latency operations, whose exact latencies cannot be known until runtime. The second solution applies incremental re-synthesis (e.g., **Figure 2-3**) in response to an erroneous split: when a split-error occurs, the resultant droplets are discarded and a recovery sub-graph is called upon to reproduce the droplet to be split again [7]. This eliminates the primary limitation of their former approach—fixing the number of times a split may be performed—but the re-synthesis process is more complex, as it re-executes lengthy operations that may not need to be performed again.

To evaluate our interpreter and virtual topology (INT), we compare against Alistar's online re-synthesis (ORS) idea in which time-redundant graphs are re-synthesized online when a fault is encountered in order to reproduce the erroneously split droplets. Their approach suffers from two limitations, which our implementation addresses here. Firstly, it appears that their placer does not address the challenge of specialized modules with external devices, e.g., detection operations must be placed on top of cells above or below an integrated sensor. Secondly, their approach does not perform routing, so those overheads are taken into account. Therefore, our implementation considers their general ORS approach to error detection and recovery, but uses a different fast online synthesis flow [31] that accounts for specialized modules during placement and includes a routing algorithm.

We used a truncated 2-level version (for the sake of clarity and demonstration) of a larger 3-level protein dilution assay [76], as shown in **Figure 2-15(a)**. We do not have a dilute operation, and thus, we replaced each 5s dilute operation with a 3s mix and 2s split operation. For the entire assay, dispense operations are 7s, mix operations are 3s, split operations are 2s and output operations are 0s. The detect operations trailing a split are 5s, while the detect operations preceding an output are 30s. These timings were kept consistent for INT and ORS and both assays were implemented using our enhancements to BioCoder. INT uses a 20x20 DMFB, allowing for 4 total work modules with 4 detectors; ORS uses a 15x19 DMFB, allowing for 6 total work modules, and has 4 detectors as well.

Figure 2-15(a) shows the initial sequencing graph executed by ORS, while **Figure 2-15(b)** shows the control flow graph executed by INT (for a failure probability of 10%). The *Lev1Split* DAG contains the first mix and split seen in **Figure 2-15(a)**. The program loops between *Lev1Det* and *Lev1MRS* (Merge and Re-Split) until the split is successful. When the first split succeeds, *Lev2Split* is scheduled and executed, which performs the 2nd level of splits (containing the bottom 2 splits). Since there are two splits, there are detect and merge-re-split (MRS) loops for the occasions when the left split, right split, or both splits fail (DAGs ending in "-L", "-R", and "-B", respectively). Finally, when all splits have been properly performed, *Lev3End* executes the final dispense, mix, detect and output nodes.



(a) **(b)**

Figure 2-15: Output from system showing (a) the initial 2-level protein DAG executed by the online re-synthesis model and (b) the control-flow graph executed by our interpreter.

We are primarily interested in the synthesis and assay runtimes introduced by split recovery errors in INT and ORS. We ran INT with error probabilities of 10%, 25%, 50%, 75% and 90% on each split node. Our simulator does not model the physics of the droplet split and detection process. Instead, our virtual sensor returns a probability in the range [0, 1). If the probability is less than the error probability/threshold, we assume that the split is successful; otherwise, we assume that the split is a failure. This applies to all splits, even if part of a merge-re-split. We performed 10 runs for each of the 5 error probabilities (50 total runs) and averaged the results for each error rate. Next, we examined the control-flow of each of the 50 runs and reproduced the same exact errors in the ORS system to obtain comparative numbers.

Table 2-4 reports the computational time that both INT and ORS spent performing synthesis and responding to faults for different error probabilities. INT uses the interpreter to synthesize the assay in an online fashion, while ORS computes an initial synthesis result up-front, and then re-synthesizes the assay each time that a fault occurs. **Table 2-5** shows the assay runtime (operation + routing time) for the same error probabilities as seen in **Table 2-4**. Each table has a fault-free baseline, meaning these numbers show synthesis and assay runtimes, respectively, for the 2-level protein application when no errors occur. For ORS, this is the DAG seen in **Figure 2-15(a)**. For INT, this represents the execution path seen in **Figure 2-15(b)** of *Lev1Split* → *Lev1Det* → *Lev2Split* → *Lev2Det-B* → *Lev3End* because this is the path that executes when no errors occur.

As shown in **Table 2-4**, ORS takes less time than INT to generate an initial fault-free schedule, although INT generally spends less total time handling errors online (T). **Table 2-4** also shows that INT spends less time, on average, in response to each error (PE). One thing to note is that the per-error time (PE) is greater than the total time (T) for both 10% and 25% error rates; this occurs because the average number of errors per run (EPR) is less than 1 for both of these error rates. Thus, an error does not occur every single run and causes the total synthesis time to be less than the synthesis-per-error time.

Average Recovery Synthesis Time With Fault-Free Baseline																
Synth. Meth.	Fault-Free Base-line	10% Error			25% Error			50% Error			75% Error			90% Error		
		EPR	Synth. (ms)		EPR	Synth. (ms)		EPR	Synth. (ms)		EPR	Synth. (ms)		EPR	Synth. (ms)	
			PE	T		PE	T		PE	T		PE	T		PE	T
INT	89.1	0.5	2.6	1.3	0.8	4.3	3.4	3.5	2.9	10.0	8.0	3.0	23.9	20.6	3.1	62.8
ORS	72.0	0.5	3.8	1.9	0.8	3.8	3.0	3.5	3.4	11.9	8.0	3.8	30.4	20.6	3.6	74.9

Table 2-4: Recovery synthesis time (averaged over 10 runs) for a 2-level protein assay with varying percentages of error for split operations. Results show the average number of errors per run (EPR), average synthesis time per error (PE), and average total synthesis (T). Results are given for the online re-synthesis (ORS) model, and our interpreter (INT) with control-flow and virtual topology.

Average Recovery Assay Runtime (Schedule + Route Length) With Fault-Free Baseline						
Synthesis Method	Assay Runtime (s)					
	Fault-Free Baseline	10% Error	25% Error	50% Error	75% Error	90% Error
INT (Control Flow)	134.89	5.17	8.30	33.03	74.38	181.82
ORS (Re-synthesis)	118.36	10.11	14.65	61.94	170.30	406.48

Table 2-5: Average recovery assay runtime (averaged over 10 runs) showing the average amount of time (schedule and route length) added to the assay by errors for a 2-level protein assay with varying percentages of error for split operations. Results are shown for the online re-synthesis (ORS) model, and our interpreter (INT) with control-flow and virtual topology.

Table 2-5 shows the baseline assay execution time for a fault-free run and the average additional overhead incurred to re-execute operations for varying error rates. Although ORS produces a better fault-free result, INT adds much less recovery time to the assay because it does a merge and re-split instead of executing more complicated recovery operations [7]. Another concern, which we did not explicitly model, is that ORS may recursively encounter further errors when re-executing recovery operations that include splits themselves, that were originally successful in the original run. This would further add to the assay execution time. This experiment demonstrates that interpretation can seamlessly address reliability challenges that arise due to operation variability in DMFBs.

2.6.2 - Experiment #2: In-vitro Diagnostics

In-vitro diagnostics is a common microfluidic application, where four human physiological fluids (plasma, serum, urine, and saliva) are assayed for glucose, lactate, pyruvate, and glutamate measurements to identify metabolic disorders [75].

The in-vitro assay mixes each of the four samples with each of the four reagents and then transports the 16 resultant droplets to optical detectors for measurement. These 16 mixes can be part of the same assay, and performed concurrently, as shown in **Figure 2-16(a)**. Our BioCoder implementation of this assay runs in 47.93s in our simulator.

It is also possible to rewrite the assay using our interpretation engine and virtual topology to preserve reagents. **Figure 2-16(b)** shows a CFG created by BioCoder that is composed of four assays, each of which performs four mix operations where a human sample is mixed with a single reaction; **Figure 2-17** shows the BioCoder specification. The four assays are executed sequentially, and the protocol stops as soon as the first positive reading occurs.

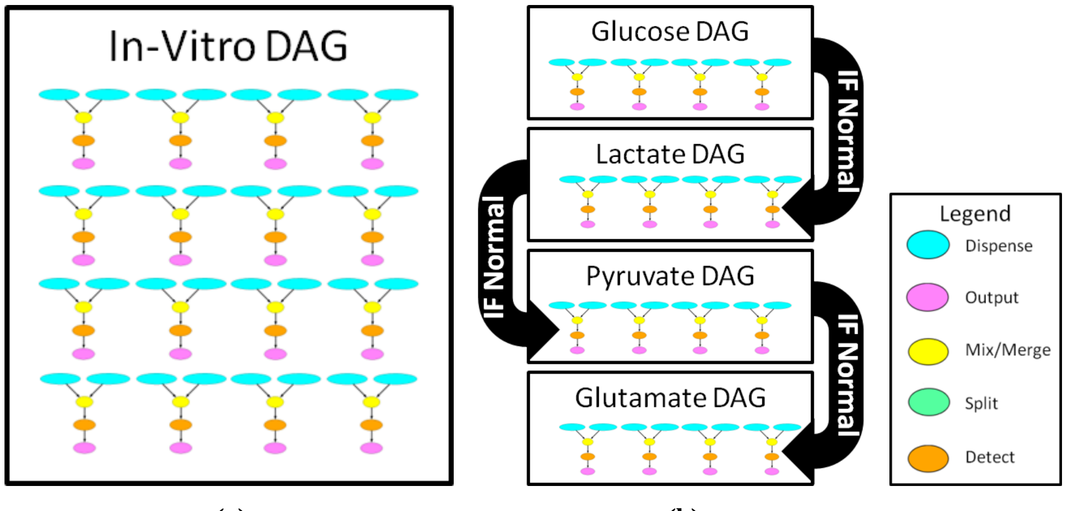


Figure 2-16: (a) Parallel and (b) sequential CFG implementations of the in-vitro diagnostics assay.

Our optical sensor returns a random probability in the range [0, 1), and we assume that a reading is irregular (i.e. positive) if the value returned is greater than some various health thresholds. **Table 2-6** shows the average results (over 10 runs) for the thresholds 75%, 90%, 95% and 99%, where a higher threshold represents a generally healthier patient. Tests run at the 75% and 90% thresholds use less time and reagents. Although the 95% and 99% tests take more time, they do use fewer reagents, which may be preferable in many contexts. **Figure 2-18** reports the simulator output for a particular run with a 99% threshold; in this example, all four DAGs were executed and no tests were determined to be irregular.

Average Sequential InVitro Completion Time/Sample Usage			
Health/Pass Rate	Completion Time (s)	Comparison To Parallel InVitro	
		Time Saving (s)	% Reagent Usage
75%	28.04	19.90	32.5
90%	36.52	11.42	42.5
95%	57.72	-9.79	67.5
99%	78.86	-30.93	92.5

Table 2-6: Average sequential InVitro completion time and sample usage compared to the parallel InVitro implementation. Averages were found over 10 runs for each health/pass rate.

```
for (int r = 0; r < numReagents; r++)
{
   BioCoder * bio = BioSys->addBioCoder();
   for (int s = 0; s < numSamples; s++)
   {
      Container tube = bio->new_container(STERILE_MICROFUGE_TUBE2ML);
      Fluid S = bio->new_fluid(sampleNames[s] ,bio->vol(100,ML));
      Fluid R = bio->new_fluid(reagentNames[r] ,bio->vol(100,ML));

      if (s == 0 && r == 0)
         bio->first_step();
      else
         bio->next_step();

      bio->measure_fluid(S, bio->vol(10, UL), tube);
      bio->measure_fluid(R, bio->vol(10, UL), tube);
      bio->vortex(tube, bio->time(sampleMixTimes[s], SECS));

      bio->next_step();
      detects.push_back(bio->measure_fluorescence(tube, bio->time(sampleMixTimes[s], SECS)));

      bio->next_step();
      bio->drain(tube, "output");
   }
   bio->end_protocol();
   assays.push_back(bio);
}
// Conditional Groups
for (int r = 0; r < numReagents-1; r++)
{
   int i = numSamples * r;
   BioConditionalGroup *bcg = BioSys->addBioCondGroup();
   BioExpression *outer = new BioExpression(OP_AND);
   outer->addOperand(new BioExpression(detects[i], OP_LT, 0.99));
   outer->addOperand(new BioExpression(detects[i+1], OP_LT, 0.99));
   outer->addOperand(new BioExpression(detects[i+2], OP_LT, 0.99));
   outer->addOperand(new BioExpression(detects[i+3], OP_LT, 0.99));
   BioCondition *bc = bcg->addNewCondition(outer, assays[r+1]);
}
```

Figure 2-17: BioCoder specification of the sequential in-vitro assay for a 99% threshold.

```
SYSTEM (0s): Executing CFG 0
SYSTEM (0s): Executing DAG1(Glucose) on DMFB 1
SYSTEM (21.52s): DAG1(Glucose) complete
SYSTEM (21.53s): Evaluating Condition Dependencies - DAG1(Glucose)
IF ((DAG1_d1_READ = 0.29 < 0.99) AND (DAG1_d2_READ = 0.28 < 0.99)
   AND (DAG1_d3_READ = 0.45 < 0.99) AND (DAG1_d4_READ = 0.90 < 0.99))
        TRUE - Branching to DAG2(Lactate)

SYSTEM (21.53s): Executing DAG2(Lactate) on DMFB 1
SYSTEM (42.77s): DAG2(Lactate) complete
SYSTEM (42.78s): Evaluating Condition Dependencies - DAG2(Lactate)
IF ((DAG2_d1_READ = 0.86 < 0.99) AND (DAG2_d2_READ = 0.98 < 0.99)
   AND (DAG2_d3_READ = 0.60 < 0.99) AND (DAG2_d4_READ = 0.94 < 0.99))
        TRUE - Branching to DAG3(Pyruvate)

SYSTEM (42.78s): Executing DAG3(Pyruvate) on DMFB 1
SYSTEM (64.13s): DAG3(Pyruvate) complete
SYSTEM (64.15s): Evaluating Condition Dependencies - DAG3(Pyruvate)
IF ((DAG3_d1_READ = 0.29 < 0.99) AND (DAG3_d2_READ = 0.83 < 0.99)
   AND (DAG3_d3_READ = 0.97 < 0.99) AND (DAG3_d4_READ = 0.31 < 0.99))
        TRUE - Branching to DAG4(Glutamate)

SYSTEM (64.15s): Executing DAG4(Glutamate) on DMFB 1
SYSTEM (85.33s): DAG4(Glutamate) complete
SYSTEM (85.61s): CFG 0 complete (all droplets off system)

____________DMFB 1____________
8560 cycles
85.6 seconds runtime at 100Hz frequency
Dispenses=32, Mixes=16, Splits=0, Heats=0, Detects=16, Outputs=16
```

Figure 2-18: Simulator output for a particular run of the sequential in-vitro assay (99% threshold) detailing the times at which each DAG in the CFG begins and ends. The output also shows the IF-ELSE control-flow statements that are executed, including the sensor readings, boolean expression evaluation and resultant branch. Finally, the overall runtime of the assay is shown at the bottom, along with the number of each type of operation that was executed.

2.6.3 - EXPERIMENT #3: BASELINE ASSAYS

Finally, we perform a standard set of benchmark assays commonly reported in literature. The first assay is the mixing tree portion of a Polymerase Chain Reaction (PCR), which is used for exponential DNA amplification in molecular biology. In-vitro diagnostics were highlighted in the last section; here, we run five common in-vitro configurations with different combinations of samples and reagents [65][75]. Finally, we run a colorimetric protein assay based on the Bradford reaction [75]. DAGs for the PCR and protein assays, as well as the 5th in-vitro assay (4 samples, 4 reagents), are seen in **Figure 2-19**. Each of the seven assays represents a DAG with no control flow.

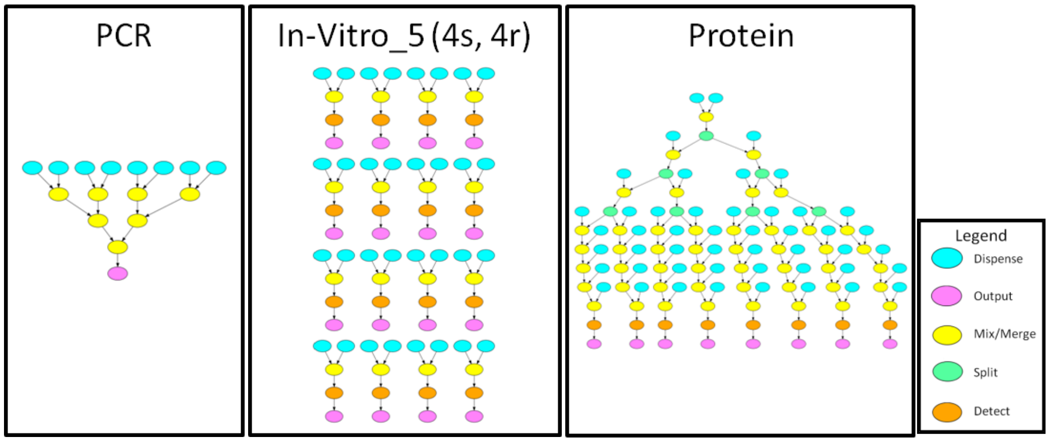


Figure 2-19: PCR, In-Vitro and Protein DAG specifications (see the Appendix for more details).

All assays have dispense times of 2s and other operation timings are as detailed in the **Appendix**. We chose the 2x4 mixer (3s) for all PCR mixes. For the protein assay, we chose the 2x4 mixer (3s) and 2x4 diluter (5s) for all mix and dilution operations, respectively; because we did not have an explicit dilution operation implemented, we divided the dilution operations into a mix node followed by a split operation that consumed a total of 5s. All operation timings and sample/reagent configurations for the in-vitro diagnostic family of assays are taken from Table I of ref. [75].

Here, we compare the performance of the interpreter and virtual topology (INT) with a long-running static compiler (LRSC), with no virtual restrictions on placement and routing; the compiler cannot handle assays featuring control flow, and therefore, could not produce results for Experiments #1 and #2, as discussed in the preceding subsections. We compare the schedule and route quality between LRSC and INT, while simultaneously showing the tradeoff that is made between computation time and assay time (schedule and route length). INT was run on a 20x20 DMFB (2x2 tile array); LRSC was able to fit the assays onto a smaller 15x19 DMFB. As in other works, we assume a droplet actuation frequency of 100 Hz [88].

For our LRSC, we chose a genetic scheduling algorithm [75], a simulated annealing-based placer [76], and a fast maze router [66] to compile the DAGs onto a DMFB. Scheduling is the most important synthesis task since it determines the bulk of the assay time (scheduling units are on the order of seconds; routing units are on the order of milliseconds). We selected a genetic scheduler because it tends to produce high quality results in a relatively reasonable amount of time. Optimal scheduling based on integer linear programming (ILP) [75] is also possible, but may run for days or weeks on DAGs of non-trivial size. Placement does not significantly affect assay performance; however, it is an important step when targeting very small DMFB architectures. We chose a placement algorithm based on iterative improvement for similar reasons. The choice of router is far less important, as prior work has noted that routing times do not significantly impact the assay completion time [75]. We chose to implement a maze router [31][66] primarily due to ease of implementation. To the best of our knowledge, the literature on routing lacks a comprehensive comparison among all previously published routing algorithms, so we do not claim that the maze router is either the fastest or best performing.

Table 2-7 and **Table 2-8** show the results of LRSC and INT, respectively, including computation times and assay times (the actual time spent executing the assay, i.e. schedule length and route lengths). The results show that LRSC can complete an assay, from first dispense to last output, from 3s to 41.5s faster than INT. Thus, overall, LRSC's scheduling-routing solutions (SL+RL) are better than INT's solutions; however, in an online setting when synthesis/interpretation time will be experienced by the end-user who is waiting for the computations to complete, these gains can be considered negligible when compared to the increase in computation times from 2.8s to 22h.

Long-Running Static Compiler (LRSC): Genetic Scheduler→Simulated Annealing Placer→Roy's Maze Router								
Benchmark	Scheduling (s)		Placement (s)	Router (s)		Total Synthesis (s)		
	CT	SL	CT	CT	RL	CT	SL+RL	CT+SL+RL
PCR	2.621	1	0.200	0.002	0.780	2.823	11.780	14.603
InVitro1	4.475	14	12.843	0.002	1.350	17.320	15.350	32.670
InVitro2	8.122	16	141.177	0.004	1.800	149.303	17.800	167.103
InVitro3	13.156	18	506.767	0.010	2.070	519.933	20.070	540.003
InVitro4	22.376	22	3,317.571	0.007	2.340	3,339.954	24.340	3,364.294
InVitro5	39.410	30	1,399.936	0.009	3.420	1,439.355	33.420	1,472.775
Protein	22.334	109	79,531.695	0.032	12.120	79,554.061	121.120	79,675.181

Table 2-7: Static compiler synthesis results for 7 deterministic benchmarks showing algorithmic computation times (CT), the computed schedule lengths (SL) and computed route lengths (RL).

Online Interpreter (INT): List Scheduler→Module Binding Placer→XY Router								
Benchmark	Scheduling (s)		Placement (s)	Router (s)		Total Synthesis (s)		
	CT	SL	CT	CT	RL	CT	SL+RL	CT+SL+RL
PCR	0.001	1	0.001	0.009	0.560	0.011	11.560	11.571
InVitro1	0.001	8	0.002	0.015	1.330	0.018	19.330	19.348
InVitro2	0.002	19	0.003	0.022	1.860	0.027	20.860	20.887
InVitro3	0.005	29	0.006	0.034	2.540	0.045	31.540	31.585
InVitro4	0.008	34	0.009	0.045	3.330	0.062	37.330	37.392
InVitro5	0.015	44	0.016	0.058	4.110	0.089	48.110	48.199
Protein	0.014	154	0.017	0.150	8.670	0.181	162.670	162.851

Table 2-8: Online Interpreter (using the virtual topology) synthesis results for 7 deterministic benchmarks showing algorithmic computation times (CT), computed schedule lengths (SL) and computed route lengths (RL).

Furthermore, although LRSC's schedules and routing times (SL+RL) are shorter by 3s to 41.5s, the interpreter can make up this ground in several ways and still maintain its fast synthesis times and remain suitable for online synthesis. One way is to utilize new, fast schedulers that are targeted at certain types of assays. Two recent examples are path scheduling [33] and force-directed list scheduling [61]. List scheduling, path scheduling and force-directed list scheduling all run quickly, making it possible to run all three, without great concern to computation time, and take the best schedule.

Table 2-9 shows the results of genetic, list, force-directed list and path schedulers on *ProteinSplit1-4* (assays with 1-4 levels of splits). Again, all results are on the low-powered Atom processor. The results show that although the genetic algorithm generally produces the best schedule, its computation time dwarfs the savings and makes it impractical for online scheduling. On the other hand, although the computation time for force-directed list scheduling begins to grow large as the assay size increases, the list, path and force-directed list schedulers all claim superior results for at least one benchmark. For the sake of simplicity and space limitations, we only include results for INT with list scheduling.

Computation Time (CT) and Schedule Length (SL) For Various Schedulers On Large Assays								
Benchmark	Genetic		List		Force-Directed List		Path	
	CT (s)	SL (s)	CT (s)	SL (s)	CT (s)	SL (s)	CT (s)	SL (s)
ProteinSplit 1	26.227	72	0.019	72	0.108	72	**0.009**	**73**
ProteinSplit 2	70.887	107	**0.054**	**107**	0.487	108	0.021	111
ProteinSplit 3	199.358	180	0.135	198	**2.132**	**182**	0.048	187
ProteinSplit 4	677.353	358	0.338	390	10.284	367	**0.105**	**339**

Table 2-9: Scheduling results for various scheduling methods for large ProteinSplit assays on a 20x20 DMFB with four work modules, each equipped with a detector. Scheduling computation times (CT) and the computed schedule lengths (SL) are shown. The best overall scheduler for each benchmark is emboldened.

In this particular case, however, the inferior schedule quality of the interpreter is mostly due to a lack of resources. On a 20x20 DMFB, the interpreter can perform 4 concurrent mix operations, one in each of its 4 modules. The static compiler, on the other hand, has sufficient room to comfortably fit 6 2x4 mixers on a smaller 15x19 DMFB. By increasing the size of the DMFB, the interpreter can close the scheduling gap. For example, we ran the largest assay, the protein assay, on a 30x20 DMFB that could fit 6 mixers and re-ran the interpreter; list-scheduler produced a schedule of 111s, only 2 seconds longer than the compiler's schedule length. With new, highly-scalable DMFBs being developed [34][60], increasing the array size does not increase the complexity and cost of a DMFB as it did in the past; trading DMFB size for algorithmic simplicity is becoming an easy tradeoff to make.

2.7 - CONCLUSION

This chapter introduced extensions to the BioCoder language to allow the specification of biochemical protocols that feature control flow, and described the design and implementation of a software interpreter that executes assays dynamically, as opposed to the prior state-of-the-art where assays were compiled statically. The key innovation that facilitates interpretation is the imposition of a virtual topology on top of a DMFB which facilitates deadlock-free droplet routing through the simple adaptation of deadlock-free routing algorithms for 2D computer mesh networks. This relieves the interpreter of the need to explicitly schedule and place assay operations on the DMFB and route droplets; instead, the interpreter binds assay operations to pre-positioned work modules, and relies on the DTP to deliver the droplets to their locations in a timely fashion. Experiment #1 (fault-tolerant splitting) and Experiment #2 (sequential in-vitro diagnostics) validate the ability of the interpreter to execute assays that feature control flow. Experiment #3 demonstrates that the computational overhead of the interpreter is far less than that of static compilation and re-compilation methods.

The drawback of this approach is area overhead: the streets, rotaries, and separation between them occupy a significant number of cells that could otherwise be used to execute more concurrent operations. It is clearly possible to pack the work modules in more tightly, thereby replacing the droplet routing protocol with something simpler, yet ineffective (e.g., route one droplet at a time), or a more complicated routing algorithm with greater complexity. In particular, DMFBs are often I/O limited, especially for portable point-of-care devices. The virtual topology introduced in this chapter is compatible with direct-addressing and active matrix addressing DMFBs (see **'Section 1.2 - DMFB Device Technology Overview'**); our approach, however, is not compatible with pin-constrained DMFBs, which use fewer control pins, but sacrifice flexibility in order to do so. In contrast, the interpreter can execute *any* assay on the DMFB that satisfies the resource constraints of the device. On the other hand, the cells shown in white in **Figure 2-4** do not require electrodes, as they are not used; thus, the number of control pins in a direct-addressing implementation of our interpreter can be reduced as well.

CHAPTER 3 AN EFFICIENT VIRTUAL TOPOLOGY

3.1 - INTRODUCTION

New features in control flow, programmability and interpretation (such as those introduced in the previous chapter) will revolutionize microfluidic applications and promise to broaden the usefulness of DMFBs. As described in previous chapters, these features will further the importance of techniques that help speedup synthesis methods for an online environment. In **CHAPTER 2**, we presented a virtual topology and briefly described how it simplified the entire synthesis flow and helped enable online interpretation; however, the main focus of **CHAPTER 2** was to present the concepts of interpretation, microfluidic programmability and control flow. In this chapter, we introduce a new synthesis flow and an optimized virtual topology, which uses less space than the topology described in **CHAPTER 2**. Furthermore, in this chapter, we thoroughly examine the algorithmic details we employ to achieve fast online synthesis and show how each method in our synthesis flow works together to achieve a valid solution.

In general, the objective of scheduling, placement, and routing is to minimize assay execution time. In addition to the basic requirements for scheduling, placement and routing (see **'Section 1.3 - High-Level Assay Synthesis Overview'**), 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 re-synthesis 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., **Figure 1-7(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.

3.1.1 - CONTRIBUTION

We present an online synthesis flow that can interpret assays and map them onto a fully-addressable or active-matrix DMFB [60] in milliseconds, making it appropriate for both offline and online synthesis. Our key contribution is a compact virtual topology that defines distinct regions for module placement and droplet routing. With our topology in mind, we present several necessary constraints and apply them to list scheduling [75] and path scheduling [33] to quickly produce schedules. Placement, which has been solved in the past by iterative improvement algorithms [76][87] or integer linear programming (ILP) [45], is simplified to a binding problem, which can be solved efficiently in polynomial-time. We introduce two binding solutions in detail; 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. The placement defined by the virtual topology provides dedicated routing cells which ease the router's job. We simplify an existing router [66] to compute droplet paths very quickly.

The overall 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, in lieu of traditional placement, can significantly speed up the synthesis process without notably lengthening assay execution time. We demonstrate how our new virtual topology can be leveraged to reduce algorithmic runtimes and guarantee routability by addressing previously-unresolved issues such as module synchronization and poor module placement. The results show that our synthesis flow always yields a successful solution, while prior methods result in synthesis failures. Finally, we show several variations of the virtual topology and present experimental results demonstrating best-design practices. The online synthesis flow presented in this chapter is compatible with individually addressable, cross-referencing, and active-matrix addressing DMFBs.

3.2 - RELATED WORK

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

3.2.1 - SCHEDULING

Su and Chakrabarty 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 [75]. 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 [65] and Maftei's tabu search scheduler [56] are iterative improvement algorithms which spend anywhere from 4 seconds to 1 hour computing schedules. We chose list scheduling as the base scheduler for our framework, but other fast schedulers being developed now [33][61] or in the future could be used as well.

3.2.2 - 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 [76] [77][83][87] use simulated annealing, which run in minutes or tens of seconds; in contrast, our online flow completes in tens of milliseconds.

Griffith et al. [26] 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.

3.2.3 - ROUTING

Böhringer [11] modeled droplet routing as an A* search, similar to path planning in robotics, achieving an optimal-length solution, when routable. Su et al. route droplets sequentially and redo placement when routing fails [78]. BioRoute [88] uses a min-cost max-flow algorithm to compute several routes at once, followed by negotiation-based detailed routing. Cho and Pan [16] route droplets one-by-one and sort them based on a *bypassability* metric; if a deadlock occurs, droplets are moved to *concession zones* to break the deadlock. Huang and Ho [38] 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 [11][16][78]. BioRoute [88] and Huang's algorithm [38] both report runtimes below 1s on a desktop PC. The router used in our online flow, a modified version of Roy's maze router [66], achieves comparable runtimes, while achieving deadlock freedom.

3.2.4 - 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 [52][76][77][83][87], 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.

3.3 - Virtual Topology

Our online interpreter utilizes a virtual topology, as seen in **Figure 3-1**, 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 2D area covering an electrode. The virtual topology shows regularly spaced modules (3x3 squares of cells) where basic droplet operations (i.e. merge, mix, split, 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 [78], 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 **Figure 3-1**) are on the perimeter of the chip.

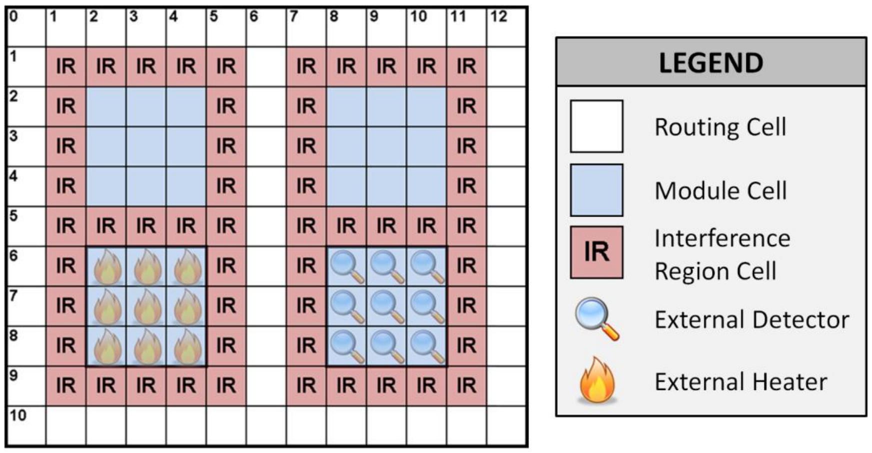


Figure 3-1: Virtual topology imposed onto a DMFB.

3.3.1 - 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 Figure 3-2(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. Figure 3-2(b) and Figure 3-2(c) show that modules can be elongated along the X- or Y-axis to accommodate larger 2x4 mixers, often used in literature [62][75].

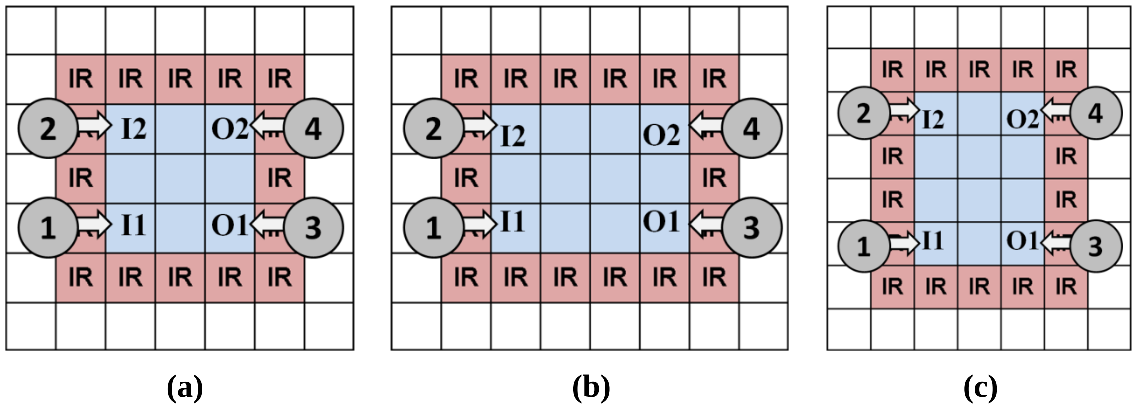


Figure 3-2: The entrance cells (I1/I2) and exit cells (O1/O2) of (a) a 3x3 module, (b) a 4x3 module and (c) a 3x4 module.

As seen in **Figure 3-3**, 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.

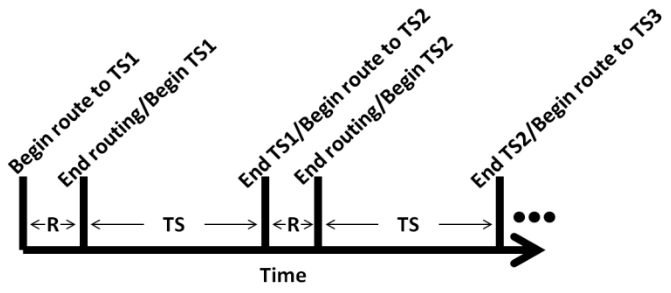


Figure 3-3: An assay time-line showing that each fixed time-step (TS) is interleaved with a variable-length routing phase (R).

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 **Figure 3-2(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 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 inter-module deadlocks because there is always an open spot at the destination module’s entrances for every incoming droplet at every module.

Figure 3-4 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 **Figure 3-4**); this eliminates the need for a droplet to exit and then re-enter the same module.

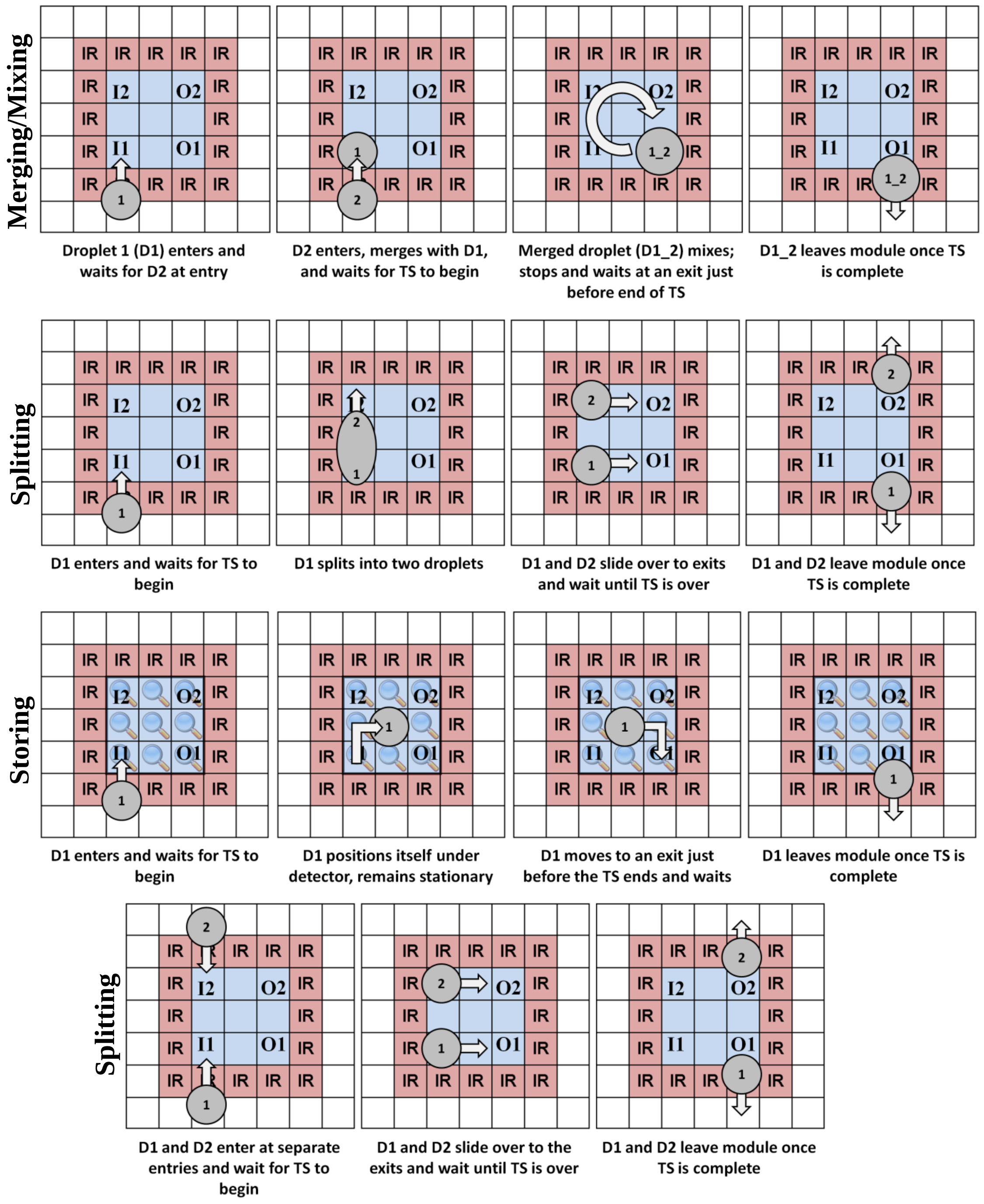

Merging/Mixing
Droplet 1 (D1) enters and waits for D2 at entry
D2 enters, merges with D1, and waits for TS to begin
Merged droplet (D1_2) mixes; stops and waits at an exit just before end of TS
D1_2 leaves module once TS is complete
Splitting
D1 enters and waits for TS to begin
D1 splits into two droplets
D1 and D2 slide over to exits and wait until TS is over
D1 and D2 leave module once TS is complete
Storing
D1 enters and waits for TS to begin
D1 positions itself under detector, remains stationary
D1 moves to an exit just before the TS ends and waits
D1 leaves module once TS is complete
Splitting
D1 and D2 enter at separate entries and wait for TS to begin
D1 and D2 slide over to the exits and wait until TS is over
D1 and D2 leave module once TS is complete
IR
I1
I2
O1
O2
1
2
1_2

Figure 3-4: Intra-module droplet processing/routing for microfluidic operations.

3.4 - Fast Online Synthesis

In this section, we show how the virtual topology presented in **'Section 3.3 - Virtual Topology'** can be leveraged to create fast online synthesis methods for scheduling, placement and routing.

3.4.1 - 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 **Equation 3.1**. 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 (see **Figure 3-1**).

$$\left\lfloor \frac{(a_x - 1)}{(m_x + 3)} \right\rfloor \times \left\lfloor \frac{(a_y - 3)}{(m_y + 1)} \right\rfloor = N_m \qquad (3.1)$$

Once the virtual topology is placed, modules with external devices above their cells are considered to be *special modules* (e.g. detect module, 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, etc.), and satisfies the following condition:

$$\sum_{i=1}^{numModTypes} availMods[i] = N_m \quad (3.2)$$

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 2 entrance and 2 exit cells, a module can store 2 droplets during a time-step (i.e. $s_m = 2$). Consider **Figure 3-5(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 module, allowing the assay to continue. However, if all modules are at maximum capacity (**Figure 3-5(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 number of droplets permitted on the DMFB during any time-step (d_{max}) as follows:

$$d_{max} = (N_m \times s_m) - 1 \quad (3.3)$$

In **'Section 3.5 - Experiments'**, we have successfully applied these constraints to two fast schedulers: list scheduling (LS) [75] and path scheduling (PS) [33]. 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 [56][65][75] or optimal algorithms based on integer linear programming [75]; 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.

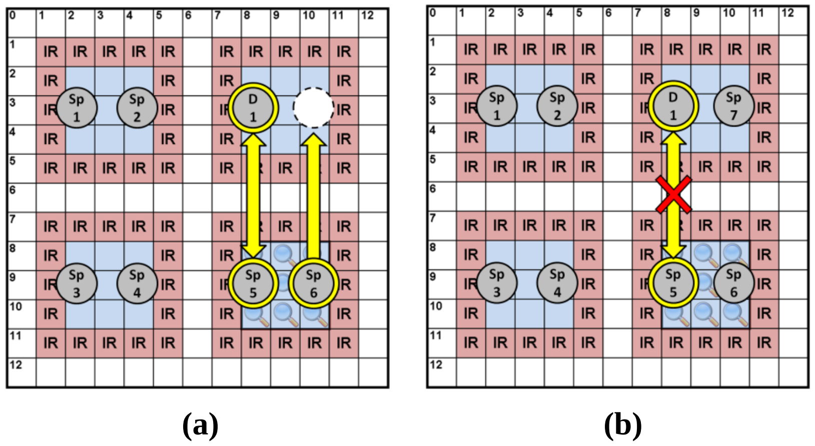


Figure 3-5: Two DMFB scenarios with droplets that are going to be split (Sp) or detected (D) during the next time-step. In (a), Sp6 can move and occupy the open space in another module, allowing D1 and Sp5 to swap so D1 can be detected in the detect-module. In (b), there is no way to isolate a single droplet and since no droplets will be mixed next time-step, the assay cannot continue.

3.4.2 - PLACEMENT

DMFB placement is NP-complete [76]; the virtual topology limits the reconfigurable capabilities of the DMFB by pre-placing the location of modules. In our framework, operations are bound to pre-placed 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. In the following sub-sections, we present two binding algorithms: a left-edge based greedy algorithm and a more-intelligent, path-based algorithm.

3.4.2.1 - Left-Edge Binding Algorithm

Our binder is based on the left-edge algorithm, which has been used in the past for register allocation and track assignment in channel routing [44]. The left-edge algorithm has an $O(|V|^2)$ time complexity, where $|V|$ is the number of assay operations.

Figure 3-6 shows how the left-edge algorithm binds operations to modules for the DMFB shown in **Figure 3-1**. **Figure 3-7** provides pseudocode for our binding algorithm and can be followed (up to *Line 20*) in the example. A *fixed-module bin* is created (*Line 2*) for each module in the virtual topology; $N_m = 4$ for this example. Then, each operation is placed into an *operation-bin* based on the module-type it was assigned during scheduling (*Lines 9-11*). Next, the operations in each bin are sorted in ascending order based on their start time (*Lines 13*).

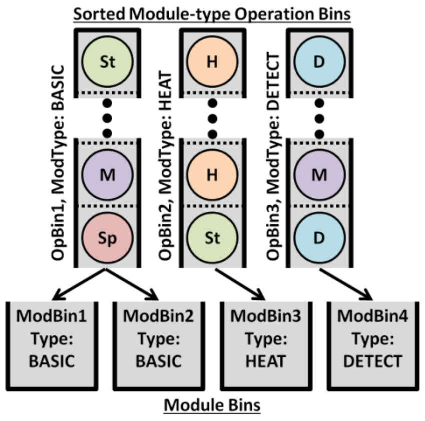


Figure 3-6: Illustration of the left-edge binding solution (reproduced from CHAPTER 2 for convenience).

```
1  Given a scheduled sequencing graph: $G = (V, E)$
2  Given a list of fixed module schedules: $fixedMods : |fixedMods| = N_m$
3
4  Operations by module-type: $opsByModType[numModTypes] = \emptyset$
5  Storage operations: $storageOps = V.storageOps$
6  Storage-holder operations: $sHoldsOps = V.storageHolderOps$
7
8  // Put operations in bins by module-type
9  for ($\forall v : v \in V$)
10   if (Type($v$) $\neq$ (storage OR input OR output))
11     Add $v$ to $opsByModType$[Type($v$)];
12
13 Sort $storageOps$and all lists in $opsByModType$[ ], ascending by start time;
14 Sort $sHoldsOps$, first by fixed-module location, then ascending by start time;
15
16 // Left-edge bind the modules
17 for ($\forall m : m \in fixedMods$)
18   for ($\forall o : o \in opsByModType$[ModType($m$)])
19     if ($o.start > m.LastOp.end$)
20       Add $o$ to end of $m$, remove $o$ from $opsByModType$[ModType($m$)];
21
22 // Bind storage into storage-holders
23 for ($\forall s : s \in storageOps$)
24   int $curEnd = 0$;
25   for ($\forall sh : sh \in sHoldsOps$)
26     if (Status($s$) $\neq$ bound AND $s.start = sh.start$ AND $sh.size < s_m$)
27       $BindStorageToHolder(s, sh)$;
28       Set $curEnd = sh.end$;
29     else if (Status($s$) = bound)
30       if ($curEnd = sh.start$ AND $sh.size < s_m$ $AND$ $s.Mod = sh.Mod$)
31         $BindStorageToHolder(s, sh)$;
32         Set $curEnd = sh.end$;
33      else
34          Split $s$ at $curEnd$ to form $s2$, add $s2$ to front of $storageOps$;
35         Return to outer $\forall s$ for loop;
36
37     if ($curEnd = s.end$)
38       Remove $s$ from $storageOps$ and return to outer $\forall s$ for loop;
39   end $\forall sh$ for
40 end $\forall s$ for
```

Figure 3-7: Pseudocode for our left-edge-based binding algorithm. NOTE: Only module binding is shown; input and output binding is left out for brevity.

Lines 17-20 perform the actual binding process. The first module, *ModBin1*, finds the operation bin matching its module-type (*OpBin1*). The binding method then examines each operation in *OpBin1*, in order of start time, to ensure it will not conflict with any other operation already assigned to *OpBin1* (i.e. that the start time of the operation in question is later than the end time of the last operation assigned to *ModBin1*). If an operation is placed in *ModBin1*, it is removed from *OpBin1*; otherwise, it remains in *OpBin1* to be bound to another module. This process is repeated for the remaining three bins; when the last module (*ModBin4*) has examined its corresponding operation bin (*OpBin3*), all operations will have been bound to a module.

Lines 23-40 describe how storage operations are bound. Note that storage operations are not placed into the operation bins in *Lines 9-11*, and thus, were not bound to specific modules; however, the storage-holder nodes were bound in *Lines 17-20*. As mentioned in **'Section 3.4.1 - Scheduling'**, a storage-holder node was created in the scheduling phase for each set of s_m droplets being stored each time-step (i.e. $\lceil st_t/s_m \rceil$ 1-time-step storage-holder nodes are created at time-step t, where st_t is the number of droplets being stored at t). Storage-holder nodes are created so that storage nodes can be broken into a number of smaller, contiguous storage nodes to prevent rare modules (e.g. detect modules) from being tied up as storage.

Each storage node is examined and assigned to one or more storage-holder nodes (*Lines 23-40*) since storage-holders are always one time-step. If storage node s has not yet been bound (*Line 26*), a suitable storage-holder is one that shares the same starting time as s and is not yet storing its maximum capacity of droplets (s_m). If this criteria is met by a storage-holder node sh, then $BindStorageToHolder(s, sh)$ (*Line 27*) marks s as bound, assigns it to the same module sh is bound to and updates the number of droplets sh is storing. A variable named $curEnd$ is then updated to keep track of how much of s has been bound (for storage nodes larger than one time-step).

Once s has been initially bound (i.e. the first time-step of s), the algorithm attempts to bind the rest of the storage node to the same module location (*Lines 30-32*). It is possible to do this by iterating to (*Line 25*) and examining the next storage-holder in the storage-holders list since the storage-holders were sorted by their module-location and start-time, as seen in *Line 14*. If the next sh shares the same module as s, has the same start-time as our running end ($curEnd$) and is not at maximum capacity, we bind s to sh to update $sh's$ storage count and update $curEnd$. If the condition in *Line 30* cannot be met, s is split at $curEnd$ because it cannot be stored in the same module any longer. The new module is inserted into the storage list to be bound later. If $curEnd$ equals $s's$ end time (*Line 37*), then s has been completely bound and the algorithm can return to *Line 23* to bind the next storage node.

3.4.2.2 - PATH-BASED BINDING ALGORITHM

In this section, we present a more-intelligent, path-based binding algorithm which 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 [79]. 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.

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, but instead, binds each operation to the first available module it finds with a matching module-type. Path binder takes parent/child module locations into consideration (reducing routing distances) and although it does use left-edge binding, performs pre-processing to reduce the graph, which eases algorithmic runtimes.

Figure 3-8(a) shows a simple sequencing graph with 7 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 **Figure 3-8(a)** also denote droplets needing to be routed. **Figure 3-8(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 produce shorter droplet routes because it will have less droplets to route. Furthermore, **Figure 3-8(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.

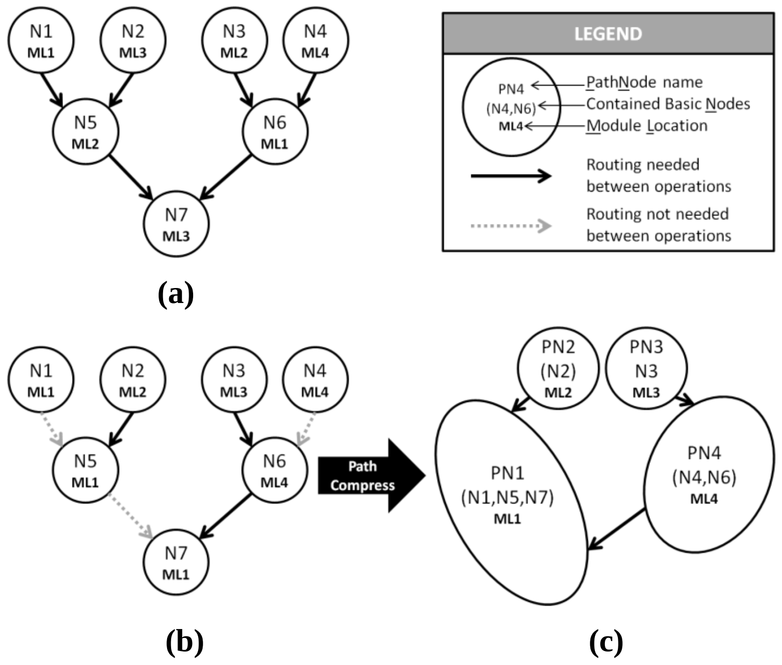


Figure 3-8: (a) Randomly-bound sequencing graph for a simple assay requiring 6 droplet routes; (b) sequencing graph with intelligent module selection requiring only 3 droplet routes allows for (c) the sequencing graph to be compressed.

Figure 3-9 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) [32]. 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.

```
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: $G_p = (V_p, E_p) = \emptyset$
9   New List of PathNodes: $pathLeaders = \emptyset$
10  Operations by module-type: $pathOpsByModType[numModTypes] = \emptyset$
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: $storageByModLoc[numMods] = \emptyset$;
25  $storageByModLoc[\,]$ = SelectModuleLocations($storageOps$) ;
26  BindStorageToHolders($storageByModLoc[\,]$);
```

Figure 3-9: Pseudocode for our path-based binder.

3.4.2.2.1 - 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 **Figure 3-8(b-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.

Figure 3-10 presents pseudocode for the path compression algorithm (**Figure 3-9**, *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 5-6*. 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.

```
1   Given a list of path leaders: $pathLeaders$
2   Sequencing graph containing path leader PathNodes: $G_p = pathLeaders$
3
4   for ($\forall p : p \in pathLeaders$)
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 ($\forall uec_i : uec_i \in uec$)
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                  $uvc = \emptyset$;
32          end while
33      end $Type(n)$ if
34  end $\forall pl$ for
35  return $G_p$
```

Figure 3-10: Pseudocode for the GeneratePathCompressedGraph( ) function.

3.4.2.2.2 - Selecting Storage Module Location

Figure 3-11 presents pseudocode to show how module locations are selected for storage nodes. Given a list of storage operations, *Lines 3-12* 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*).

```
1   Given a list of storage operations: $storageOps$
2
3   while ($storageOps.size > 0$)
4       Node $sn = storageOps.$RemoveFront( );
5       New List of ModuleLocations: $potentialMods = \emptyset$;
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
```

Figure 3-11: Pseudocode for the SelectModuleLocations( ) function.

3.4.2.2.3 - 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.

Figure 3-12 presents pseudocode for the BindStorage-ToHolders(_) function (**Figure 3-9**, *Line 26*). Instead of adding further pseudocode,

Figure 3-12 contains links (a-k) to pictorial transformations (**Figure 3-13**) 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, **Figure 3-13** 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.

Figure 3-12 shows that we hold a running-start variable ($rStart$) to denote that any portion of a storage node before $rStart$ has already been bound. **Figure 3-13** shows how much of the storage node is bound in each case by examining the before/after positions of the running-start (RS).

Examining **Figure 3-12** and **Figure 3-13**, **(b-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 **Figure 3-13**, 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 **Figure 3-13(b) (Figure 3-13(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, that was originally spanned by the beginning (end) of the pre-bound h.

```
Given lists of storage nodes, sorted by module location: storageByModLoc[ ]
List of holder nodes, sorted by module location: holdersByModLoc[ ] = Ø

for each (ModuleLocation ml)
    List holders = holdersByModLoc.at(ml)
    List stores = storageByModLoc.at(ml)
    sortByStartThenEnd(stores)

    for each (s in stores)
        if (no holders in holders)
            (a)
        else // holders already exist
            int rStart = s.start // Running start for s
            while (rStart < s.end)
                Node h = holders.getNext();
                if (rStart < h.start) // Starts before h
                    if (s.end ≤ h.start) // s & h do not overlap
                        (b)
                    else // s & h overlap
                        (c)
                else if (rStart == h.start) // Starts at same time as h
                    if (s.end < h.end)  // s ends in middle of h
                        (d)
                    else // s ends at h
                        (e)
                    else // s ends after h
                        (f)
                else if (h.start < rStart < h.end) // Starts in middle of h
                    if (s.end > h.end) // s extends past h
                        (g)
                    else if (s.end == h.end) // s ends with h
                        (h)
                    else if (s.end < h.end) // s encompassed by h
                        (i)
                else if (rStart ≥ h.end AND holders.hasNext() == false)
                // rStart  starts as or after h ends AND no more holders
                    if (rStart  > s.start) // Part of s already bound
                        (j)
                    else
                        (k)
                end rStart if
            end while
        end holders if
    end stores for
end ModuleLocation for
```

Figure 3-12: Pseudocode for the BindStorageToHolders( ) function (Figure 3-9, Line 26) with references (a-k) to pictorial transformations in.

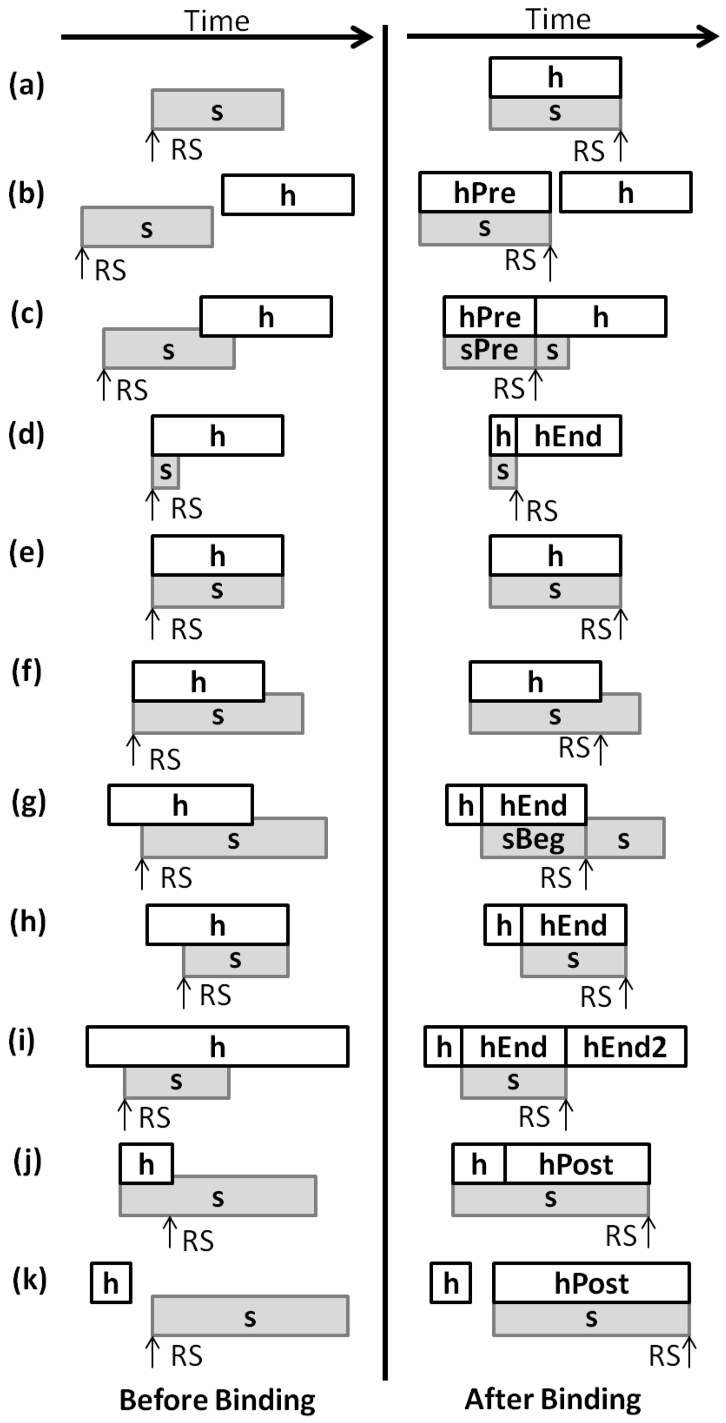


Figure 3-13: (a-k) Transformations that take place at the corresponding times in the pseudocode (see Figure 3-12). RS denotes the rStart (running start) variable. An alignment of a storage (gray box) and holder (white box) node indicate that the storage node is bound to the overlapping holder node (after binding).

3.4.3 - ROUTING

To complete the synthesis flow, we use a simplified version of an existing droplet router by Roy [66]. 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 [70] 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 that we have implemented here, by Pranab Roy et al., does not support rip-up and re-route. We chose this algorithm because it offers a good tradeoff 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 re-route 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.

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 **Figure 1-8**. Droplet routes are computed one routing sub-problem at a time. As seen in **Figure 3-3**, 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. Because of 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 [70]. 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 must avoid interference by obeying the static and dynamic constraints described in **'Section 1.3.3 - Droplet Routing'**.

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 [66]. We simplify the process by taking advantage of our virtual topology. With our module synchronization, described in **'Section 3.3.1 - Module Topology and Synchronization'**, 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 **Figure 3-14** in which droplets 1 and 2 are being routed from their sources (S1 and S2) to their target cells (T1 and T2). As seen in **Figure 3-14(a)**, 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. **Figure 3-14(b)** 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.

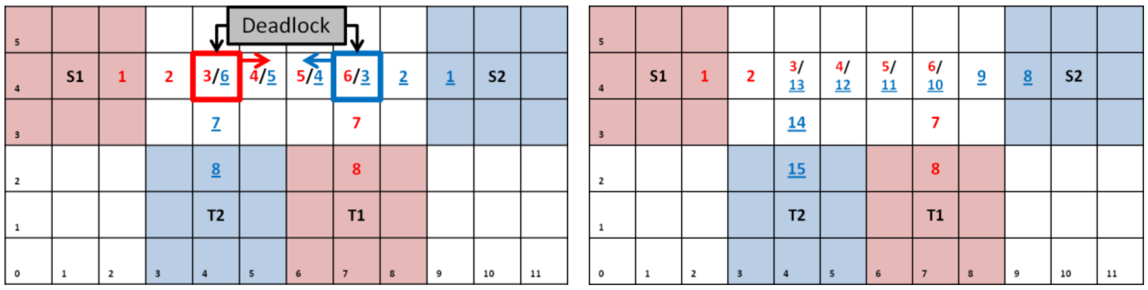


Figure 3-14: Droplets 1 and 2 are traveling from source 1 and 2 (S1/S2) to target 1 and 2 (T1/T2), respectively. The red and blue (blue also underlined for clarity) numbers are time-stamps for droplets 1 and 2, respectively); (a) shows that deadlock can occur when routes 1 and 2 are compacted and stalls are added mid-route; (b) shows that both routes are safely completed if droplet 2 stalls at its source location until droplet 1 is safely out of the way.

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 pre-route-stall compaction method only when a deadlock occurs.

3.5 - EXPERIMENTS

In this section, we present experimental results evaluating the left-edge binder, path-binder, topology spacing and comparisons to traditional, fast, chaotic placers.

3.5.1 - BENCHMARKS

We used three benchmark families: PCR, in-vitro diagnostics, and a protein assay (see **Table 3-1**), whose base DAGs are detailed in the **Appendix**; 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 4 samples with 4 reagents (e.g., up to 16 mix-and-detect operations). We use the 5 in-vitro assays, along with mixing/detection times, as listed in Table 1 of ref. [73].

We also use the protein-split benchmark, described in ref. [33], which represents the traditional protein assay with varying levels of splitting from 1 to 7 (the traditional protein has 3 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 (5s) and 4×2 mixing times (3s) 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 5s in total.

Assay Benchmarks					
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 3-1: Table of benchmarks showing the number of different operation types and dispense times.

Mixing Module Library		
Mix Module Dimensions (without interference region)	Mix Module Dimensions (with interference region)	Mixing Time
2×2	4×4	10s
4×1	3×6	5s
3×2	5×4	6s
4×2	6×4	3s

Table 3-2: Module library for mix operations for PCR and ProteinSplit assays, repeated from ref. [74] (no longer available online).

We assume a droplet actuation frequency of 100 Hz [88] and all droplet input times are assumed to be 2s in length. The *ProteinSplit* assays in Experiments 1 and 2 were all scheduled using path scheduler [33]. All assays in Experiment 3 were scheduled with list scheduler [75]. 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 (see **Table 3-2**) that would fit inside our standard 4×3 module. 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.

3.5.2 - IMPLEMENTATION DETAILS

All code was implemented in C++ using the University of California, Riverside's (UCR's) DMFB Synthesis Framework [27]. We evaluated performance on a 64-bit Windows 7 desktop PC, with 4GB of RAM and an Intel Core i7™ CPU operating at 2.8GHz. This platform represents a typical use case for a controlled laboratory setting.

3.5.3 - EXPERIMENT 1: LEFT-EDGE BINDING VS. PATH BINDING

We first compared the left-edge binder with the path binder, both described in **'Section 3.4.2 - Placement'**, on a 15W×19H DMFB with the basic topology described in **Figure 3-1** 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 pre-processing 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 3-3 shows the results for left-edge and path binding. For *ProteinSplit1-5*, the problem instances are too small to really see a difference in computation time. However, as the assays grow larger (*ProteinSplit6-7*) path binder's improvements are clearly seen since it produces a valid binding 10× faster than the left-edge binder.

Table 3-3 also shows the total length of the droplet routes generated by the router stage (described in **'Section 3.4.3 - Routing'**) 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 3.8s on the largest assay. It should also be noted that, although not seen in **Table 3-3**, the computation time for routing is decreased due to the spatial enhancements of path binder; from *ProteinSplit1-7*, the router saves from 2ms to 6.4s, respectively, further adding to the time savings when using path binder. For the remainder of this work, we use path binder as our binder of choice.

Left-Edge Binding vs. 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

Table 3-3: 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.

3.5.4 - EXPERIMENT 2:TOPOLOGY EXPLORATION

Here, we explore several topological configurations and the effects on routing. **Figure 3-15** shows three different configurations 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. **Figure 3-15(a)** shows the tightest configuration, which is the case where there are no HRCs. **Figure 3-15(b)** and **(c)** illustrate the cases where there is a single HRC between every two modules and every module, respectively. The design seen in **Figure 3-15(c)** allows for maximum routability and provides the fewest blockages (at the cost of using more space). **Figure 3-15(a)** is the tightest design (with the most blockages for routing); **Figure 3-15 (b)** presents a compromise between the two.

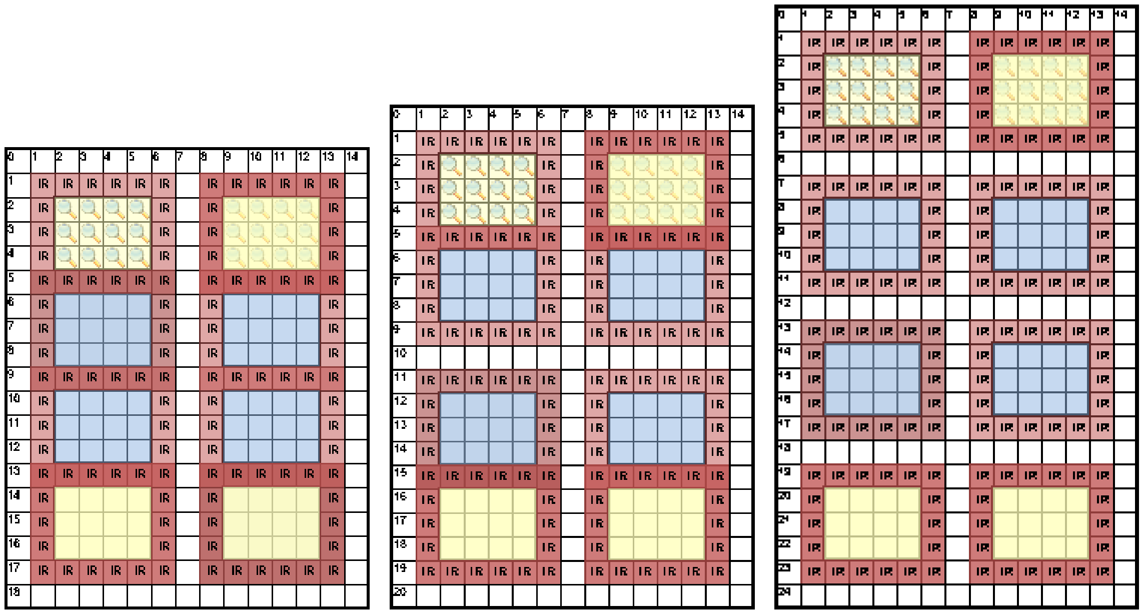


Figure 3-15: Three different topologies showing modules stacked vertically (2W×4H) with (a) no horizontal routing channels (HRC, the white cells between modules) between modules, (b) 1 HRC between every 2 modules and (c) 1 HRC between every module.

In **Table 3-4** and **Table 3-5**, we show how the topologies affect schedule length and routing times. **Table 3-4** 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 3-4**, the tight topology with no HRCs (similar to that seen in **Figure 3-15(a)**) could fit 10 modules on a 15W×23H DMFB, while the topology with one HRC between each module (similar to that seen in **Figure 3-15(c)**) could only fit 6 modules. The results are clearly seen in that, as the number of modules increases, the schedule lengths are reduced.

Table 3-5 gives results for the ProteinSplit assays when the number of resources are fixed (8 modules), in order to show the results on routing. In this case, the DMFB topologies and sizes are exactly those seen in **Figure 3-15**. 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 3-5**, 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 3.3 - Virtual Topology'**, droplets can cut through inactive modules (essentially creating a temporary HRC) to reduce routing times. Hence, **Table 3-4** and **Table 3-5** show that the topology with no HRCs (**Figure 3-15(a)**) uses the least space (which can lead to greater utilization and shorter schedules) and yields the shortest routing lengths.

Schedule Length for Fixed-Size DMFB (15W×23H)								
HRC Spacing	# Mods	Schedule Length (s) for 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

Table 3-4: 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×23H DMFB.

Total Route Length for Fixed-Module-Count (8 Modules)				
Benchmark	Schedule Length (s)	Route Length (s) for HRC Spacings		
		None	Every 2 Modules	Every Modules
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)		

Table 3-5: 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 8 modules with the given topology.

3.5.5 - 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 KAMER (Keep All Maximal Empty Rectangles) placement [10][56]. The KAMER placer works by quickly computing all the *maximal empty rectangles* (*MERs*) (i.e. the empty rectangles which 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 [7].

We compare the KAMER Placer (KP) to Path Binding (PB) with the virtual topology seen in **Figure 3-15(a)** (no HRCs), both on an identical 15W×19H DMFB, such that 8 mixers could be accommodated. Both synthesis flows used list scheduling [75] and Roy's maze router [66], described in **'Section 3.4.1 - Scheduling'** and **'Section 3.4.3 - Routing'**, 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 **Figure 3-4** (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 [75]) instead of storing two/three droplets in a larger 4×2 mixing module.

Table 3-6 shows the results for 10 runs of *PCR*, *InVitro1-5* and *ProteinSplit1-6* for PB and KP for two and three storage droplets per module; PB_2 is the solution presented in this chapter. The first section shows that, in 10 runs, PB_2 has no failures until *ProteinSplit6*, 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 *ProteinSplit4-6*. The third section shows the schedule length and total assay time (which includes routing); this section shows that PB_2 and PB_3's schedules did not differ until *ProteinSplit4-6*. Thus, the scheduler did not need 3 droplets per module until *ProteinSplit4*, 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.

Path Binding (PB) with Virtual Topology VS. Kamer Placer (KP)												
Assay	# Scheduling/Placement/Routing Failures in 10 Runs				Place/Route Comp. Time (ms) (First Success)				Schedule (top) Assay Length (bottom) (First Success, in seconds)			
	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	10PF	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 -

Table 3-6: Results showing Path Binding (PB) vs. KAMER Placement (KP) with 2 and 3 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 10 runs ('-' means no failures). The second section shows the computation time of placement and routing for the first successful run ('-' means all 10 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).

The middle section of **Table 3-6** shows the synthesis times for placement and routing of the first successful run of the 10 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 3-6**, when comparing PB_2 vs. KP_2 and PB_3 vs. 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 3-6** 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 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 3-6**, often failing on placement because there was not enough room to randomly place the modules freely with the extra space.

3.6 - CONCLUSION

The online synthesis flow introduced in this chapter can run in real-time on a typical laboratory desktop system, as typified by the Intel i7^{TM} processor used in our experiments. Empirically, this work 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 re-computing 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.

CHAPTER 4 PIN-CONSTRAINED TOPOLOGY

4.1 - INTRODUCTION

As digital microfluidic biochips (DMFBs) have matured over the last decade, efforts have been made both to produce general-purpose devices (i.e., non assay-specific) and to reduce their cost. However, these two goals are typically solved as if they are mutually exclusive and there have been no DMFB solutions that maintain flexibility and programmability while reducing cost. Work done to generalize DMFBs typically depends on the flexibility of individually controlled electrodes; these devices are considered among the most costly because of the wiring complexity of independent electrodes (see **'Section 1.3.4 - Pin-Mapping'** and **'Section 1.3.5 - Wire Routing'**). In contrast, pin-constrained DMFBs are thought to reduce the wiring complexity by reducing the number of external I/O pins, but diminish the flexibility of droplet coordination, limiting their use to assay-specific chips [14][39][40][41][42][46][47][51][59][84][86][89][91][92].

CHAPTER 2 and **CHAPTER 3** both present different virtual topologies to aid in online interpretation and provide an underlying abstraction for programmability. These virtual topologies depend on independently addressable electrodes, and thus, are only compatible with more-expensive individually addressable and still-developing active-matrix DMFBs. Hence, to this point in the dissertation, we have demonstrated how virtual topologies can be leveraged toward the general-purpose goal, but not toward reducing the cost of DMFBs.

In the simplest sense and in the context of DMFBs, a topology is an arrangement and partitioning of electrodes into designated functions (e.g., droplet transport, mixing, etc.). As discussed in previous chapters, a virtual topology is an abstract arrangement of electrodes represented in the computing device's memory; that is, the physical properties of the device remain the same and each electrode can still perform each function, despite the fact that a software construct is preventing it from doing so. In contrast, a physical topology contains electrodes that are physically wired to perform certain functions. Thus, the arrangement and partition is no longer a software abstraction, but a physical reality in the DMFB.

In this chapter, we present the first physical topology specifically designed to execute generic sequences of basic microfluidic operations. Like a virtual topology, our physical topology designates unique electrodes and regions for operations and droplet transportation channels; however, the physical topology does not employ an expensive individually addressable design, but rather, an intelligently-mapped pin-constrained design which enforces a permanent, physical location for each module and transportation channel.

We introduce our physical topology design as the *field-programmable, pin-constrained digital microfluidic biochip* (*FPPC-DMFB*). This design, like the virtual topologies presented in **CHAPTER 2** and **CHAPTER 3**, is designed to execute a generic sequence of basic microfluidic operations, thus achieving the goal of remaining general-purpose. However, the pin-constrained nature of our FPPC-DMFB also dictates that it is inexpensive to fabricate. As a result, this chapter presents the first programmable, low-cost DMFB.

4.1.1 - CONTRIBUTION

The contribution of this chapter is a set of pin assignment and wire routing schemes that enable the creation of a low-cost, general-purpose pin-constrained DMFB that can perform a generic sequence of the microfluidic operations, shown in **Figure 1-3**, at pre-determined locations; this architecture is referred to as the field-programmable pin-constrained (FPPC-) DMFB, since it can be programmed after, rather than before, it is manufactured. We also introduce a high-level synthesis flow targeting the FPPC-DMFB, which establishes automatic compilation, and a detailed cost model for laying out the PCB. Although the FPPC-DMFB uses a few more pins than state-of-the-art assay-specific pin constrained designs, we show it to be competitive with the most recent general-purpose chips in terms of assay execution time. Our results also show that the PCB costs for an optimized FPPC-DMFB can be significantly cheaper than the PCB costs for a variety of application-specific pin-constrained DMFBs that have been previously reported in the literature. These results offer new insights into the relationship between pin count, PCB layer count, and cost.

4.2 - RELATED WORK

Early work on pin assignment focused exclusively on minimizing pin count, and did not consider the impact of pin reduction on wire routing. Under *array partitioning* [84], different groups of control pins are assigned to each partition; pre-synthesis partitioning reduced droplet interface significantly, while post-synthesis partitioning completely eliminated all droplet interferences. *Broadcast electrode addressing* [89] examines the electrode activation sequence produced by a synthesis tool and identifies compatible electrodes that can share a control input. Luo and Chakrabarty [51] introduced a general-purpose pin assignment scheme that provably facilitates interference-free and deadlock-free concurrent transport of up to two droplets. Quite a few papers have also been published that optimize pin assignment in conjunction with other synthesis tasks, especially routing [14][42][46][47][59][91].

Escape routing for PCBs routes known pins in a large array (e.g., a DMFB) to the array perimeter [55][85]. For pin-constrained DMFBs, the escape routing problem is extended to accommodate multi-terminal nets for control inputs that drive multiple electrodes. To date, one paper has been published that focuses explicitly on the DMFB escape routing problem [12], and one other optimizes the PCB layout for multiple DMFBs that execute the same protocol concurrently in a lock-step fashion [68]. Quite a few papers have also been published that optimize pin assignment in conjunction with wire routing [39][40][41][86], but all of these papers focus on the design and optimization of application-specific, rather than general-purpose pin-constrained chips.

The FPPC-DMFB introduced in this chapter is general-purpose, as opposed to being assay-specific. In many respects, it can be viewed as a pin-constrained implementation of a virtual topology [26][30][31]. A virtual topology segregates the surface area of a direct-addressing DMFB into modules which perform assay operations (mixing, splitting, storage, detection, etc.) and a network of streets that transport droplets between modules and I/O reservoirs. In a direct addressing context, one criticism of virtual topologies is that they limit the flexibility and reconfigurability of the device; however, prior work has shown that virtual topologies enable fast online synthesis algorithms, which can effectively respond to sensory feedback provided by the device in real-time [30][31]. Pin-constrained DMFBs also suffer from limited flexibility and reconfigurability; thus, the imposition of a virtual architecture in order to achieve general-purpose, as opposed to assay-specific, device operation is a favorable innovation [29].

Chang et al. [13] introduced a pin-constrained DMFB layout that shares many principle similarities with the FPPC-DMFB proposed here. Their device does not account for some of the finer details of module/device synchronization and I/O addressed in this chapter (e.g., the ability to independently load/unload droplets into modules). It is also unclear if the layout and wiring solution is scalable to larger devices. In contrast, this chapter presents a design variation of the FPPC-DMFB which can be routed in one PCB layer, and can scale to arbitrary numbers of operational and storage modules.

4.3 - PIN-CONSTRAINED ASSIGNMENT

The FPPC-DMFB employs a pin assignment scheme that enables all of the basic assay operations (**Figure 1-3**) to execute in a conflict-free manner. **Figure 4-1** shows a 10×16 example of an FPPC-DMFB. Similar to prior work on virtual topologies [26][30][31], the FPPC-DMFB reserves specific regions for assay operations and others for routing. The topology contains a vertical column of mixing modules on the left (blue/orange electrodes, 10-20) and a vertical column of modules on the right (orange electrodes, 31-36) that perform splitting, storage, and detection (which requires an external detector affixed above the module); we call these modules *SSD modules*.

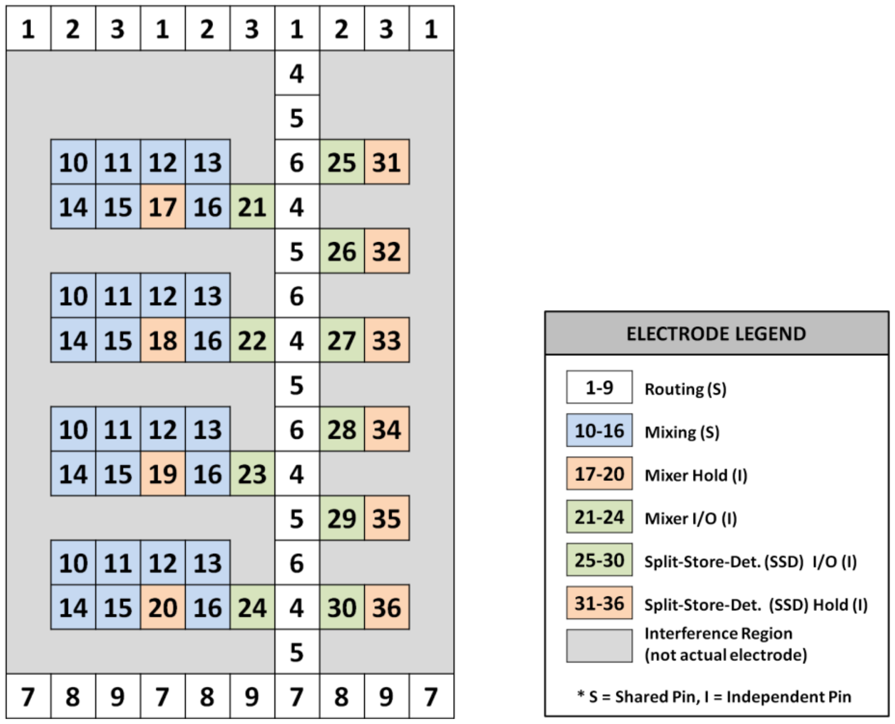

ELECTRODE LEGEND
1-9 Routing (S)
10-16 Mixing (S)
17-20 Mixer Hold (I)
21-24 Mixer I/O (I)
25-30 Split-Store-Det. (SSD) I/O (I)
31-36 Split-Store-Det. (SSD) Hold (I)
Interference Region (not actual electrode)
* S = Shared Pin, I = Independent Pin

Figure 4-1: Pin diagram for a 10×16 FPPC-DMFB which can accommodate 4 mixing modules and 6 split/store/detect (SSD) modules. Routing and mixing pins are shared; the interference region is empty space and does not contain any electrodes. Holding and I/O electrodes are independently wired to single control pins for flexibility and programmability.

White electrodes labeled 1-9 define the droplet routing regions, which ensure full connectivity between all modules. I/O reservoirs can be placed anywhere along the top or bottom of the chip. The green electrodes labeled 21-30 indicate pins that allow droplets to enter/exit each module. An *interference region* (gray) surrounds each module to isolate droplets within it from droplets in the routing region or adjacent modules. These regions are not functional and do not contain electrodes.

The layout is designed for operation concurrency and scalability. Since routing times are much shorter than operation times [75], we devote more pins to modules (as opposed to the routing region) to allow more operations to be executed simultaneously at any time-step. The architecture can also be lengthened or shortened in the vertical dimension to produce a DMFB with any desired number of modules.

4.3.1 - DMFB Operations and Synchronization

The following subsections show how the FPPC-DMFB implements the basic fluidic instruction set shown in **Figure 1-3**.

4.3.1.1 - Droplet Transport

Figure 4-2 shows that at least 3 pins are required to successfully transport a droplet along a straight path; this is called a 3-phase transport bus [72]. In **Figure 4-1**, *Pins 1-3* and *Pins 7-9* control two horizontal busses; *Pins 4-6* drive a vertical transport bus at the center of the array. The FPPC-DMFB facilitates droplet transfer between horizontal and vertical transport busses, and routable paths exist between all modules and I/O reservoirs on the chip's perimeter. Chips of arbitrary height can be instantiated without remapping transport electrodes or altering the wire-routing pattern (see **'Section 4.4.4 - Co-optimizing Pin Assignment and Wire Routing'**). The mix and SSD module hold electrodes (**Figure 4-1**, *Pins 17-20* and *Pins 31-36*, respectively) remain active during routing to ensure that droplets within the modules do not drift.

Droplets are routed one at a time because the 3-phase transport busses do not provide a sufficient number of unique pins to hold droplets in the routing area while other droplets enter/exit a module. Additional cells could be added to the bus to increase routing parallelism (e.g., **Figure 4-17**); however, given that routing times (milliseconds) are much smaller than operations (seconds), they are typically considered negligible and are often ignored [75]. With this in mind, we use a 3-phase bus because it simplifies the general purpose-nature and reduces the overall cost of the device.

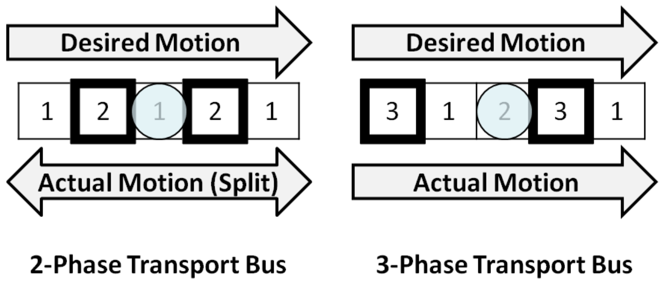


Figure 4-2: At least 3 repeatable pins are needed to move a droplet along a straight path without causing the droplet to split. Electrodes with bold borders indicate electrodes being activated next cycle.

Consider **Figure 4-3** which shows our field-programmable, pin-constrained design with two different numbers of modules and module sizes. Notice that, despite the central vertical bus ending with *Pin 5* or *Pin 6*, a clean transition can be made between buses because all of the pins adjacent to the intersection are guaranteed to be unique. Thus, the same algorithms can be used to map assays to field-programmable, pin-constrained arrays of various sizes, given that they keep the same general form. This is important because it would allow an end-user to design an assay and then go purchase the cheapest compatible pin-constrained DMFB; here,–compatibility means that there are sufficient resources available (meaning mixing and SSD modules) and that the SSD modules have appropriate detectors for the desired assay.

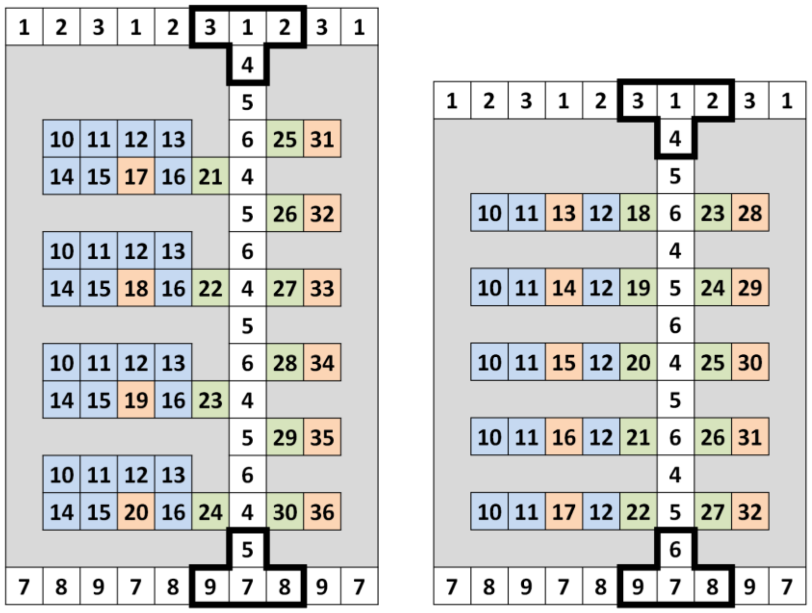
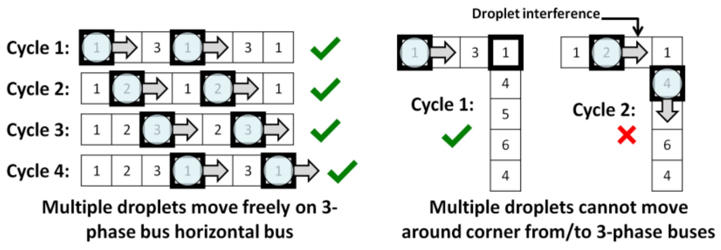


Figure 4-3: The number or size of modules can be changed and the 3-phase bus can be repeated, regardless of array size, without causing pin-conflict at the vertical-horizontal bus intersections (bold borders).



Cycle 1:
Cycle 2:
Cycle 3:
Cycle 4:
Multiple droplets move freely on 3-phase bus horizontal bus
Droplet interference
Cycle 1:
Cycle 2:
Multiple droplets cannot move around corner from/to 3-phase buses

(a) (b)

Figure 4-4: Moving two droplets concurrently is (a) feasible when moving in a straight path, but (b) not always possible when moving around a bend because droplet interference can occur.

As seen in **Figure 4-4(a)**, it is always possible to move multiple droplets along a straight path on the 3-phase bus because there is sufficient space between repeating pin numbers; however, **Figure 4-4(b)** shows that droplet interference can occur when moving around a corner. In cycle two, if the next two pins are activated (*Pin 3* and *Pin 5*), the droplets will most likely merge. It would be possible to hold *Pin 2* in cycle 3 such that the top droplet would stall and avoid the droplet interference in cycle 3; however, consider that droplets will only be making this transition when traveling to or from an I/O reservoir.

Most of the opportunities to parallelize routing occur when routing between modules. In light of this, consider **Figure 4-5**, which shows multiple droplets in the central vertical routing bus. For the lower droplet to enter the lower mixing module, the DMFB must activate *Pin 20*, while simultaneously deactivating *Pin 4*. This is possible, but notice that the top droplet requires *Pin 5* to be activated to continue downward on its path. Activating this pin will cause two adjacent electrodes to be activated near the lower droplet, which will result in a split. Moving the top droplet up, down or keeping it stationary will require *Pins 6, 5*, or *4* to be activated in cycle 2, respectively, which will each cause the bottom droplet to split. If *Pins 4-6* are not activated, then the top droplet will drift and the assay will not execute correctly.

Rather than deal with these complications, we choose to route droplets one-at-a-time instead, as the impact on total assay execution time is a small percentage of the overall time.

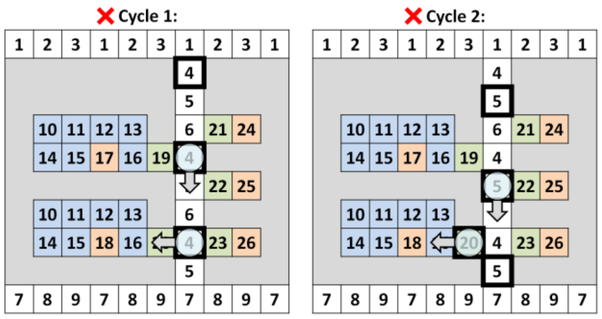


Figure 4-5: Multiple droplets moving through the vertical bus will result in an unintentional split when one tries to enter a module.

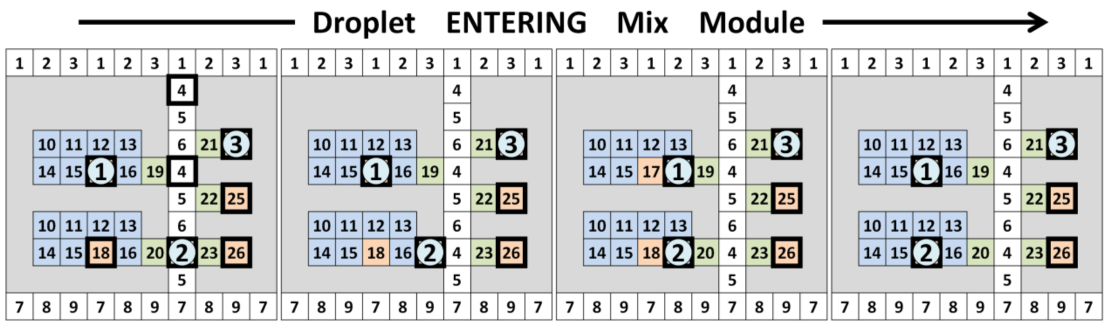
4.3.1.2 - Droplet Dispensing and Outputting

I/O reservoirs are placed above and below the top and bottom horizontal transport buses. Each I/O reservoir has an individually controlled electrode, requiring an additional control pin, which allows a droplet to enter/exit the chip via one of the horizontal transport busses. These details have been omitted from **Figure 4-1** to conserve space.

4.3.1.3 - Merging/Mixing

Figure 4-6(a) illustrates a droplet (*D2*) entering and exiting a mixing module (*M2*) without conflicting with droplets in other modules (*D1*, *D3*). At the top, *D2* has reached the routing electrode adjacent to the mixing module (*M2*) it will enter; *D1* is stored in mixing module *M1* and *D3* is stored in SSD module *SSD1*. All SSD module electrodes are activated (*Pins 24-26*) to hold all stored droplets in place during mixing module I/O. Activating *Pin 20* (*M2's* I/O cell) moves droplet *D2* to a position adjacent to *M2*. Activating *Pin 16* draws *D2* into *M2*, while transporting *D1* to an adjacent cell within *M1*. Next, all mixer hold cells (*Pins 17 and 18*) move *D1* and *D2* to identical positions within *M1* and *M2*, respectively. **Figure 4-6(a)** also shows that the electrode sequence is simply reversed to facilitate a droplet leaving a mixing module.

(a)



(b)

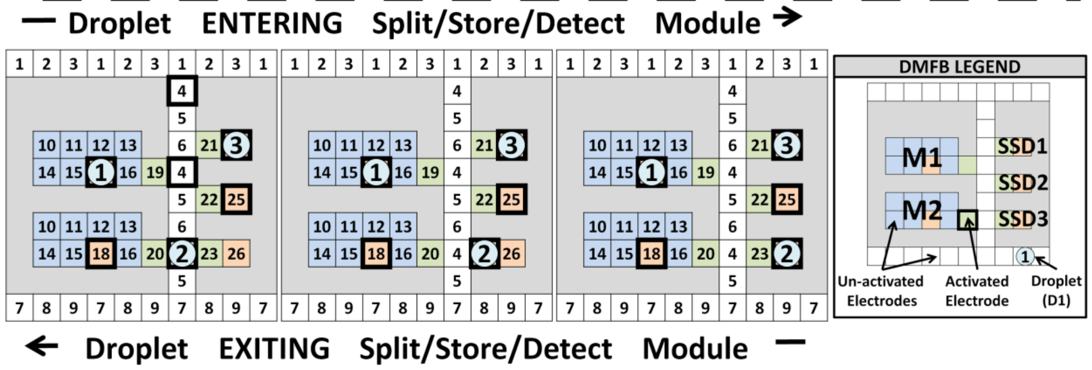

Droplet ENTERING Split/Store/Detect Module
DMFB LEGEND
M1
M2
SSD1
SSD2
SSD3
Un-activated Electrodes
Activated Electrode
Droplet (D1)
Droplet EXITING Split/Store/Detect Module

Figure 4-6: Pin-activation sequence showing how a single droplet (D2) can enter/exit (a) mix modules and (b) split/store/detect modules. Sequences are designed to allow a droplet to enter/exit any module without adversely affecting droplets (D1, D3) in other modules.

Before mixing, two droplets must first merge (i.e., collide into each other). **Figure 4-7** shows how a droplet (*D4*) merges with an existing droplet (*D2*) in *M2* to become *D5*. Once merged, the new droplet (*D5*, with twice the volume) is synced with *D1* back to the mixers' hold locations (Cycle 4, **Figure 4-7**). Mixing can then begin, presuming that *D1* is merged.

M1 and *M2* perform concurrent, synchronized mixing operations by activating *Pins 10-16*, in sequence, starting with *Pin 15* and continuing counterclockwise (i.e., *Pin 15, 14, 10, 11, 12, 13, 16*), followed by *Pin 17 and 18* together. This permits both droplets to complete one clockwise cycle in the mixing modules. Mixing can be paused whenever a droplet needs to enter/exit another mixing module.

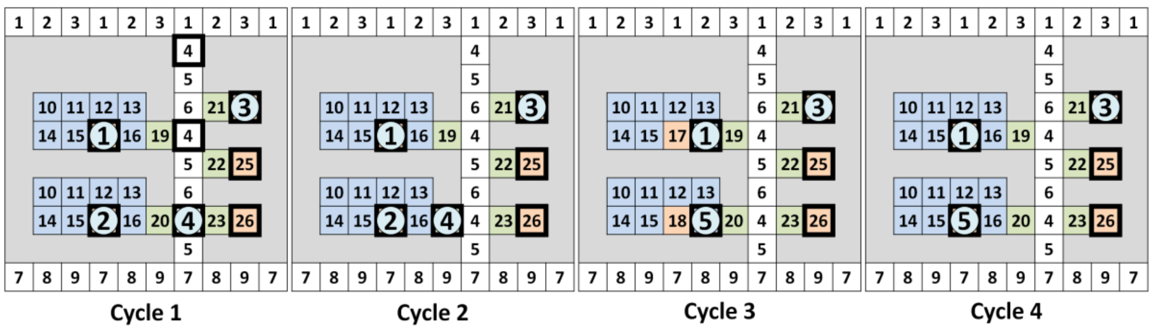


Figure 4-7: The electrode/pin activation sequence (from Cycle 1 to 4) that merges *D4* with *D2* (in *M2*) to become *D5* (twice the volume) and re-sync with any other droplets in mix modules (i.e., *D1* in *M1*).

4.3.1.4 - STORAGE, DETECTION, AND SPLITTING

SSD modules perform storage and detection (if equipped with an external detector). Both operations require a droplet to enter an SSD module and remain in place. **Figure 4-6(b)** illustrates the process by which a droplet enters/exits an SSD module (*SSD3*) without affecting droplets in other modules. All SSD hold electrodes are activated, except for *SSD3's*, which allows droplet *D2* to enter. *SSD3's* I/O electrode is then activated, followed by its hold electrode, to complete the entrance. This sequence is reversed to let a droplet exit an SSD module.

Figure 4-8(a)-(c) illustrate droplet splitting. In cycle 1, the initial position of droplet *D2*, which will be split, is on a vertical transport bus next to an SSD module's I/O cell. The cell on the transport bus is activated throughout the split. In cycle 2, the I/O cell is then activated, which stretches *D2* to cover both cells. Next, in cycle 3 the SSD module's hold cell is activated and the I/O cell is deactivated; this splits *D2* into two separate droplets: *D2*, on the hold cell, and *D4*, in the transport bus. If storage is required for *D4*, then it must be routed to an available SSD module, as shown in **Figure 4-8(d)**.

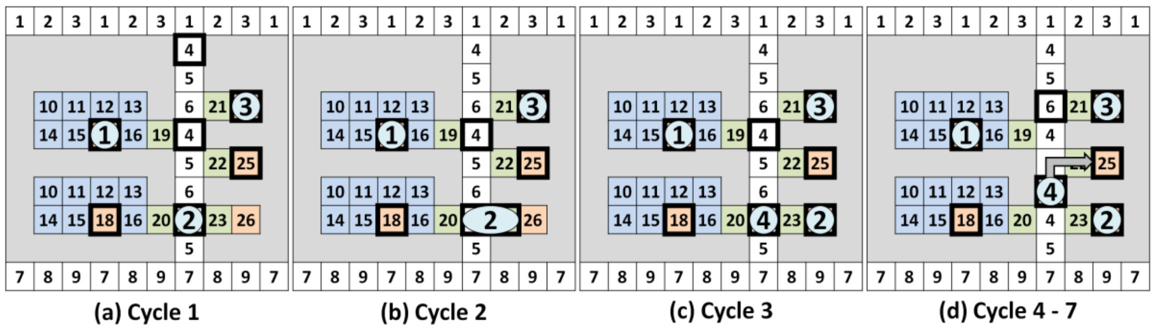


Figure 4-8: Pin-activation sequence for splitting a droplet (*D2*) and storing in split/store/detect (SSD) modules. Sequences are designed to allow a droplet to split and store without adversely affecting droplets (*D1*, *D3*) in other modules.

4.4 - FPPC-DMFB SYNTHESIS

This section describes the synthesis flow (**Figure 1-5**) that maps an assay to the field-programmable pin-constrained DMFB.

4.4.1 - SCHEDULING

We have implemented variants of list scheduling [32][75] and path scheduling [33] to target the FPPC-DMFB; in principle, other DMFB scheduling algorithms could be modified as well. The most important difference is that prior schedulers treat the DMFB as being reconfigurable, where all operations other than I/O and detection can be performed anywhere; when targeting the FPPC-DMFB, the number of mixing and SSD modules determine the resource limit.

For example, in **Figure 4-8(d)** split operations may require two SSD modules if both droplets that are produced must be stored for any period of time. In **Figure 4-9**, the split node is converted into an instantaneous split followed by two storage operations.

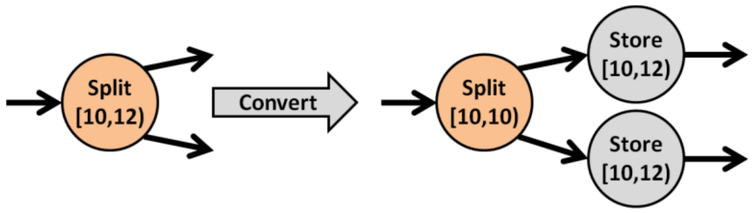


Figure 4-9: Split operations are converted to a split and two stores for synthesis.

The scheduler reserves one SSD module to address routing deadlocks, as explained later in **'Section 4.4.3 - Droplet Routing'**. Thus, in **Figure 4-1**, only 5 of the 6 SSD modules are available for storage and detection. Since only SSD modules perform storage and each module stores at most one droplet, there is no need to transport droplets between SSD modules during storage; thus, a stored droplet remains in a one SSD module for the entirety of its storage lifetime.

In contrast, modules in direct-addressing DMFBs can perform all assay operations, including storage of multiple droplets. Schedulers targeting direct-addressing DMFBs may route stored droplets from one module to another in order to free up modules to perform other operations [32][33][61]; this increases operational concurrency, yielding shorter schedules, but increases the number of droplets that need to be routed.

4.4.2 - Placement/Binding

Like Grissom and Brisk [30][32], we reduce placement to a binding problem, which is solved using the left-edge algorithm [44]. Synthesis software targeting the FPPC-DMFB does not bind a split operation to a module, as the split yields two immediate storage operations (**Figure 4-9**). Instead, the software binds the children to the SSD modules directly.

4.4.3 - Droplet Routing

4.4.3.1 - Route Computation

A routing *sub-problem* refers to the set of droplets that must be routed just before each time-step begins [78][88]. Droplets are routed one-at-a-time. Three types of routes must be computed: input reservoir to module, module to output reservoir, and module to module. To route a droplet from an input reservoir to module, the router computes a deterministic path over the horizontal and vertical busses, and applies the appropriate module input sequence (as discussed in **'Section 4.3.1 - DMFB Operations and Synchronization'**) when the droplet arrives; a similar approach is taken to route droplets from modules to output reservoirs, starting with an appropriate module output sequence. Module-to-module routing uses the vertical column in the center of the chip, applying appropriate input/output sequences at the start/end of the route.

4.4.3.2 - Droplet Dependencies and Deadlock

Routing deadlock occurs when one or more droplets wait for resources to become available that will never become free; for example, **Figure 4-10(a)** shows a cyclic dependency involving two droplets. To break the cycle, one droplet (*D3*) is routed to an empty SSD module (*SSD2*), as shown in **Figure 4-10(b).** In **Figure 4-10(c)**, the dependency is broken; however, droplet *D3* must wait for *D1* to complete its route. The scheduler always keeps one SSD module unallocated as a *routing buffer* in order to rectify any cyclic dependencies that may result from binding. The next sub-section describes the algorithm for eliminating droplet dependencies in detail.

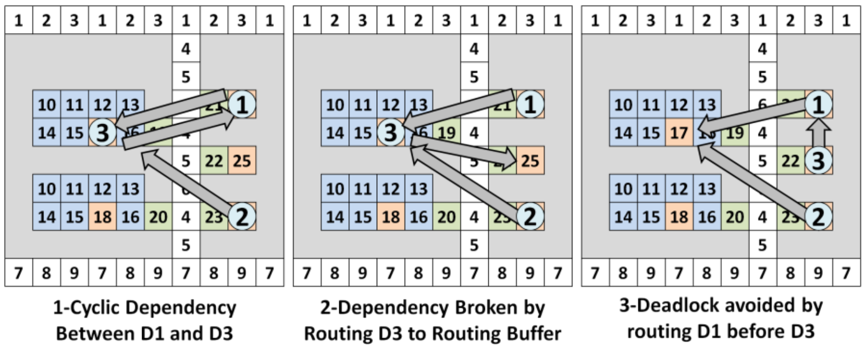

1-Cyclic Dependency Between D1 and D3
2-Dependency Broken by Routing D3 to Routing Buffer
3-Deadlock avoided by routing D1 before D3

Figure 4-10: Cyclic routing dependencies can be broken by first routing a droplet in the cycle to the routing buffer module (one of the SSD modules). Arrows indicate that the droplet at the tail end is about to travel to the module at the head end. NOTE: Legend same as Figure 4-6.

4.4.3.2.1 - Routing Algorithm

This section elaborates on the routing process discussed in **Figure 4-10** and presents pseudocode for our detailed routing algorithm in **Figure 4-11**. The router receives a scheduled and placed DAG $G = (V, E)$, where vertices represent operations and edges represent droplets that must be transferred between operations. Each vertex has a $location$, which indicates the module or I/O reservoir where the corresponding operation will take place.

Each vertex in V is scheduled to begin at a certain time-step, as computed by the scheduler. A time-step typically lasts one or two seconds and represents the time when operations are processed by their respective modules or I/O reservoirs. When a new time-step begins, then a new operation may start. This requires droplets to be routed to the module that will execute the operation. Thus, we start at time-step 0 (*Line 2*) and repeat the routing process for each time-step until the last scheduled operation begins (*Lines 3-23*); each iteration handles one routing sub-problem (time-step).

```
1   Given sequence graph $G = (V, E)$
2   int $timeStep = 0$;
3   Repeat {
4     graph $d = \emptyset$; // Dependencies
5       for ($\forall v \in V: v.startTime = timeStep$)
6         for ($\forall p \in v.parents$)
7            $d.add(p.location, v.location)$;
8       end for
9
10      list $cc = \emptyset$; // Connected Components
11      list $scc = \emptyset$; // Strongly Connected Components
12      $cc = findAllConnectedComponents(d)$;
13      $scc = findStronglyConnectedComponents(cc)$;
14      $resolveDependencies(scc)$;
15      $reverseTopologicalSort(cc)$; // $scc \subseteq cc$
16
17      for ($\forall c \in cc$)
18        for ($\forall o \in c: o.startTime = timeStep$)
19          for($\forall op \in o.parents$)
20             $routeSrcToDest(op.location, o.location)$;
21
22      $timeStep$++;
23  }  until ($time \geq \max(v.startTime, \forall v \in V)$)
```

Figure 4-11: Psuedocode for route computation

First, a graph of dependencies (d) is created based on the location of each node that is relevant to the current time-step (*Lines 4-8*). An edge *(D_x, D_y)* in the dependency graph means that droplet D_x will be routed to droplet D_y's current location, so D_y must be routed first. As seen in *Line 7*, dependencies are added to the graph based on the *location* field because droplets are being routed from the parents' location to the newly-executing node's location.

The next step is to decompose *d* into its connected components (*Line 12*), which can be computed using a simple recursive multi-directional, depth-first search [37]. Connected components are processed on-by-one. To simplify further discussion, we will assume that *d* is composed of a single connected component.

Routing is simple if *d* is acyclic. Since the algorithm routes droplets one-at-a-time, edge *(D_x, D_y)* indicates that D_y must be routed before D_x; otherwise, D_x would merge inadvertently with D_y upon completing its route. A legal routing solution for the sub-problem can be achieved by routing the droplets one-by-one in reverse topological order [43]. *Lines 10-20* in **Figure 4-11** solve the more complicated cyclic case, which is described next; in the simple acyclic case, *Lines 11*, *13*, and *14* are unnecessary.

If *d* is cyclic, routing becomes more complicated, as a cycle means that no droplet can complete its route without inadvertently merging with a droplet waiting at its destination. This problem is solved by temporarily allocating DMFB resources for storage.

The first step is to compute *strongly connected components (SCCs)* (*Line 13*) from the connected components using Gabow's path-based, depth-first search [24]. One minor modification is that we only need to identify the SCCs that contain more than one node, as single-node SCCs do not have cyclic droplet dependencies.

Once the SCCs that represent cycles are identified, the cycles must be resolved (*Line 14*). As demonstrated in **Figure 4-10**, the router randomly selects a droplet D_y from the SCC and routes it to an empty SSD module for temporary storage, which breaks the dependency cycle. The dependency graph *d* is then modified to account for the relocated droplet's new location: each edge of the form (D_x, D_y) is removed from *d* as D_x is now free to move to its destination, since D_y has moved out of the way.

The scheduler always leaves at least one SSD module free so that there is room to break one cycle in the SCC. If the SCC contains multiple intersecting cycles, then any other free SSD or mixing module could be used for temporary storage. This process repeats until *d* becomes acyclic. Once *d* becomes acyclic, a legal routing solution can be found, as previously discussed.

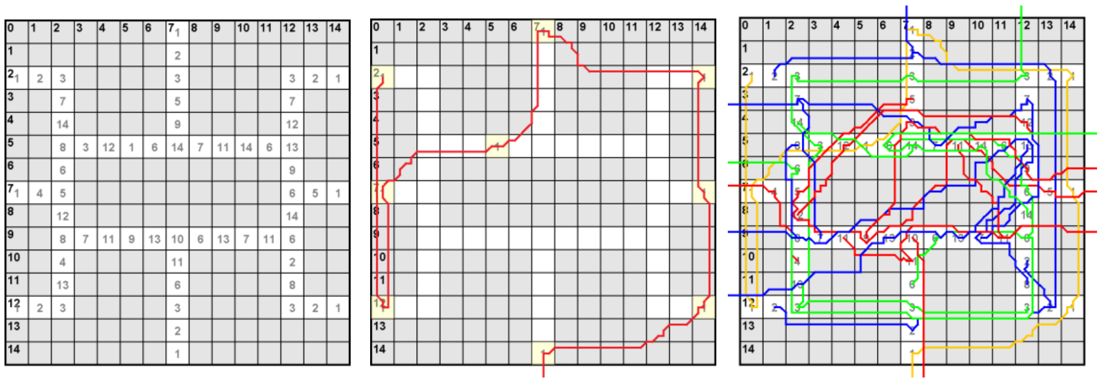
One optimization that can reduce the extra storage requirement (not shown in **Figure 4-11**) is to break SCCs one-by-one. Droplets corresponding to vertices with no predecessors in *d* are routed immediately, and the corresponding vertex is removed from *d*. Then, an SCC is chosen that satisfies the following property: for every vertex D_x belonging to the SCC and each outgoing edge (D_x, D_y), D_y also belongs to the SCC. Breaking all of the cycles in this particular SCC will ensure that at least one vertex in the updated graph *d* will have no successors.

The advantage of the second approach is that it reduces the need for temporary storage resources. As an example, suppose that *d* has two SCCs, scc_1 and scc_2, and that each requires one additional storage resource to resolve. Under the first approach, two storage resources must be allocated in order to convert *d* to an acyclic graph before the droplets can be routed. Under the second scheme, all of the droplets in scc_1 will be routed before all of the droplets in scc_2, or vice-versa. Therefore, both SCCs can use the same storage resource, so just one available module suffices. In general, if *d* contains *k* SCCs, and scc_i requires m_i storage modules, then the first scheme requires $M_1 = m_1 + m_2 + \ldots m_k$ modules for storage, whereas, the second requires $M_2 = max\{m_1, m_2, \ldots, m_k\}$ modules.

All droplet dependency cycles encountered in the benchmarks run later in **Table 4-5** were successfully resolved. The largest assay, ProteinSplit4, contains 238 nodes and represents one of the largest and most complex microfluidic benchmark assays. Thus, although a more complex droplet dependency problem could still occur in theory, it seems unnecessary, from a practical standpoint, to devote a large number of resources to resolving droplet dependency cycles.

4.4.4 - Co-optimizing Pin Assignment and Wire Routing

We introduce a wire routing scheme tailored specifically to the pin mapping scheme that defines the FPPC-DMFB. As motivation, consider a previous-published pin assignment for a pin-constrained 15x15 assay-specific DMFB designed for the PCR assay [89]. **Figure 4-12(a)** shows a 14-pin layout; **Figure 4-12(b)** highlights the wire routing solution for *Pin 1*. In **Figure 4-12(b)**, *Pin 1* drives 9 electrodes, many of which are on the perimeter of the chip. The wire routing solution for this one pin effectively blocks the ability to route additional wires into the chip on the same PCB layer. **Figure 4-12(c)** shows a complete wire routing solution for all 14 pins; a total of four PCB layers are required.



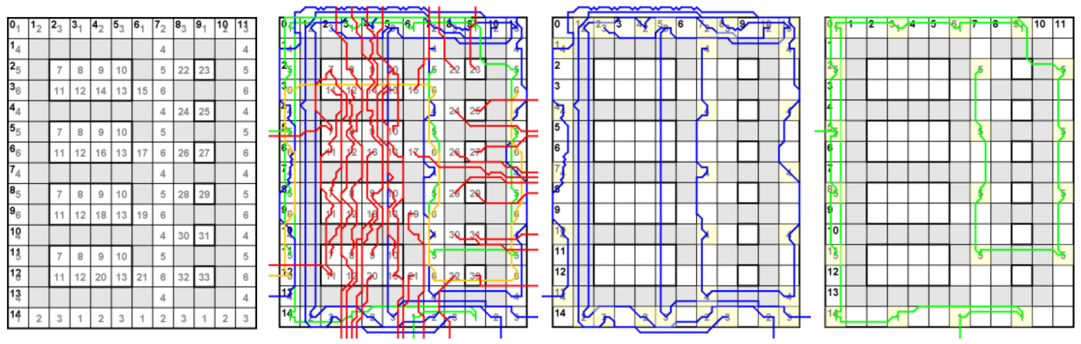
(a) (b) (c)

Figure 4-12: (a) The pin-mapping for a pin-constrained DMFB for a PCR assay [89]; (b) A wire-routing solution for *Pin 1*; (c) A complete 4-layer, wire-routing solution (each layer is represented by a different color). NOTE: Gray cells do not contain electrodes.

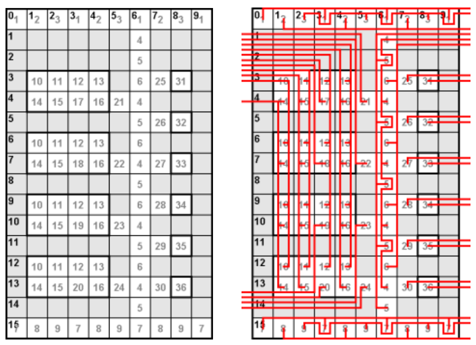
The *orthogonal capacity* of a wire routing network is the number of wires that can be routed in between the center of orthogonally adjacent electrodes. Similar to previous works, we assume an orthogonal capacity of 3 wires throughout this section [39][86]; this allows for a diagonal capacity of 6 (i.e., at most 6 wires can be routed between diagonally adjacent electrodes). For details on how to correctly model horizontal and diagonal capacities in escape routing, please refer to ref. [85]. All routing results for architectures other than the FPPC-DMFB presented in this chapter were obtained using an internally implemented multi-terminal variant of an escape routing algorithm based on negotiated congestion [55].

Figure 4-13 presents pin mapping and wire routing solutions for two FPPC-DMFB variants. **Figure 4-13(a)** presents the original pin mapping architecture [29], and **Figure 4-13(b)** shows the wire-routing solution obtained by the negotiated-congestion escape router [55]. This particular variant has three vertical busses (as opposed to the one central vertical bus shown in **Figure 4-1**. Four PCB layers are required for routing, as shown in **Figure 4-13(b)**.

Figure 4-13(c-d) depicts two of these four wire-routing layers. Wires that connect to electrodes on the 3-phase busses must span the entire array, blocking other wires from escaping to the perimeter on the same PCB layer. To eliminate this problem, we removed the two side busses and use separate groups of three control pins (*1-3*, *4-6*, *7-9*) to control the three remaining busses, as shown in **Figure 4-1** and **Figure 4-13(e)**. This yielded a single-PCB layer wire routing solution, shown in **Figure 4-13(f)**; we computed this solution manually, but have implemented an algorithm to generate our solution given different DMFB sizes as part of the automatic compilation flow.



(a) (b) Original Pin-Mapping (c) (d)



(e) Enhanced Pin-Mapping (f)

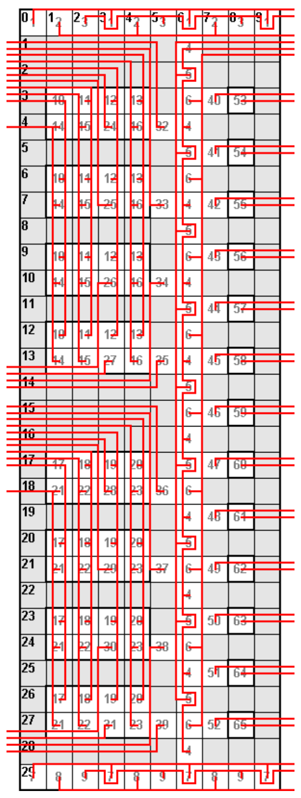
Figure 4-13: The original FPPC-DMFB [29] detailing the (a) pin-mapping and (b) 4-layer wire-routing solution; (c) layer 2 from Figure 4-13(b), showing that *Pin 2* and *Pin 3* from the horizontal busses and *Pin 4* from the vertical busses prevent other pins from escaping; (d) layer 3 from Figure 4-13(b), showing *Pin 1* from the horizontal bus and *Pin 5* from the vertical bus prevent other pins from escaping. (e-f) The pin-mapping (same as Figure 4-1) and wire-routing solution for the enhanced FPPC-DMFB introduced in this chapter.

Removing the left and right vertical busses may reduce the number of potential I/O locations; if extra I/O is required, the horizontal busses at the top and/or bottom of the chip can be extended; alternatively, mixing or SSD modules in the center of the chip could be replaced with an I/O reservoir that is attached to the central vertical bus. Providing independent control of the two horizontal 3-phase busses requires three extra control pins, but reduces the PCB layers from 2 to 1.

Another subtle detail is that an extra horizontal row is added between the top vertical bus and the topmost mixing and SSD modules; this extra space is needed to provide access for control wires that drive electrodes in the center of the chip to escape, as shown in **Figure 4-13(f)**.

The original design assumed that *Pins 7-13* (see **Figure 4-13(a)**) could be shared by an arbitrary number of mixing modules, regardless of the height of the chip; however, because of the independently controlled module hold and I/O pins (*Pins 14-21* in **Figure 4-13(a)**), there is not enough room to extend the shared pins indefinitely without introducing additional PCB layers to facilitate wire routing to these shared electrodes.

The solution is to limit the number of shared electrodes to groups of four continuous mixing modules, which happens to be the number shown in **Figure 4-13**. For chips with more than four mixing modules, as shown in **Figure 4-14**, the same basic layout and wiring pattern shown in **Figure 4-13(e)** and **Figure 4-13(f)** must be repeated. **Figure 4-14(a)** shows two groups of four mixing modules, while **Figure 4-14(b)** shows how the scheme can generalize to an arbitrary number of mixing modules that is not just an integer multiple of four.


(a)

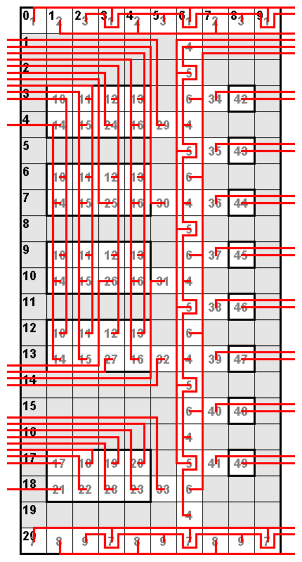

(b)

Figure 4-14: The wire-routing model for the FPPC-DMFB generalizes to an unlimited number of modules; each group of up to four mixing modules shares seven common pins as seen in FPPC-DMFBs with (a) eight mixing modules and (b) 5 mixing modules.

4.5 - EXPERIMENTAL METHODOLOGY

4.5.1 - WIRE ROUTING COST ANALYSIS

Our experimental methodology and results focus on obtaining a precise cost (in terms of US dollars) of the PCB for different DMFB architectures, including direct-addressing, a variety of assay-specific pin-constrained architectures, and several variants of the FPPC-DMFB presented in this chapter. Prior work has reported pin-count and the number of PCB layers as a rough proxy for cost, but have not reported the actual cost (price) of the PCB itself. For example, this makes it difficult to determine whether or not it is profitable to increase the pin count if doing so reduces the number of PCB layers.

With real-time cost estimates, we can present a more accurate picture of these tradeoffs, using the experimental methodology outlined here.

4.5.1.1 - COST COMPUTATION

We use Advanced Circuits' online instant quote feature to estimate the cost of each PCB [1]. The primary metrics that need to be set by Advanced Circuits' cost calculator are the PCB length and width and the number of wire-routing layers. Vias are used to connect multiple layers, and thus it is necessary to specify the via size, along with the wire trace spacing and size, which dictates the thickness of wire traces and the minimum spacing between tracing. All other metrics are left at their default values. It is important to note that wire-length does not directly affect the cost, as long as the PCB is routable without increasing its area by adding extra space.

We assume that all DMFBs are driven by a low-cost Atmega 1284 microcontroller with 32 general purpose I/Os (GPIOs) that can be used to address the DMFB array [2]. If a DMFB has 32 or less pins that need to be driven, then the microcontroller can be used without any further circuitry. However, if a DMFB has more than 32 pins, shift registers must be used to drive the additional pins. Shift registers can be daisy chained to feed an arbitrary number of additional inputs such that only 4 microcontroller signals can control the shift register chain: the serial data input (SER), shift register clock input (SCK), storage register clock input (RCK) and the reset input (SCLR). Thus, if there are more than 32 pins, all but 28 will need to be shifted in through the 8-bit shift registers. We assume the Fairchild 74VHC595MTC 8-bit shift register [3], which can be purchased from Mouser for roughly \$0.14 per unit in quantities of 2,500 [5]. **Equation 4.1** shows the number of shift registers necessary to properly drive the DMFB.

$$numShiftRegs = \begin{cases} \left\lceil \frac{numPins-28}{8} \right\rceil, numPins > 32 \\ 0, otherwise \end{cases} \tag{4.1}$$

The Atmega 1284 microcontroller operates at 20MHz [2], while the *droplet actuation frequency* (i.e., the time it takes to transport a droplet between two adjacent electrodes) of a typical DMFB is only 100Hz [88]. Thus, with a conservative estimate of 5 cycles per shift operation, the Atmega 1284 could load 400 pin values into the shift registers in just 1% of the droplet actuation cycle, maintaining the integrity of the signal needed for proper droplet transportation.

As shown in **Equation 4.2**, we compute the wire routing cost to be the sum of the price of the PCB estimate from Advanced Circuits and the shift registers necessary to connect all pins. We do not include the price of the microcontroller since this cost is essentially a constant, irrespective of the choice of DMFB. We also do not consider the cost of circuitry to amplify the voltage produced by the microcontroller to levels appropriate to drive the DMFB; the voltages required vary based on underlying technology parameters. Typical actuation voltages are in the 50-70V range [60][63]; low voltage devices that operate at ~15V have also been reported [19][58].

$$Cost_{WR} = Cost_{PCB} + Cost_{SR} \tag{4.2}$$

The wire-routing cost is primarily a function of the number of PCB layers, its area, and wire trace width, as shown in **Equation 4.3**. Assuming an orthogonal capacity of 3, the FPPC-DMFB designs described in this work can always generate a solution in a single layer, as shown in **Figure 4-13(f)** and **Figure 4-14.** In all other cases, we use the negotiated congestion escape router [55] to determine the number of layers required to achieve a legal route. In general, using larger feature sizes (e.g., wire trace size, via size) reduce the PCB cost estimation [1].

$$Cost_{PCB} = f(numLayers, width_{PCB}, height_{PCB,} width_T) \tag{4.3}$$

Equation 4.4 computes the orthogonal capacity according to the metrics in **Table 4-1** and diagram in **Figure 4-15**. For example, as seen later in **Table 4-3**, with an electrode pitch of 2mm (i.e., 2mm between the center of orthogonally adjacent electrodes), a wire trace width and spacing of 0.007in, via width of 0.014in and via contact width of 0.024in, we compute the orthogonal capacity to be $o_{cap} = \lfloor 3.12 \rfloor = 3$.

$$o_{cap} = \left\lfloor \frac{width_{ELEC} - width_{VC} - width_{TS}}{width_T + width_{TS}} \right\rfloor \tag{4.4}$$

Once the number of layers is computed, we next compute the width and height of the PCB. **Figure 4-16** shows the basic layout for estimating the dimensions of the PCB ($width_{PCB} \times\ height_{PCB}$). As shown in **Equation 4.5** and **Figure 4-16**, the PCB height is simply the array's height plus one inch; **Equation 4.6** shows that, in addition to the array's width and a 1 inch buffer, the PCB's width accounts for the extra space needed for shift registers. The amount of extra space added to the PCB width depends on the number of shift registers needed by the DMFB and is computed by **Equation 4.7**. In short, shift registers are stacked vertically until there is no more room, at which point additional columns of shift registers are added.

$$height_{PCB} = height_A + 1 \tag{4.5}$$

$$width_{PCB} = width_A + width_{PCB\_SR} + 1 \tag{4.6}$$

$$width_{PCB_{SR}} = \left\lceil \frac{numShiftRegs}{\lfloor height_{PCB}/(height_{SR} + width_{SRS}) \rfloor} \right\rceil \times (width_{SR} + width_{SRS}) \tag{4.7}$$

PCB Fabrication Parameters	
Feature	**Symbol**
Electrode Pitch	$\boldsymbol{width_{ELEC}}$
Via (Hole) Width	$\boldsymbol{width_{V}}$
Via (Hole) Contact Width	$\boldsymbol{width_{VC}}$
Wire Trace Width	$\boldsymbol{width_{T}}$
Min. Space Between Wire Trace	$\boldsymbol{width_{TS}}$
Shift Register Width [3]	$\boldsymbol{width_{SR}}$
Shift Register Height [3]	$\boldsymbol{height_{SR}}$
Spacing Between Shift Registers	$\boldsymbol{width_{SRS}}$

Table 4-1: PCB Fabrication Parameters.

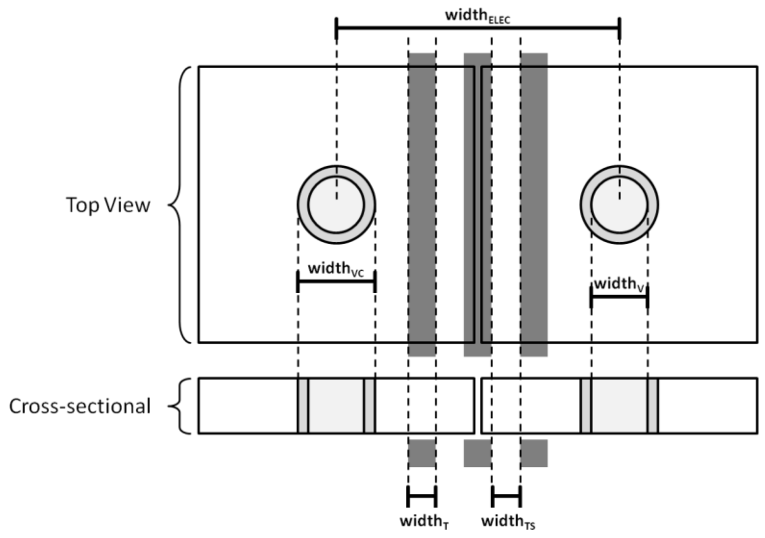


Figure 4-15: A top-down and cross-sectional view of a PCB showing dimensions for the electrode pitch (ELEC), via hole (V), via contact (VC), wire trace (T) and minimum wire trace spacing (VS) in a DMFB.

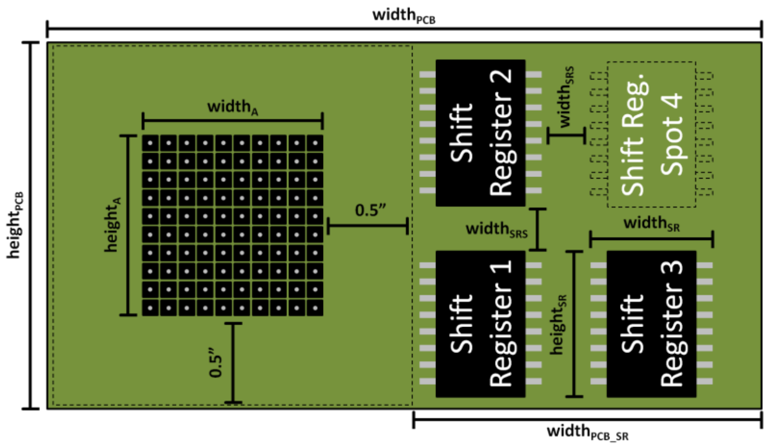


Figure 4-16: The component layout for PCB size estimation. The electrode array is surrounded by a 0.5 inch perimeter of empty space. The PCB width is extended to add as many shift registers as necessary.

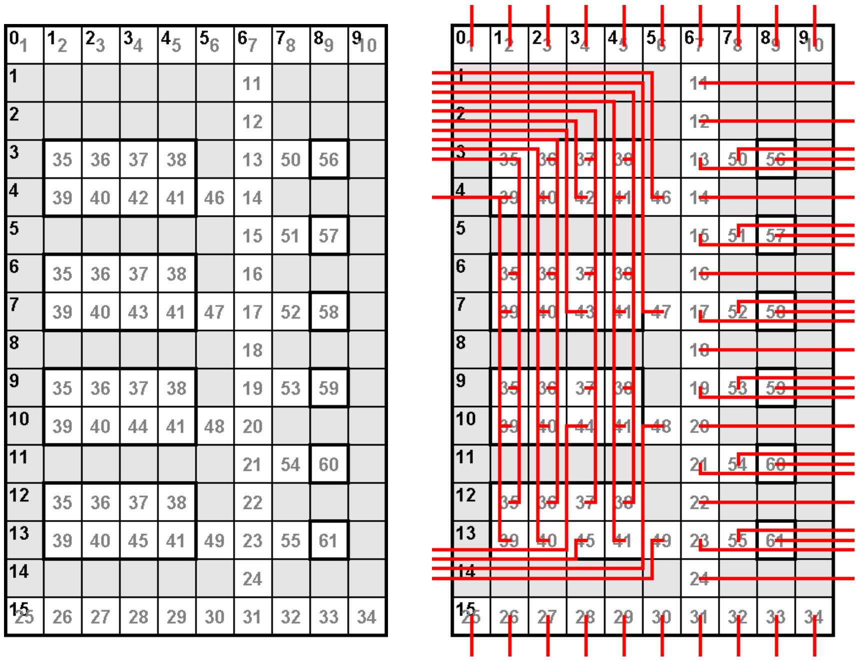
4.6 - EXPERIMENTAL RESULTS

We implemented the FPPC-DMFB and associated synthesis algorithms in a publicly available open-source software framework, written in C++ [27]. All experiments were performed using a 2.8GHz Intel Core i7 CPU and 4GB RAM running a 64-bit version of Windows 7.

4.6.1 - BENCHMARKS

We extracted pin-constrained designs for PCR, in-vitro diagnostic, protein synthesis and multi-functional DMFBs (two versions of each, eight total) from previous DAC papers [51][89]; these benchmarks are labeled *ZHAO_XXX* [89] and *LUO_XXX* [51], where "*XXX*" is the name of one of the three assays used in their experiments (*PCR*, *INVITRO*, *PROTEIN*) or a multi-functional chip that is co-designed to perform all three of those assays (*MULTI*). Since the electrode layouts are identical for both of these works (only the pin-assignment is different), we created directly addressable versions of the four different assays, entitled *XXX_DA*.

The FPPC-DMFB design presented in this chapter attempts to reduce the number of control pins required to achieve general-purpose, rather than assay-specific, operation. Thus, we denote this benchmark as *FPPC_4_MODULE* for the 4-mixer version seen in **Figure 4-13(e)** and *FPPC_8_MODULE* for the 8-mixer version seen in **Figure 4-14(a)**. We also introduce a routing-optimized FPPC-DMFB which replaces the 3-phase horizontal and vertical routing busses with independently-addressable busses, as shown in **Figure 4-17(a)**. This architecture increases the number of control pins, but facilitates concurrent droplet routing, which can improve performance.



(a) (b)

Figure 4-17: An FPPC-DMFB variant that replaces 3-phase busses with direct addressing busses, allowing for concurrent droplet transport: (a) pin assignment and (b) wire routing solutions.

In terms of routability, this design is scalable to an arbitrary number of mixing modules, as shown in **Figure 4-17(b)**, using one PCB layer. This pin-assignment is named *FPPC_4_DA_BUS* and *FPPC_8_DA_BUS* for the 4- and 8-mixer versions, respectively.

We also consider several direct-addressing DMFBs, including two versions of the FPPC with 4 and 8 mixing modules (*FPPC_4_DA* and *FPPC_8_DA*), which use the layouts in **Figure 4-13(e)** and **Figure 4-14(a)**, respectively, but allocate a unique control input to drive each electrode. Lastly, we include results for three direct-addressing DMFBs having dimensions: of 15x15 (*15x15_DA*), 10x10 (*10x16_DA*), and 10x30 (*10x30_DA*).

Wire routing solutions for all FPPC-DMFBs, except for the directly addressable ones, were computed as described in **'Section 4.4.4 - Co-optimizing Pin Assignment and Wire Routing'** (*FPPC*); all other DMFB wire routes were computed using a multi-terminal implementation of an escape router based on negotiated congestion [55]. If a pin's wire net cannot be routed on a top-level layer, it is routed on a lower layer in its entirety such that vias are used to connect the wire net to its corresponding electrodes and to the external driving pin.

4.6.2 - PCB LAYERS & ORTHOGONAL CAPACITY

Next, we examine the relationship between orthogonal capacity, the number of control pins and the number of PCB layers. The right side of **Table 4-2** shows the resultant number of PCB layers it takes to route each DMFB as the orthogonal capacity varies from 2 to 10. The dark squares highlight the lowest orthogonal capacity that achieves the smallest number of PCB layers for each benchmark. These results show that 16/21 ≈ 76% and 19/21 ≈ 90% of the benchmarks reach their smallest number of layers when the orthogonal capacity is 3 or 4. For several of the assay-specific pin-constrained designs (*ZHAO_XXX* and *LUO_XXX*), the minimum number of PCB layers ranges from 3 to 6.

DMFB Benchmark Description and Number of Layers Per Orthogonal Capacity

DMFB Characteristics						Number of Layers Per Orthogonal Capacity								
Name	Wire Routing Algorithm	Array Dimensions		# Elecs.	# Pins	2	3	4	5	6	7	8	9	10
		X	Y											
ZHAO_PCR	[55]	15	15	62	14	**4**	4	4	4	4	5	4	4	4
ZHAO_INVITRO	[55]	15	15	59	25	4	4	**3**	4	4	4	4	4	4
ZHAO_PROTEIN	[55]	15	15	54	27	4	**3**	4	4	4	4	4	4	4
ZHAO_MULTI	[55]	15	15	81	32	5	5	6	6	6	5	**4**	5	5
LUO_PCR	[55]	15	15	62	22	**5**	5	5	5	5	5	5	5	5
LUO_INVITRO	[55]	15	15	59	21	**5**	5	5	5	5	5	5	5	5
LUO_PROTEIN	[55]	15	15	54	21	**4**	4	4	4	4	5	4	4	4
LUO_MULTI	[55]	15	15	81	27	**6**	6	6	6	6	6	6	6	7
PCR_DA	[55]	15	15	62	62	**1**	1	1	1	1	1	1	1	1
INVITRO_DA	[55]	15	15	59	59	**1**	1	1	1	1	1	1	1	1
PROTEIN_DA	[55]	15	15	54	54	**1**	1	1	1	1	1	1	1	1
MULTI_DA	[55]	15	15	81	81	**1**	1	1	1	1	1	1	1	1
FPPC_4_MODULE	FPPC	10	16	82	36	2	**1**	1	1	1	1	1	1	1
FPPC_8_MODULE	FPPC	10	30	146	65	2	**1**	1	1	1	1	1	1	1
FPPC_4_DA_BUS	FPPC	10	16	82	61	2	**1**	1	1	1	1	1	1	1
FPPC_8_DA_BUS	FPPC	10	30	146	104	2	**1**	1	1	1	1	1	1	1
FPPC_4_DA	[55]	10	16	82	82	**1**	1	1	1	1	1	1	1	1
FPPC_8_DA	[55]	10	30	146	146	**1**	1	1	1	1	1	1	1	1
15x15_DA	[55]	15	15	225	225	3	2	2	2	**1**	1	1	1	1
10x16_DA	[55]	10	16	160	160	2	2	**1**	1	1	1	1	1	1
10x30_DA	[55]	10	30	300	300	2	2	**1**	1	1	1	1	1	1

Table 4-2: The left side gives a description of the 21 DMFB wire-routing benchmarks showing the wire-routing algorithm used (WR Alg.), array dimensions (Array Dims.), number of electrodes (# Elecs.) and number of control pins (# Pins). The right side shows the number of PCB layers yielded for each DMFB as a function of orthogonal capacity, which varies from 2 to 10; dark squares highlight the lowest orthogonal capacity which achieves the minimum number of PCB layers for each chip.

As an example, consider *ZHAO_PCR*, which is shown in **Figure 4-12**. This particular pin mapping requires long wires along the perimeter of the chip, which prevent other wires from escaping on the same PCB layer; thus, additional layers are necessary. In this case, reducing the wire size to increase more internal bandwidth (orthogonal capacity) does not have a significant effect on the number of PCB layers. On the other hand, converting these pin-constrained designs to direct-addressing DMFBs (*XXX_DA*) yields single-layer routing solutions for all orthogonal capacities, although the number of additional control pins increases from 2x to 4.4x.

The FPPC-DMFBs shown in **Table 4-2** can be routed in 1 or 2 PCB layers, even with the smallest orthogonal capacities; furthermore, *FPPC_4_MODULE* uses a competitive number of pins, compared to *ZHAO_XXX* and *LUO_XXX*, indicating that the FPPC-DMFB offers the simultaneous advantages of being general-purpose and low cost when compared to these previously-published assay-specific designs. Further details are discussed in **'Section 4.6.4 - PCB Cost Results'**.

4.6.3 - Wire Routing Cost Analysis

This section analyzes the impact of different PCB design parameters on the overall design. We vary the electrode pitch (1 mm, 2 mm, 2.54 mm), and determine an appropriate set of parameters for each electrode size in order to minimize cost. The following subsection translates these results into cost estimates for PCBs designed for the DMFBs in **Table 4-2**.

4.6.3.1 - Metric Selection

As mentioned in **'Section 4.5.1.1 - Cost Computation'**, the wire-trace width/spacing ($\boldsymbol{width_T/width_{TS}}$), smallest via size ($\boldsymbol{width_V}$), PCB dimensions ($\boldsymbol{width_{PCB}}$, $\boldsymbol{height_{PCB}}$) and number of layers are the predominant factors that impact PCB costs [1]. **Table 4-2** shows that the number of layers depends on the orthogonal capacity, which in turn is a function of the aforementioned trace and via metrics, as per **Equation 4.4**.

Table 4-3 presents various parameter combinations that yield a range of orthogonal capacities for 1 mm, 2 mm and 2.54 mm electrodes. Advanced Circuits' PCB cost estimator provides the following wire trace sizes (all in inches): 0.0025, 0.003, 0.004, 0.005, 0.006, 0.007, 0.008, 0.010 and 0.012. To achieve each orthogonal capacity, we chose the lowest trace size and highest minimum via size and via contact size such that $\boldsymbol{width_V \geq 2 \times width_T}$ and $\boldsymbol{width_{VC} \geq width_V + 0.01}$; these requirements were relaxed for the smaller 1 mm electrodes (such that $width_{VC} \geq \boldsymbol{width_V + 0.003}$) because there was not enough space to utilize such conservative size estimates.

For each electrode pitch, we select a particular set of parameters and report the resultant orthogonal capacity. For 1 mm, estimations were not available from the estimation calculator for orthogonal capacities of 4 and 5; it was not possible to select parameters small enough to achieve an orthogonal capacity of 6. For 1 mm electrodes, we conservatively selected an orthogonal capacity of 2: even as the number of layers increases to 4, the cost is still less than a single-layer PCB with orthogonal capacity of 3. Although our main goal is to achieve a single-layer wire-routing solution, which requires an orthogonal capacity of 3, **Figure 4-18** shows a generic two-layer solution for *FPPC_4_MODULE* with an orthogonal capacity of 2 (e.g., every third wire needs to be moved to a second PCB layer); as explained in the following paragraphs, single and dual layer solutions are equally priced.

Price Estimations for Varying Numbers of Layers and Parameters of a 2”×2” PCB

Electrode Pitch		Advanced Circuit Metrics			oCap	2" × 2" Price (@ 2,500 QTY) with Varying Number of Layers				
mm	in	Trace Size/ Space	Via Size	Via Contact Size		1	2	3	4	5
1	0.0394	**0.005**	**0.010**	**0.013**	**2**	**$1.20**	**$1.20**	**$1.73**	**$1.81**	**$2.09**
		0.004	0.008	0.011	3	$1.88	$1.88	$2.43	$2.52	$2.81
		0.003	0.009	0.012	4	N/A	N/A	N/A	N/A	N/A
		0.002	0.008	0.011	5	N/A	N/A	N/A	N/A	N/A
2	0.0787	0.008	0.028	0.038	2	$0.99	$0.99	$1.41	$1.46	$1.79
		0.007	0.014	0.028	3	$0.99	$0.99	$1.41	$1.47	$1.79
		0.006	**0.012**	**0.024**	**4**	**$0.99**	**$0.99**	**$1.46**	**$1.53**	**$1.80**
		0.005	0.013	0.023	5	$1.10	$1.10	$1.63	$1.71	$1.99
		0.004	0.008	0.026	6	$1.88	$1.88	$2.43	$2.52	$2.81
2.54	0.1000	0.012	0.030	0.040	2	$0.99	$0.99	$1.41	$1.41	$1.79
		0.010	0.020	0.030	3	$0.99	$0.99	$1.41	$1.41	$1.79
		0.008	0.018	0.028	4	$0.99	$0.99	$1.41	$1.46	$1.79
		0.006	0.024	0.034	5	$0.99	$0.99	$1.46	$1.53	$1.80
		0.006	**0.012**	**0.022**	**6**	**$0.99**	**$0.99**	**$1.46**	**$1.53**	**$1.80**
		0.005	0.010	0.020	7	$1.20	$1.20	$1.73	$1.81	$2.09
		0.004	0.022	0.032	8	$1.38	$1.38	$1.93	$2.02	$2.31

Table 4-3: The left side shows various metrics used for the Advanced Circuit PCB cost estimator [1] and resultant orthogonal capacity (oCap). The right side provides corresponding price estimates from the Advanced Circuit PCB cost estimator for a 2”×2” PCB with varying numbers of layers; dark rows represent the selected metrics for each electrode pitch.

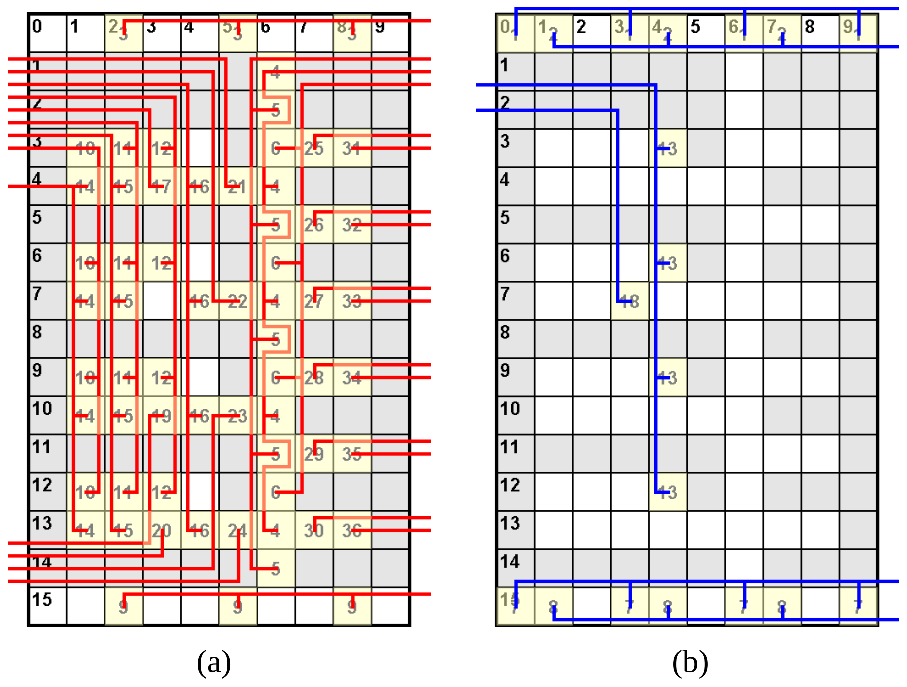


Figure 4-18: Two-layer wire-routing solution for *FPPC_4_MODULE* with an orthogonal capacity of 2; (a) Layer 1; (b) Layer 2.

For 2 mm and 2.54 mm electrode pitches, we select the metric set corresponding to orthogonal capacities of 4 and 6, respectively. These are the highest capacities before the price increases significantly for each electrode size and reductions in layer count do not typically occur at higher orthogonal capacities, as shown in **Table 4-2**.

Table 4-3 reveals that, in general, as the feature sizes decrease, particularly the wire trace size, the fabrication costs increase, and that 1- and 2-layer solutions are identical for all cases; this is because PCBs can be printed on two sides. **Table 4-3** reflects this observation, as prices tend to jump more significantly as each odd numbered layer is added, meaning that a new, physical dual-sided PCB layer must be added.

4.6.4 - PCB COST RESULTS

Table 4-4 presents detailed cost results for each DMFB architecture listed in **Table 4-2** assuming 1 mm electrode sizes and an orthogonal capacity of 2, as discussed in the preceding subsection. **Table 4-4** reports the number of pins and the subsequent number of required shift registers (SR), along with the PCB dimensions, which include extra space allocated for shift registers. The PCB dimensions are fed directly into the online cost estimator. The reported cost of the PCB is estimated under the assumption of a 4-week deliver time (slowest) at a quantity of 2500 [1]; the shift registers cost \$0.139 apiece at a quantity of 2500 [5]. As described in **Equation 4.2**, the total cost is that of the PCB plus shift registers.

Cost Estimates for 21 DMFBs with 1 mm Electrode Pitch and Orthogonal Capacity of 2								
DMFB Details						Cost ($)		
DMFB Name	# Pins	# SR	Adjusted PCB Dim		# Layers	Board	SR	Total
			X (in)	Y (in)				
FPPC_4_MODULE	36	1	1.7638	1.6299	2	$1.01	$0.14	$1.15
ZHAO_PCR	14	0	1.5906	1.5906	4	$1.33	$0.00	$1.33
ZHAO_INVITRO	25	0	1.5906	1.5906	4	$1.33	$0.00	$1.33
ZHAO_PROTEIN	27	0	1.5906	1.5906	4	$1.33	$0.00	$1.33
LUO_PROTEIN	21	0	1.5906	1.5906	4	$1.33	$0.00	$1.33
ZHAO_MULTI	32	0	1.5906	1.5906	5	$1.55	$0.00	$1.55
LUO_PCR	22	0	1.5906	1.5906	5	$1.55	$0.00	$1.55
LUO_INVITRO	21	0	1.5906	1.5906	5	$1.55	$0.00	$1.55
LUO_MULTI	27	0	1.5906	1.5906	6	$1.58	$0.00	$1.58
PROTEIN_DA	54	4	1.9606	1.5906	1	$1.08	$0.56	$1.64
INVITRO_DA	59	4	1.9606	1.5906	1	$1.08	$0.56	$1.64
FPPC_4_DA_BUS	61	5	1.7638	1.6299	2	$1.01	$0.70	$1.71
PCR_DA	62	5	1.9606	1.5906	1	$1.08	$0.70	$1.78
FPPC_8_MODULE	65	5	1.7638	2.1811	2	$1.16	$0.70	$1.86
MULTI_DA	81	7	2.3307	1.5906	1	$1.12	$0.97	$2.09
FPPC_4_DA	82	7	2.1339	1.6299	1	$1.19	$0.97	$2.16
FPPC_8_DA_BUS	104	10	2.1339	2.1811	2	$1.37	$1.39	$2.76
FPPC_8_DA	146	15	2.5039	2.1811	1	$1.57	$2.09	$3.66
10x16_DA	160	17	2.8740	1.6299	2	$1.37	$2.36	$3.73
15x15_DA	225	25	3.4409	1.5906	3	$2.52	$3.48	$6.00
10x30_DA	300	34	3.6142	2.1811	2	$2.94	$4.73	$7.67

Table 4-4: Wire routing costs for the 21 benchmarks, sorted in order of increasing total cost, showing the adjusted PCB dimensions (after adding room for shift registers (SR)), number of layers and resultant breakdown of costs (PCB and shift registers).

In **Table 4-4** the DMFB chips are sorted in increasing order of cost, and the results show that *FPPC_4_MODULE* is the cheapest design by at least $0.18 per board, compared to assay-specific chips *ZHAO_XXX* and *LUO_XXX*. The *FPPC_4_DA_BUS* design, which can transport multiple droplets concurrently because the 3-phase busses are replaced by direct-addressing busses, is $0.56 more expensive than its pin-optimized counterpart. In the following section (**'Section 4.6.5 - Performance'**), we show that our FPPC-DMFB design with serial routing is competitive with recent pin-constrained designs, and in many cases, outperforms state-of-the-art direct addressing; thus, given that droplet transport times are orders of magnitudes faster than operation times [75][88] and that the FPPC-DMFB design with serial routing is competitive (or better) to prior designs in terms of performance, we believe that *FPPC_4_MODULE* is the best overall solution when considering price and performance.

Table 4-4 also shows that the PCB cost for many directly addressable DMFBs are less expensive than pin-constrained counterparts. For example, even though *PCR_DA* is 0.4 in longer than *LUO_PCR*, the PCB is $0.47 cheaper because it requires fewer layers; however, the directly addressable version requires 5 shift registers, increasing the cost of the design to be $0.23 more expensive than *LUO_PCR*.

This overall trend of shift-register costs is shown in the bottom half of **Table 4-4**; starting with *PROTEIN_DA* and looking downward, the number of pins, shift registers and total cost are non-decreasing. ***Thus, it is important to minimize the pin-count; however, it must be balanced with a reduction in the number of PCB layers, as seen with our FPPC-DMFB, to achieve a comparably low-cost wire-routing solution.***

Figure 4-19 presents the final cost estimates for each of the 21 DMFBs with 1 mm, 2 mm and 2.54 mm electrodes, using the parameters reported in **Table 4-3**, and the shift register costs described previously. All estimates assumed for a 4-week delivery time and shipment quantity of 1000 because price estimates of 2500 could not be obtained online for all benchmarks. The benchmarks are sorted, from left to right, in order of increasing average cost. Although there is some variation when comparing the cost between different electrode sizes (e.g., from *MULTI_DA* to *LUO_INVITRO*, the 1 mm and 2 mm costs decrease, while the 2.54 mm cost increases), the overall trend of increasing price follows the average for each electrode pitch. Other than the obvious fact that bigger electrodes take up more space, and thus require a larger and more expensive PCB, this indicates that electrode pitch is not a major factor for wire-routing cost since PCB manufacturing parameters can be adjusted to compensate for orthogonal capacity to keep the number of layers as low as possible.

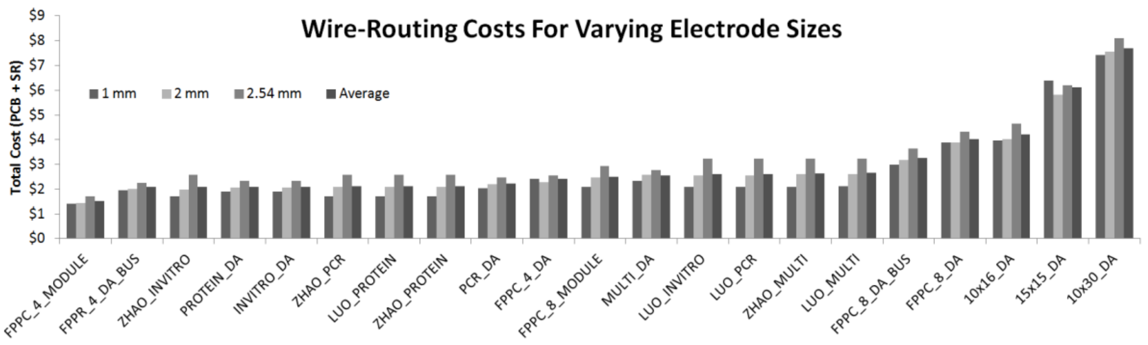


Figure 4-19: Total wire-routing fabrication costs per DMFB board, including PCB manufacturing and shift register costs, for 21 DMFB designs utilizing 1 mm, 2 mm and 2.54 mm electrodes. The benchmarks are sorted, from left to right, in order of increasing average cost.

For several of the benchmarks, the 1 mm boards are more expensive than their 2 mm counterparts. Recall that shift registers are stacked vertically and that additional width is added in the X-dimension to accommodate new columns, as necessary. Since the 2 mm instances provide a greater PCB height ($height_{PCB}$) than the 1 mm instances, the 2 mm boards sometimes require less additional space ($width_{PCB\_SR}$) to accommodate the extra shift registers, which indirectly offsets the initial area cost of using larger electrodes. This occurs, for example, for *FPPC_4_DA* and *FPPC_8_DA*. The extra cost for *15x15_DA* is due to an additional layer (3 vs. 2 layers) for the smaller 1 mm case, which is due to its lower orthogonal capacity.

4.6.5 - PERFORMANCE

This section analyzes the performance of our FPPC-DMFB design when compared to state-of-the art direct addressing (with topologies) and pin-constrained DMFBs.

4.6.5.1 - COMPARISON TO GENERAL DMFB

We first compare our implementation to the most recent programmable direct-addressing DMFB design [31]. We run a set of 10 assays based on the PCR, in-vitro diagnostics [73] and protein-split benchmarks [33] (see the **Appendix** for more details). List scheduling [32][75] is used for PCR and in-vitro assays, while path scheduling [33] is used to schedule the protein-split assays. **Table 4-5** shows the number of seconds spent both routing and executing assay operations; the total time is the sum of the two. Results are also given for the number of usable electrodes (i.e., those tied to a control pin), number of external control pins, and cost; 1mm electrodes are assumed used and the same metrics used in **Table 4-4** are used here again to compute the cost. For each design, we start with the smallest DMFB that can fit at least 4 mixing modules and increase the size to add more resources only if the scheduler cannot yield a schedule for the current size; as seen in **Table 4-5**, for *ProteinSplit3-4*, the array dimensions had to be increased to execute the assay for one or both of the DMFBs.

Direct-Addressing DMFB (DA) vs. Field-Programmable Pin-Constrained DMFB (FP)															
Benchmarks	Array Dim.		# Electrodes Used		# Pins		Routing Time (s)		Operations Time (s)		Total Time (s)		Total Cost ($)	
	DA	FP	DA	FP	DA	FP	DA	FP	DA	FP	DA	FP	DA	FP
PCR	15x11	10x16	165	82	165	36	0.4	1.4	11	11	11.4	12.4	$5.19	$1.41
InVitro1	15x11	10x16	165	82	165	36	0.6	2.2	14	14	14.6	16.2	$5.19	$1.41
InVitro2	15x11	10x16	165	82	165	36	1.0	3.2	20	18	21.0	21.2	$5.19	$1.41
InVitro3	15x11	10x16	165	82	165	36	1.4	4.7	24	19	25.4	23.7	$5.19	$1.41
InVitro4	15x11	10x16	165	82	165	36	2.0	6.1	30	23	32.0	29.1	$5.19	$1.41
InVitro5	15x11	10x16	165	82	165	36	2.4	8.1	44	29	46.4	37.1	$5.19	$1.41
ProteinSplit1	15x11	10x16	165	82	165	36	1.2	2.2	54	54	55.2	56.2	$5.19	$1.41
ProteinSplit2	15x11	10x16	165	82	165	36	2.9	5.0	102	70	104.9	75.0	$5.19	$1.41
ProteinSplit3	15x11	10x18	165	86	165	38	5.1	10.9	207	125	212.1	135.9	$5.19	$1.59
ProteinSplit4	15x15	10x30	225	146	225	65	12.7	28.5	235	151	247.7	179.5	$6.39	$7.42
Avg. Normalized FP Improvement: (> 1 is improvement)			**1.96**		**4.45**		**0.39**		**1.29**		**1.16**		**3.36**	

Table 4-5: Experimental results comparing the direct-addressing DMFB (DA) [31] with our FPPC-DMFB (FP).

The bottom row of **Table 4-5** shows the average improvement of our field-programmable DMFB compared to the direct-addressing DMFB. We calculated this metric by computing the improvement of FP over DA (baseline) for each benchmark and then averaging these values over the entire set of benchmarks. Any value over 1 means FP is an improvement. Notice that, although FP's average routing time is 61% slower, its average operation time is 29% faster. The shorter run times are experienced on larger assays where storage is required; in these assays, FP takes advantage of the fact that its storage and mixing modules are separate. Thus, FP does not have to utilize any mixers to store droplets, leaving mixing resources free to perform more useful work which will help complete the assay more quickly. On average, FP gains an average 16% speedup in total execution time while reducing the cost by 3-4×.

4.6.5.2 - Comparison To Pin-Constrained DMFBs

Table 4-6 presents results for the 8 pin-constrained DMFBs (*ZHAO_XXX* [89] and *LUO_XXX* [51]) discussed in **'4.6.1 - Benchmarks'**. Many differences exist between these designs and the FPPC-DMFB; most notably, they are assay specific while our design is field-programmable. Also, they use linear array mixing modules, which have longer latencies than the 4×2 mixers we employ in our FPPC-DMFB implementation. Thus, the schedules are different, and it is unclear if their reported results include droplet routing times, as their primary objective was to reduce the cost of their pin-constrained DMFBs by reducing the pin-count. **Table 4-6** is reproduced from Refs. [89].

Pin-Constrained Results Zhao [89] vs. Luo [51]						
Benchmark	Array Dim.	# Electrodes Used	# Pins		Total Time (s)	
			[89]	[51]	[89]	[51]
PCR	15x15	62	14	22	20	30
InVitro1	15x15	59	25	21	73	90
ProteinSplit3	15x15	54	26	20	150	170
Multi-Function	15x15	81	37	27	150	170

Table 4-6: Results from Zhao's [89] and Luo's [51] pin-constrained designs for chips which can run PCR, InVitro1, ProteinSplit3 and a multi-functional chip which can run all three.

Table 4-7 reports the performance and pin-count for *PCR*, *InVitro1*, and *ProteinSplit3* for FPPC-DMFBs of varying sizes. For *PCR* and *InVitro1*, execution times decrease as the DMFB size (and thus the number of available modules) increase, saturating at 10×21. For larger DMFBs, performance degrades slightly due to longer routing times from the taller vertical routing channel/bus.

Total Assay Times for Increasing FPPC-DMFB Array Size						
Array Dim.	#Module (Mix/SSD)	# Electrodes Used	# Pins	Total Time(s)		
				PCR	InVitro1	ProteinSplit3
10x10	2/3	52	26	18.39	18.72	-
10x13	3/4	66	30	15.50	16.90	-
10x16	4/6	82	36	12.44	16.24	196.83
10x21	5/8	100	49	12.69	16.64	197.37
10x24	6/10	116	55	12.84	16.88	197.77

Table 4-7: The three benchmark assays from Xu [82] and Luo [51] (PCR, InVitro1 and ProteinSplit3) on various sizes of the FPPC-DMFB.

The *ProteinSplit3* assay requires 5 droplets to be stored at several instances during the assay; thus, the 10×16 array with 6 SSD modules (5 available to the scheduler) is the smallest compatible device. The total execution time remains steady, regardless of resources (we also tested on a 12×81 DMFB with abundant resources) at ~197s. In this case, the total execution time is not limited by resource availability, but by the 7s droplet dispense times (see the **Appendix**). It is unclear what droplet dispense times were assumed in prior work [51][82]; reducing the dispense times to 2s instead of 7s reduces the assay execution time to approximately 109s for the 10×24 FPPC-DMFB.

Luo and Chakrabarty's pin assignment scheme [51] theoretically provides some flexibility, as two droplets are guaranteed to be able to move without interfering with one another; however, they did not provide details on how synthesis was performed, so it is difficult to provide a direct comparison. In contrast, the different sizes of our FPPC-DMFB reported in **Table 4-7** are the same generic design, but with a different number of resources. As seen in **Table 4-5**, if we pick dimensions of reasonable size (10×18), we can run all three assays in **Table 4-6**, as well as others, due to the field-programmable nature of our design.

4.7 - CONCLUSION

This chapter has extended the initial development of FPPC-DMFBs [29] with an enhanced design to facilitate more efficient wire routing, and has presented the first cost estimates, in terms of US dollars, for PCB fabrication for DMFBs. A complete synthesis flow, which addresses architectural issues that are specific to the FPPC-DMFB, has been presented, along with a detailed description of its general-purpose, as opposed to assay-specific, capabilities.

We show that, in terms of performance and overall assay length, our general-purpose FPPC-DMFB design is competitive with prior pin-constrained assay-specific DMFBs, and in many cases, superior to state-of-the-art direct-addressing designs; we also demonstrate that the FPPC-DMFB is less expensive than previous pin-constrained designs, which are optimized for cost and PCB design. Thus, the flexibility provided by the FPPC-DMFB is unmatched by prior pin-constrained DMFBs, which were optimized for specific assays, and offers a significant advancement in terms of programmability at a lower overall per-unit cost.

CHAPTER 5 CONCLUSION

In this dissertation, we presented several novel topological designs for digital microfluidic biochips (DMFBs) and demonstrates the advantages of using both virtual and physical topologies in the synthesis and design of droplet-based DMFBs. As discussed in the introduction, a DMFB performs biochemical reactions, called assays, by manipulating tiny droplets on a 2D array of electrodes. Droplets are maneuvered orthogonally (i.e., up, down, left, right) across the surface of the electrode array, one electrode at a time; these movements, performed in the proper sequence, form the basic building blocks used to describe assays (e.g., mix, merge, split, transport, store, etc.).

Direct-addressing DMFBs offer the highest flexibility since each electrode is individually addressable, which means droplets can be transported and operations can be performed almost anywhere within the 2D plane of electrodes. In an attempt to minimize completion time (along with other secondary metrics), most prior works employed long-running optimization algorithms to generate a sequence of electrode activations to perform a particular assay. Thus, given that many of these algorithms could take minutes, hours or even days to complete, the algorithms were performed at design time and the resultant electrode activation sequence was packaged into the DMFB as a type of microfluidic binary. Unfortunately, the binaries produced from this static offline compilation flow were typically deterministic in nature. Thus, DMFB devices running these binaries had no way to respond to live feedback from error detection (e.g., an uneven split of a droplet) or uncertainty (e.g., mix until solution turns green), making it extremely difficult, if not impossible, to know if an assay completed properly.

DMFBs have already started to incorporate sensors, and reliable DMFBs of the future will be cyber-physical systems in which software constructs must be able to dynamically respond to live feedback from onboard hardware. Thus, instead of optimizing algorithms for application-specific DMFBs, we aim to contribute to the development of DMFBs that are programmable in nature and can respond to live-feedback. Toward this goal, this work begins by presenting the idea of dynamic interpretation in **CHAPTER 2**. As opposed to static offline compilation, dynamic interpretation manages the DMFBs resources in an online environment (while the assay is being executed). An interpreter can evaluate live feedback, perform synthesis computations (e.g., scheduling, placement, routing), and as a result, decide how to complete an assay in light of runtime errors and uncertainties. A number of language constructs were introduced that allow for control flow; this allows for error handling, uncertainty and simple if-else type statements which enable an entirely new class of non-deterministic assays. A dynamic interpreter must perform synthesis and re-synthesis steps (in response to live feedback) very quickly because any time spent on computations will result in a longer completion time for the assay being executed.

In this dissertation, we defend the usage of virtual topologies for DMFB synthesis and present several novel designs. A virtual topology utilizes the same flexible 2D array of electrodes, but restricts operations to be performed exclusively overtop fixed groups of electrodes, called modules; the virtual topology also requires certain electrodes to always be kept free for droplet transportation (i.e., these electrodes can never be used to perform an operation). Although **CHAPTER 2** does present the basic fundamentals of a virtual topology with an original design, **CHAPTER 3** presents an optimized, novel virtual topology. We present a complete microfluidic synthesis solution, discussing how we solve typically-overlooked problems such as module synchronization and droplet routing deadlock. We also demonstrate how a virtual topology prunes the search space for the major steps of synthesis (i.e., scheduling, placement, routing), allowing for algorithms which can generate quality solutions for these NP-complete problems in just milliseconds, enabling dynamic interpretation with minimal computational overhead.

Our results in **CHAPTER 3** show that, when compared to the long-running optimal algorithms of past static compilation flows, our algorithms can leverage the virtual topology to yield competitive solutions in orders-of-magnitude less time. One of the common arguments against virtual topologies is that they limit the flexibility of the DMFBs and that more compact designs could produce better results. Although we never claim optimality, our results clearly show the strength of our virtual topology design when compared to more ambitious scheduling and placement algorithms. The results show that, in fact, other algorithms can generate more compact schedules and placements; however, these solutions often yield failures during the droplet routing phase because no clear path was left between each source and destination. In contrast, the virtual topology guarantees that a valid schedule will always yield a valid placement, routing and, in turn, a valid overall solution. Our results support these claims and demonstrate how past methods tend to yield intermediate solutions that cause failures down the road. Thus, we trade optimality in area (which often leads to droplet routing failures) for fast algorithmic runtimes and overall synthesis reliability.

In **CHAPTER 4** we present a physical pin-constrained topology to address the cost problem of DMFBs. Classical direct-addressing DMFBs claim the greatest flexibility of any DMFB class because each electrode is wired directly to its own microcontroller signal and can be activated independently of any other electrode; however, this architecture requires a high number of wires to be routed below the electrode array (connecting electrodes to the edge of the DMFB), leading to an increasing number of PCB layers, and ultimately, a higher fabrication cost. Pin-constrained DMFBs, in contrast, connect a single microcontroller output to multiple electrodes. Prior to our work, pin-constrained DMFBs were designed and optimized to execute a single or small set of assays. This meant that, once the microfluidic device was manufactured, it could only be used to run the particular assay or experiment it was designed to perform; however, the primary benefit and driving factor for these pin-constrained devices is that they are cheaper than standard direct addressing DMFBs.

We introduce the field-programmable, pin-constrained (FPPC-)DMFB, which is the first general-purpose, pin-constrained DMFB with no assay-specific restrictions. The FPPC-DMFB utilizes a physical topology to restrict droplet transport and generic operations to specific regions of the DMFB, allowing for a flexible device that can operate basic microfluidic instructions. In addition, we present intelligent pin mapping and wire routing solutions which solve the DMFB cost problem.

Our results provide the first detailed cost analysis for DMFB PCB fabrication and offer new insights on the relationship between PCB layer count, pin count and cost. Altogether, we show that PCB fabrication costs for our new FPPC-DMFB design is less expensive than prior pin-constrained DMFBs; furthermore, our design is much more flexible since it can perform any sequence of generic operations instead of specific assays. In terms of assay completion time, the results also show that the FPPC-DMFB is competitive with prior pin-constrained designs, and often outperforms state-of-the-art direct addressing synthesis algorithms.

As microfluidic devices mature, they continue to offer a viable platform for the automation and miniaturization of biochemistry and could revolutionize the natural sciences and medical fields. DMFB devices must be inexpensive, programmable and dynamic (able to respond to live feedback) before those outside of select research labs will begin to realize the potential of microfluidic technologies. Toward this end, we show that both virtual and physical topologies can simplify algorithmic and DMFB hardware complexity, allowing for general-purpose, inexpensive devices which can efficiently respond to live feedback. In this dissertation, we contribute to the development and ubiquitous adoption of digital microfluidic technology.

REFERENCES

[1] Advanced Circuits. (2014, Apr. 4). Instant Quote [Online]. Available: http://www.4pcb.com

[2] Atmel. Atmel ATmega 1284 Microcontroller [Online]. Available: http://www.atmel.com/devices/atmega1284.aspx

[3] Fairchild Semiconductor. 74VHC595 – 8-Bit Shift Register with Output Latches [Online]. Available: http://www.fairchildsemi.com

[4] Microfluidics Lab at the University of California, Riverside. Digital Microfluidic Biochip Simulator [Online]. Available: http://microfluidics.cs.ucr.edu

[5] Mouser Electronics. Fairchild Semiconductor 74VHC595MTC [Online]. Available: http://www.mouser.com

[6] M. Alistar et al., "Synthesis of biochemical applications on digital microfluidic biochips with operation variability," in *Proceedings of the Symposium on Design, Test, Integration, and Packaging of MEMS/MOEMS*, 2010, pp. 350-357.

[7] M. Alistar et al., "Online synthesis for error recovery in digital microfluidic biochips with operation variability," in *Proceedings of the Symposium on Design, Test, Integration, and Packaging of MEMS/MOEMS*, 2012, pp. 25-27.

[8] A. Amin et al., "Aquacore: a programmable architecture for microfluidics," in *Proceedings of the International Symposium on Computer Architecture*, 2007, pp. 254-265.

[9] V. Ananthanarayanan and W. Theis, "Biocoder: a programming language for standardizing and automating biology protocols," *Journal of Biological Engineering*, vol. *4*, no., Article #13, Nov., 2010.

[10] K. Bazargan et al., "Fast template placement for reconfigurable computing," *IEEE Design and Test of Computers*, vol. 17, no. 1, pp. 68-83, Jan., 2000.

[11] K. F. Bohringer, "Modeling and controlling parallel tasks in droplet-based microfluidic systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. *25*, no. 2, pp. 334-344, Feb., 2006.

[12] J-W. Chang et al., "An ILP-based routing algorithm for pin-constrained EWOD chips with obstacle avoidance," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 32, no. 11, pp. 1655-1667, Nov., 2013.

[13] J-W. Chang et al., "Integrated fluidic-chip co-design methodology for digital microfluidic biochips," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 32, no. 2, pp. 216-227, Feb., 2013.

[14] S. Chatterjee et al., "Multi-objective optimization algorithm for efficient pin-constrained droplet routing technique in digital microfluidic biochip," in *Proceedings of the International Symposium on Quality Electronic Design*, 2013, pp. 252-256.

[15] G.-M. Chiu, "The odd-even turn model for adaptive routing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 11, no. 7, pp. 729-738, Jul., 2000.

[16] M. Cho and D. Z. Pan, "A high-performance droplet routing algorithm for digital microfluidic biochips," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 10, pp. 1714-1724, Oct., 2008.

[17] W. J. Dally and C. L. Seitz, "Deadlock-free message routing in multiprocessor interconnection networks," *IEEE Transactions Computers*, vol. C-36, no. 5, pp. 547-553, May, 1987.

[18] W. J. Dally and B. P. Towles, "Principles and practices of interconnection networks," *Morgan Kaufmann*, 2004.

[19] M. Dhindsa et al., "Reliable and low-voltage electrowetting on thin parylene films," *Thin Solid Films*, vol. 519, no. 10, pp. 3346-3351, Mar., 2011.

[20] J. Ding et al., "Scheduling of microfluidic operations for reconfigurable two-dimensional electrowetting arrays," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, no. 10, pp. 1463-1468, Dec., 2001.

[21] R. Evans et al., "Optical detection heterogeneously integrated with a coplanar digital microfluidic lab-on-a-chip platform," in *Proceedings of the IEEE Sensors Conference*, 2007, pp. 423-426.

[22] R. B. Fair et al., "Chemical and biological applications of digital-microfluidic devices," *IEEE Design and Test of Computers*, vol. 24, no. 1, pp. 10-24, Feb., 2007.

[23] S-K. Fan et al., "Manipulation of multiple droplets on an N×M grid by cross-reference EWOD driving scheme and pressure-contact packaging," in *Proceedings of the IEEE MEMS Conference*, Kyoto, Japan, 2003, 694-697.

[24] H. Gabow, "Path-based depth-first search for strong and biconnected components," *Information Processing Letters*, vol. 74, no. 3-4, pp. 107-114, May, 2000.

[25] C. J. Glass and L. M. Ni, “The turn model for adaptive routing,” *Journal of the ACM 1*, vol. 41, no. 5, pp. 874-902, Sep., 1994.

[26] E. J. Griffith et al., “Performance characterization of a reconfigurable planar-array digital microfluidic systems,” *Design Automation Methods and Tools for Microfluidic-Based Biochips*, Springer-Verlag, pp. 329-356.

[27] D. Grissom et al., “A digital microfluidic biochip synthesis framework,” in *Proceedings of the International Conference on VLSI*, Santa Cruz, CA, 2012.

[28] D. Grissom et al., “Interpreting assays with control flow on digital microfluidic biochips,” *ACM Journal on Emerging Technologies in Computing Systems*, vol. 10, no. 3, Article #24, Apr., 2014.

[29] D. Grissom and P. Brisk, “A field-programmable pin-constrained digital microfluidic biochip,” in *Proceedings of the Design Automation Conference*, Austin, TX, 2013.

[30] D. Grissom and P. Brisk, “A high-performance online assay interpreter for digital microfluidic biochips,” in *Proceedings of the Great Lake Symposium on VLSI*, Salt Lake City, UT, 2012, pp. 103-106.

[31] D. Grissom and P. Brisk, “Fast online synthesis of digital microfluidic biochips,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 33, no. 3, pp. 356-369, Mar., 2014.

[32] D. Grissom and P. Brisk, "Fast online synthesis of generally programmable digital microfluidic biochips," in *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis*, Tampere, Finland, 2012, pp. 413-422.

[33] D. Grissom and P. Brisk, “Path scheduling on digital microfluidic biochips,” in *Proceedings of the Design Automation Conference*, San Francisco, CA, 2012, pp. 26-35.

[34] B. Hadwen et al., “Programmable large area digital microfluidic array with integrated droplet sensing for bioassays,” *Lab-on-a-Chip*, vol. 12, no. 18, pp. 3305-3313, May, 2012.

[35] A. Hashimoto and J. Stevens, “Wire routing by optimizing channel assignment within large apertures,” in *Proceedings of the 8th Workshop on Design Automation*, 1971, pp. 155-169.

[36] T. Ho et al., "Digital microfluidic biochips: recent research and emerging challenges," in *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis*, Taipei, Taiwan, 2011, pp. 335-343.

[37] J. Hopcroft and R. Tarjan, “Algorithm 447: efficient algorithms for graph manipulation,” *Communications of the ACM*, vol. 16, no. 6, pp. 372-378, Jun., 1973.

[38] T-W. Huang et al., “A fast routability- and performance-driven droplet routing algorithm for digital microfluidic biochips,” in *Proceedings of the International Conference on Computer-Aided Design*, Lake Tahoe, CA, 2009, pp. 445-450.

[39] T-W. Huang et al., “A network-flow based pin-count aware routing algorithm for broadcast-addressing EWOD chips,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 12, pp. 1786-1799, Dec., 2011.

[40] T-W. Huang et al., “Progressive network-flow based power-aware broadcast addressing for pin-constrained digital microfluidic biochips,” in *Proceedings of the Design Automation Conference*, San Diego, CA, 2011, pages 741-746.

[41] T-W. Huang et al., “Reliability-oriented broadcast electrode-addressing for pin-constrained digital microfluidic biochips,” in *Proceedings of the International Conference on Computer-Aided Design*, San Jose, CA, 2012, pages 448-455.

[42] T-W. Huang and T-Y. Ho, “A two-stage integer linear programming-based droplet routing algorithm for pin-constrained digital microfluidic biochips,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 2, pp. 215-228, Feb., 2011.

[43] A. Kahn, “Topological sorting of large networks,” *Communications of the ACM*, vol. 5, no. 11, pp. 558-562, Nov., 1962.

[44] F. J. Kurdahi and A. C. Parker, “REAL: a program for REgister Allocation,” in *Proceedings of the Design Automation Conference*, Miami Beach, FL, 1987, pp. 210-215.

[45] C. Liao and S. Hu, “Multiscale variation-aware techniques for high-performance digital microfluidic lab-on-a-chip component placement,” *IEEE Transactions on NanoBioscience*, vol. 10, no. 1, pp. 51-58, Mar., 2011.

[46] C. C-Y. Lin and Y-W. Chang, “Cross-contamination aware design methodology for pin-constrained digital microfluidic biochips,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 6, pp. 817-828, Jun., 2006.

[47] C. C-Y. Lin and Y-W. Chang, “ILP-based pin-count aware design methodology for microfluidic biochips,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 29, no. 9, pp. 1315-1327, Sep., 2010.

[48] R. L'Orsa et al., "Detailed droplet routing and complexity characterization on a digital microfluidic biochip," in *Proceedings of the SPIE*, Orlando, FL, 2009.

[49] L. Luan et al., "Integrated optical sensor in a digital microfluidic platform," *IEEE Sensors Journal*, 2008, vol. 8, no. 5, pp. 628-635, May, 2008.

[50] L. Luo and S. Akella, "Optimal scheduling of biochemical analyses on digital microfluidic systems," in *Proceedings of the Conference on Intelligent Robots and Systems,* San Diego, CA, 2007, pp. 3151-3157.

[51] Y. Luo and K. Chakrabarty, "Design of pin-constrained general-purpose digital microfluidic biochips," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 32, no. 9, pp. 1307-1320, Sep., 2013.

[52] Y. Luo et al., "A cyberphysical synthesis approach for error recovery in digital microfluidic biochips," in *Proceedings of Design Automation and Test in Europe*, Dresden, Germany, 2012, pp. 1239-1244.

[53] Y. Luo et al., "Design of cyberphysical digital microfluidic biochips under completion-time uncertainties in fluidic operations," in *Proceedings of the Design Automation Conference,* Austin, TX, 2013.

[54] Y. Luo et al., "Dictionary-based error recovery in cyberphysical digital microfluidic biochips," in *Proceedings of the International Conference on Computer-Aided Design*, San Jose, CA, 2012, pp. 369-376.

[55] Q. Ma et al., "A negotiated congestion based router for simultaneous escape routing," in *Proceedings of the International Symposium on Quality Electronic Design,* San Jose, CA, 2010, pp. 606-610.

[56] E. Maftei et al., "Tabu search-based synthesis of digital microfluidic biochips with dynamically reconfigurable non-rectangular devices," *Design Automation for Embedded Systems,* vol. 14, no. 3, pp. 287-307, Sep., 2010.

[57] J. Melin and S. R. Quake, "Microfluidic Large-Scale Integration: The Evolution of Design Rules for Biological Automation," *Annual Review of Biophysics and Biomolecular Structure*, vol. 36, pp. 213-231, Jun., 2007.

[58] H. Moon et al., "Low voltage electrowetting on dielectric," *Journal of Applied Physics*, vol. 92, no. 7, pp. 4080-4087, Sep., 2002.

[59] R. Mukherjee, et al., "A heuristic method for co-optimization of pin assignment and droplet routing in digital microfluidic biochip," in *Proceedings of the International Conference on VLSI Design*, Hyderabad, India, 2012, pages 227-232.

[60] J. Noh et al., “Toward active-matrix lab-on-a-chip: programmable electrofluidic control enabled by arrayed oxide thin film transistors,” *Lab-on-a-Chip*, vol. 12, no. 2, pp. 353-360, Dec., 2011.

[61] K. O’Neal et al., “Force-directed list scheduling for digital microfluidic biochips,” in *Proceedings of the International Conference on VLSI*, Santa Cruz, CA, 2012.

[62] P. Paik et al., "Rapid droplet mixers for digital microfluidic systems," *Lab on a Chip*, vol. 3, no. 4, pp. 253-259, Sep., 2003.

[63] M. G. Pollack et al., “Electrowetting-based actuation of droplets for integrated microfluidics,” *Lab on a Chip*, vol. 2, pp. 96-101, Mar., 2002.

[64] H. Ren et al., “Automated on-chip droplet dispensing with volume control by electro-wetting actuation and capacitance metering,” *Sensors and Actuators B: Chemical*, vol. 98, no. 2-3, pp. 319-327, 2004.

[65] A. J. Ricketts et al., “Priority scheduling in digital microfluidics-based biochips,” in *Proceedings of Design Automation and Test in Europe*, Munich, Germany, 2006, pp. 329-334.

[66] P. Roy et al., “A novel droplet routing algorithm for digital microfluidic biochips,” in *Proceedings of the Great Lakes Symposium on VLSI*, Providence, RI, 2010, pp. 441-446.

[67] P. Roy et al., “Two-level clustering-based techniques for intelligent droplet routing in digital microfluidic biochips,” *Integration, the VLSI Journal*, vol. 45, no. 3, pp. 316-330, Jun., 2012.

[68] S. Roy et al., “Congestion-aware layout design for high-throughput digital microfluidic biochips,” *ACM Journal on Emerging Technologies in Computing Systems*, vol. 8, no. 3, pp. 17.1-17.23, Aug., 2012.

[69] K. Singha et al., “Method of droplet routing in digital microfluidic biochip,” in *Proceedings of the IEEE/ASME International Conference on Mechatronics and Embedded Systems and Applications*, Qingdao, China, 2010, pp. 251-256.

[70] J. Soukup, "Fast maze router," in *Proceedings of the Design Automation Conference*, Las Vegas, NV, 1978, pp. 100-102.

[71] V. Srinivasan et al., “A digital microfluidic biosensor for multianalyte detection,” in *Proceedings of the International Conference on Micro Electro Mechanical Systems*, 2003, pp. 327–330.

[72] V. Srinivasan et al. “An integrated digital microfluidic lab-on-a-chip for clinical diagnostics on human physiological fluids,” *Lab on a Chip*, no. 4, pp. 310-315, May, 2004.

[73] F. Su and K. Chakrabarty, “Architectural-level synthesis of digital microfluidic-based biochips,” in *Proceedings of the International Conference on Computer Aided Design*, San Jose, CA, 2004, pp. 223-228.

[74] F. Su and K. Chakrabarty, “Benchmarks for digital microfluidic biochip design and synthesis,” *Duke Univ., Dept. of Electrical and Computer Engineering Website*. http://www.ee.duke.edu/~fs/Benchmark.pdf.

[75] F. Su and K. Chakrabarty, “High-level synthesis of digital microfluidic biochips,” *ACM Journal on Emerging Technologies in Computing Systems*, vol. 3, no. 4, Article #16, January, 2008.

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

[77] F. Su and K. Chakrabarty, “Unified high-level synthesis and module placement for defect-tolerant microfluidic biochips,” in *Proceedings of the Design Automation Conference*, Anaheim, CA, 2005, pp. 825-830.

[78] F. Su et al., “Droplet routing in the synthesis of digital microfluidic biochips,” in *Proceedings of Design Automation and Test in Europe*, Munich, Germany, 2006, pp. 1-6.

[79] K.-H. Tseng et al., “A network-flow based valve-switching aware binding algorithm for flow-based microfluidic biochips,” in *Proceedings of the Asia South Pacific Design Automation Conference*, Yokohama, Japan, 2013, pp. 213-218.

[80] Z. Xiao and E. F. Y. Young, "Placement and routing for cross-referencing digital microfluidic biochips," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 7, pp. 1000-1010, Jul. 2011.

[81] T. Xu et al., “Defect-tolerant design and optimization of a digital microfluidic biochip for protein crystallization,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 29, no. 4, pp. 552-565, Apr., 2010.

[82] T. Xu and K. Chakrabarty, “Broadcast electrode-addressing for pin-constrained multi-functional digital microfluidic biochips,” in *Proceedings of Design Automation Conference*, Anaheim, CA, 2008, pp. 173-178.

[83] T. Xu and K. Chakrabarty, "Integrated droplet routing in the synthesis of microfluidic biochips," in *Proceedings of the Design Automation Conference*, San Diego, CA, 2007, pp. 948-953.

[84] T. Xu et al., "Automated design of pin-constrained digital microfluidic biochips under droplet-interference constraints," *ACM Journal on Emerging Technologies in Computing Systems*, vol. 3, no. 3, Article #14, Nov., 2007.

[85] T. Yan and M. D. F. Wong, "Correctly modeling the diagonal capacity in escape routing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 31, no. 2, pp. 285-293, Feb., 2012.

[86] S-H. Yeh et al., "Voltage-aware chip-level design for reliability-driven pin-constrained EWOD chips," in *Proceedings of the International Conference on Computer-Aided Design*, San Jose, CA, 2012, pages 353-360.

[87] P-H. Yuh et al., "Placement of defect-tolerant digital microfluidic biochips using the T-tree formulation," *ACM Journal on Emerging Technologies in Computing Systems*, vol. 3, no. 3, Article #13, Nov., 2007.

[88] P-H. Yuh et al., "BioRouter: a network-flow-based routing algorithm for the synthesis of digital microfluidic biochips," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 11, pp. 1928-1941, Nov., 2008.

[89] Y. Zhao et al., "Broadcast electrode-addressing and scheduling methods for pin-constrained digital microfluidic biochips," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 7, pp. 986-999, Jul., 2011.

[90] Y. Zhao et al., "Integrated control-path design and error recovery in the synthesis of digital microfluidic lab-on-chip," *ACM Journal on Emerging Technologies in Computing Systems*, vol. 6, no. 3, Article #11, Aug., 2010.

[91] Y. Zhao et al., "Optimization techniques for the synchronization of concurrent fluidic operations in pin-constrained digital microfluidic biochips," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 20, no. 6, pp. 1132-1145, Jun., 2012.

[92] Y. Zhao and K. Chakrabarty, "Simultaneous optimization of droplet routing and control-pin mapping to electrodes in digital microfluidic biochips," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 31, no. 2, pp. 242-254, Feb., 2012.

APPENDIX

This section contains the DAGs for each of the assay benchmarks used in the experimental and simulation sections for **CHAPTER 2**, **CHAPTER 3**, and **CHAPTER 4**:

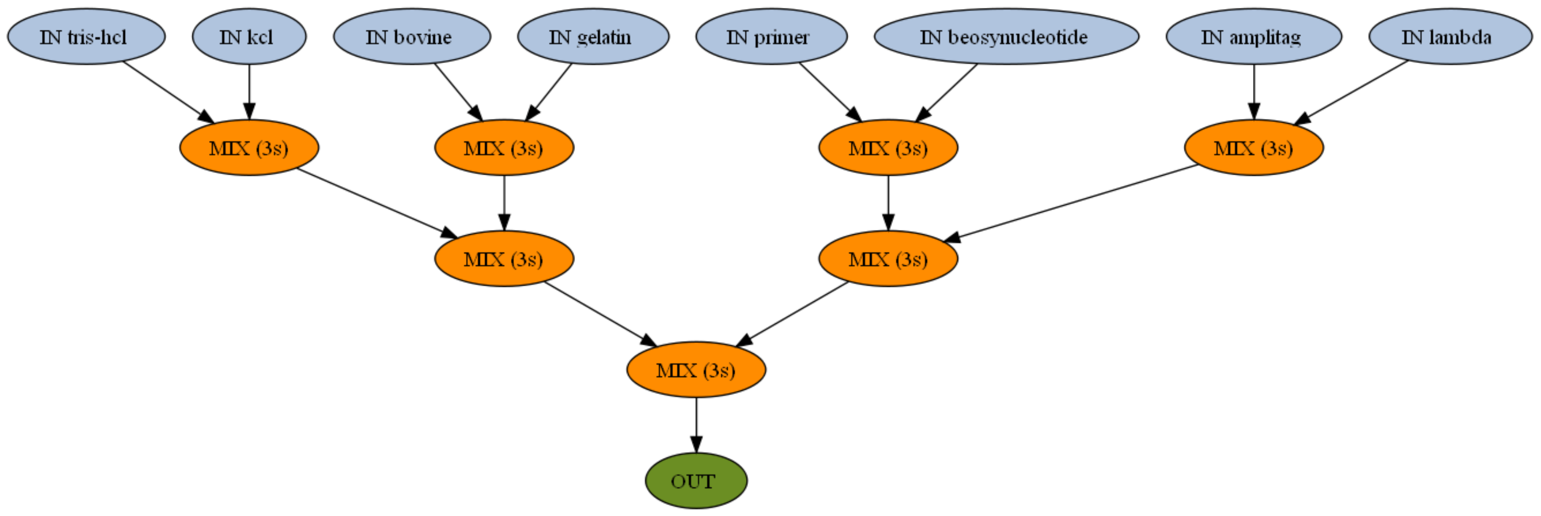


Figure A - 1: *PCR* Benchmark; DAG for a mixing tree for polymerase chain reaction (PCR) assay. All input operations (blue) are 2s and all mix operations (orange) are 3s in length (assuming 4x2 mix modules).

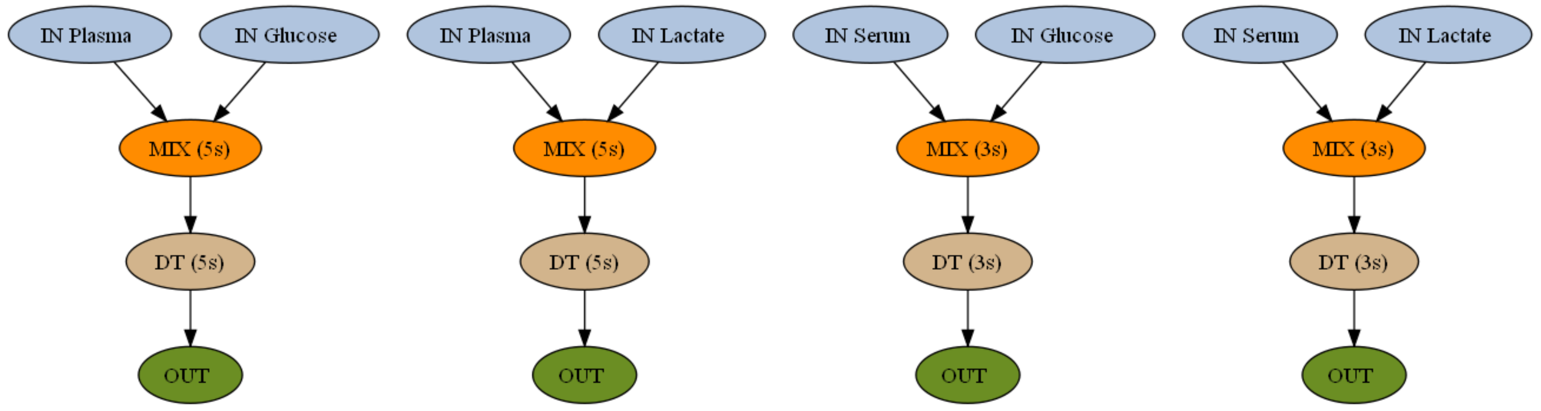


Figure A - 2: *InVitro1* Benchmark; DAG for an in-vitro diagnostics assay with 2 samples and 2 reagents. All input operations (blue) are 2s. All detect operations (tan) and mix operations (orange) are as marked (assuming 4x2 modules for mixes). Output operations (green) are instantaneous.

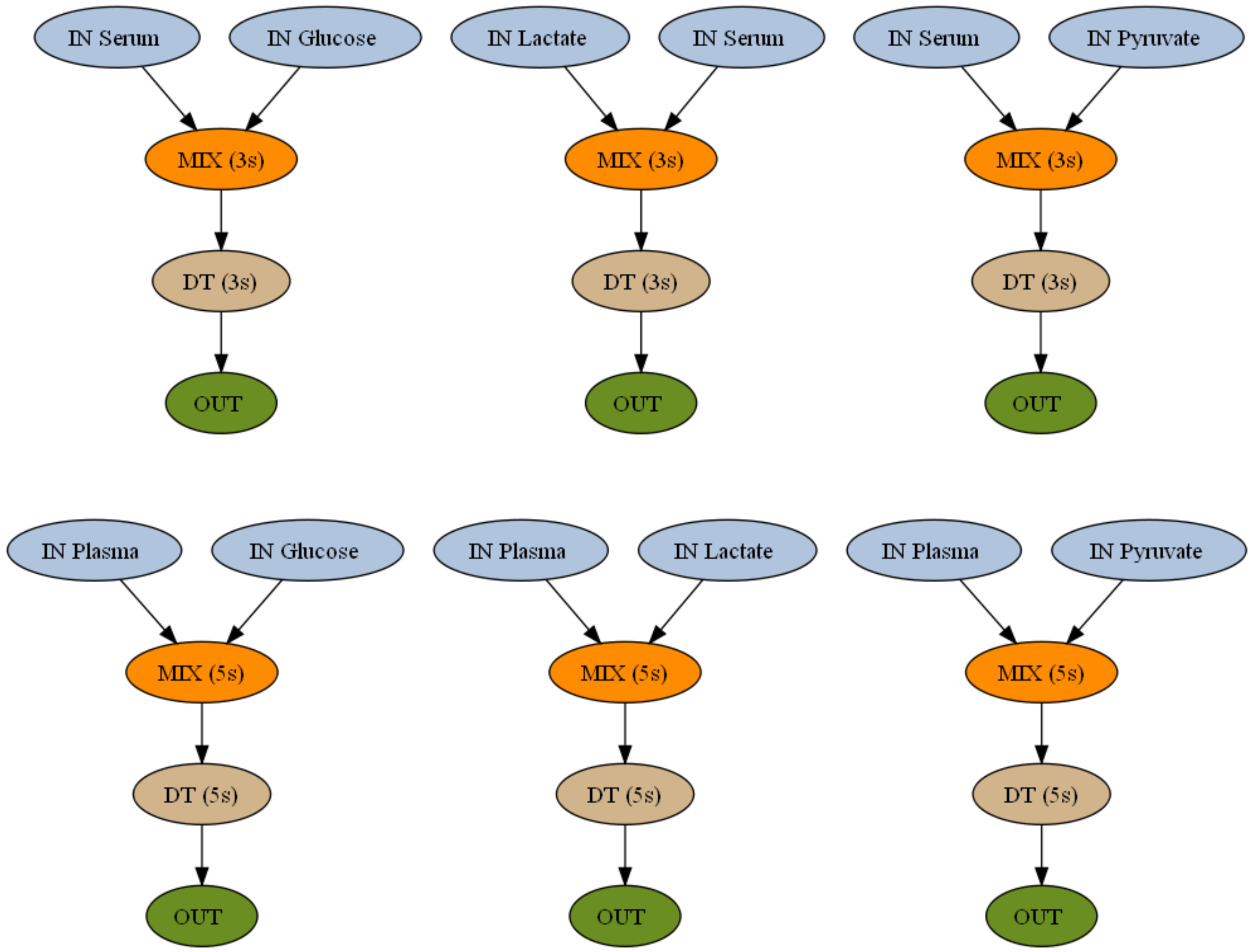


Figure A - 3: *InVitro2* Benchmark; DAG for an in-vitro diagnostics assay with 2 samples and 3 reagents. All input operations (blue) are 2s. All detect operations (tan) and mix operations (orange) are as marked (assuming 4x2 modules for mixes). Output operations (green) are instantaneous.

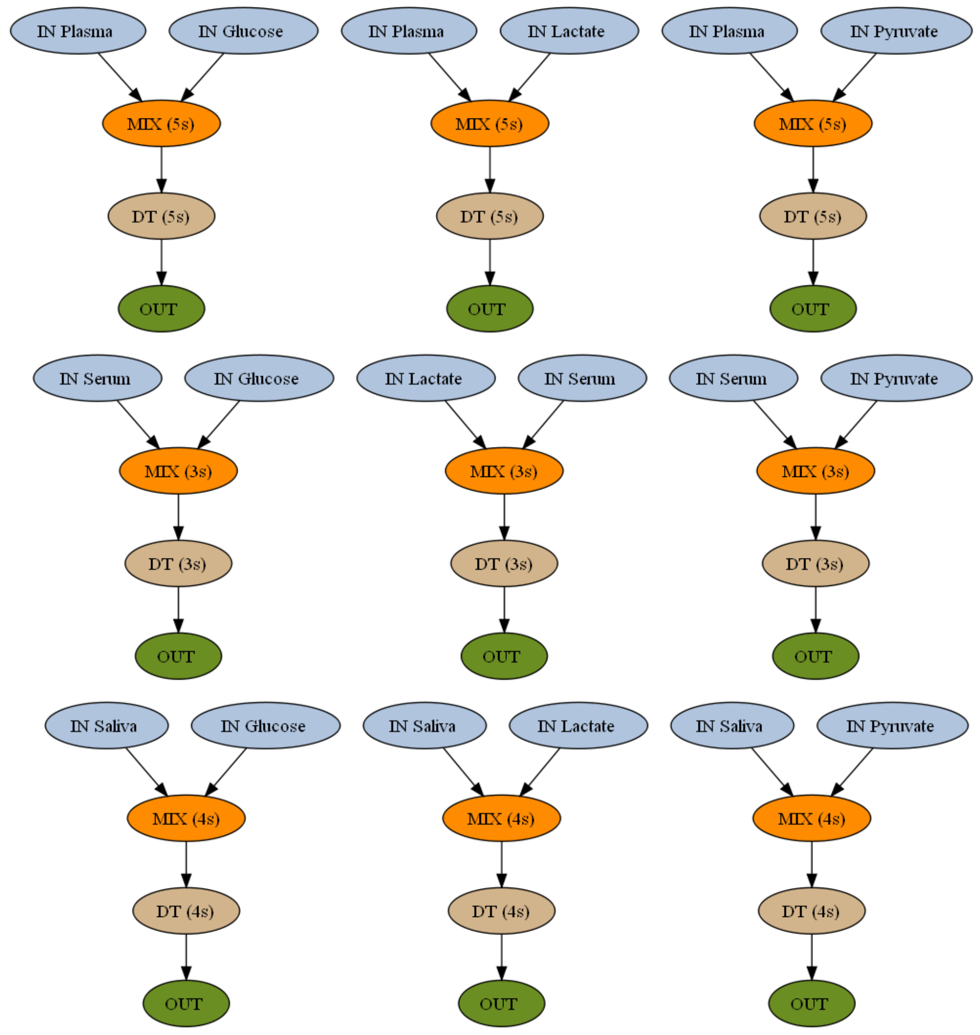


Figure A - 4: *InVitro3* Benchmark; DAG for an in-vitro diagnostics assay with 3 samples and 3 reagents. All input operations (blue) are 2s. All detect operations (tan) and mix operations (orange) are as marked (assuming 4x2 modules for mixes). Output operations (green) are instantaneous.

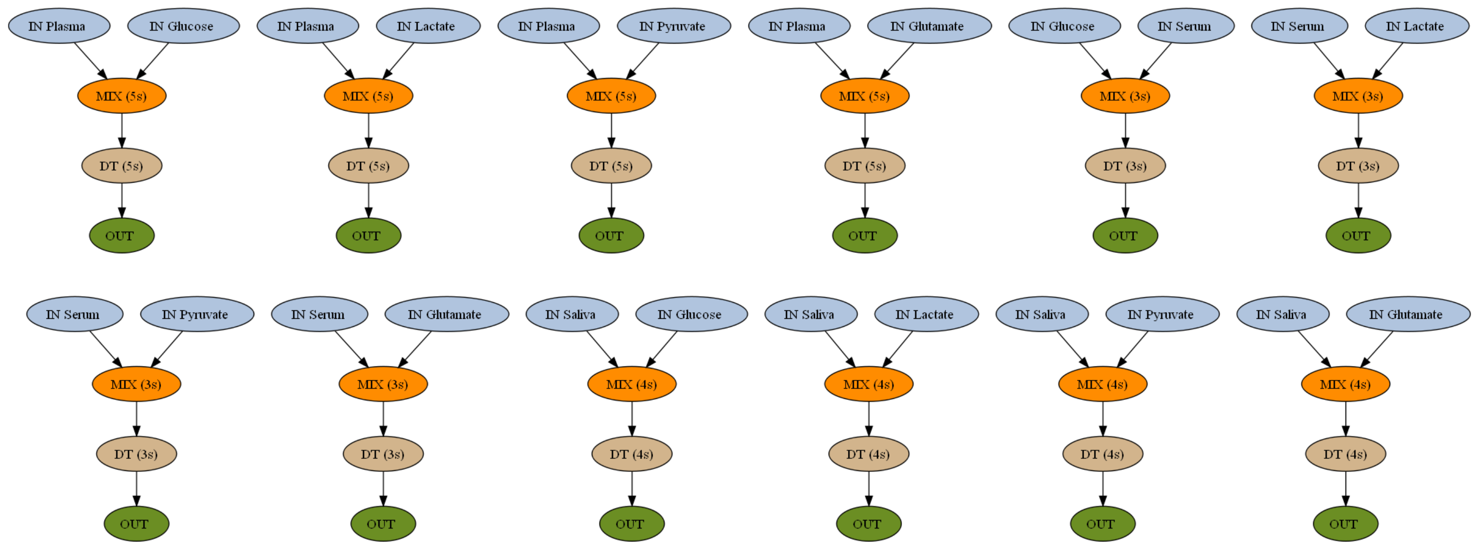


Figure A - 5: *InVitro4* Benchmark; DAG for an in-vitro diagnostics assay with 3 samples and 4 reagents. All input operations (blue) are 2s. All detect operations (tan) and mix operations (orange) are as marked (assuming 4x2 modules for mixes). Output operations (green) are instantaneous.

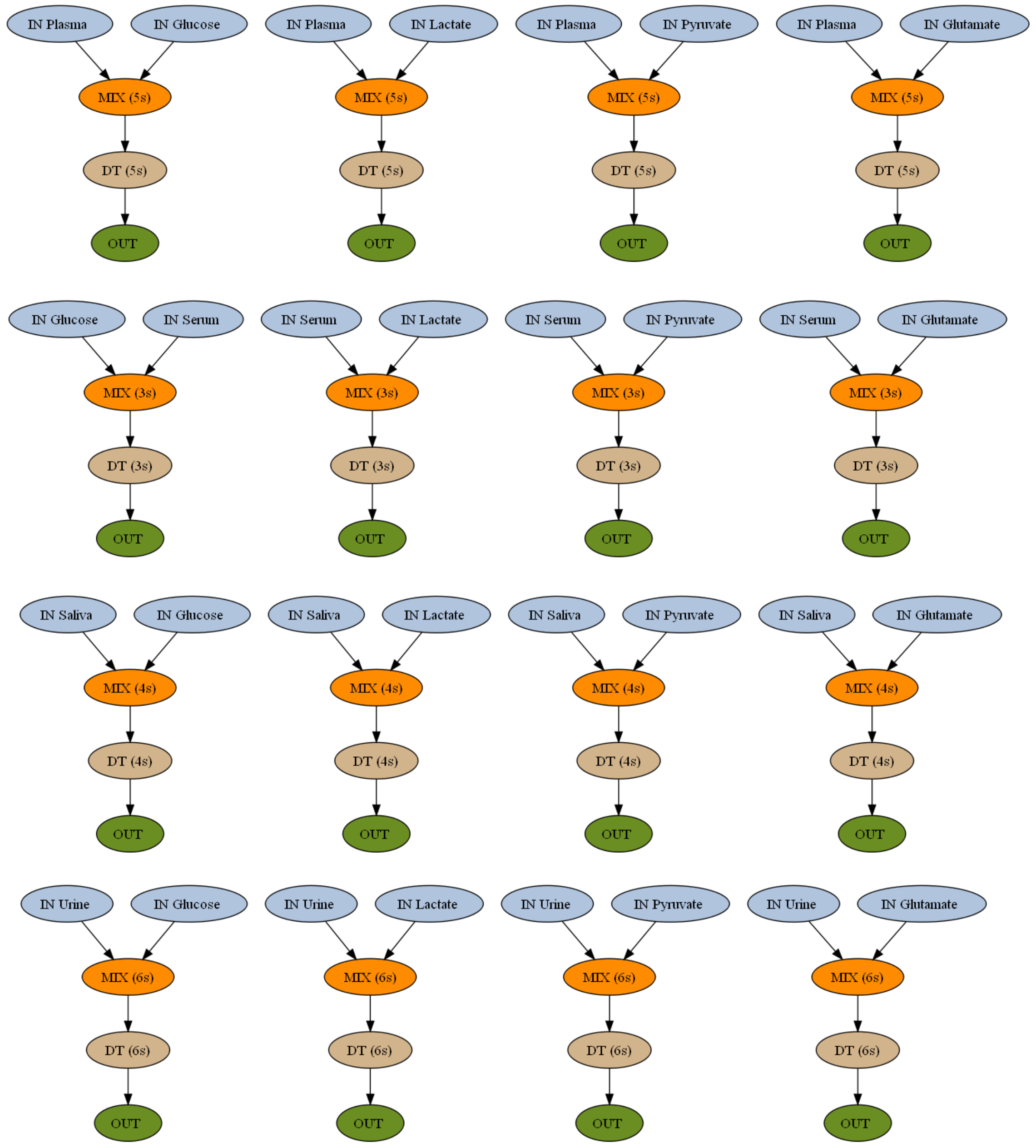


Figure A - 6: ***InVitro5*** **Benchmark; DAG for an in-vitro diagnostics assay with 4 samples and 4 reagents. All input operations (blue) are 2s. All detect operations (tan) and mix operations (orange) are as marked (assuming 4x2 modules for mixes). Output operations (green) are instantaneous.**

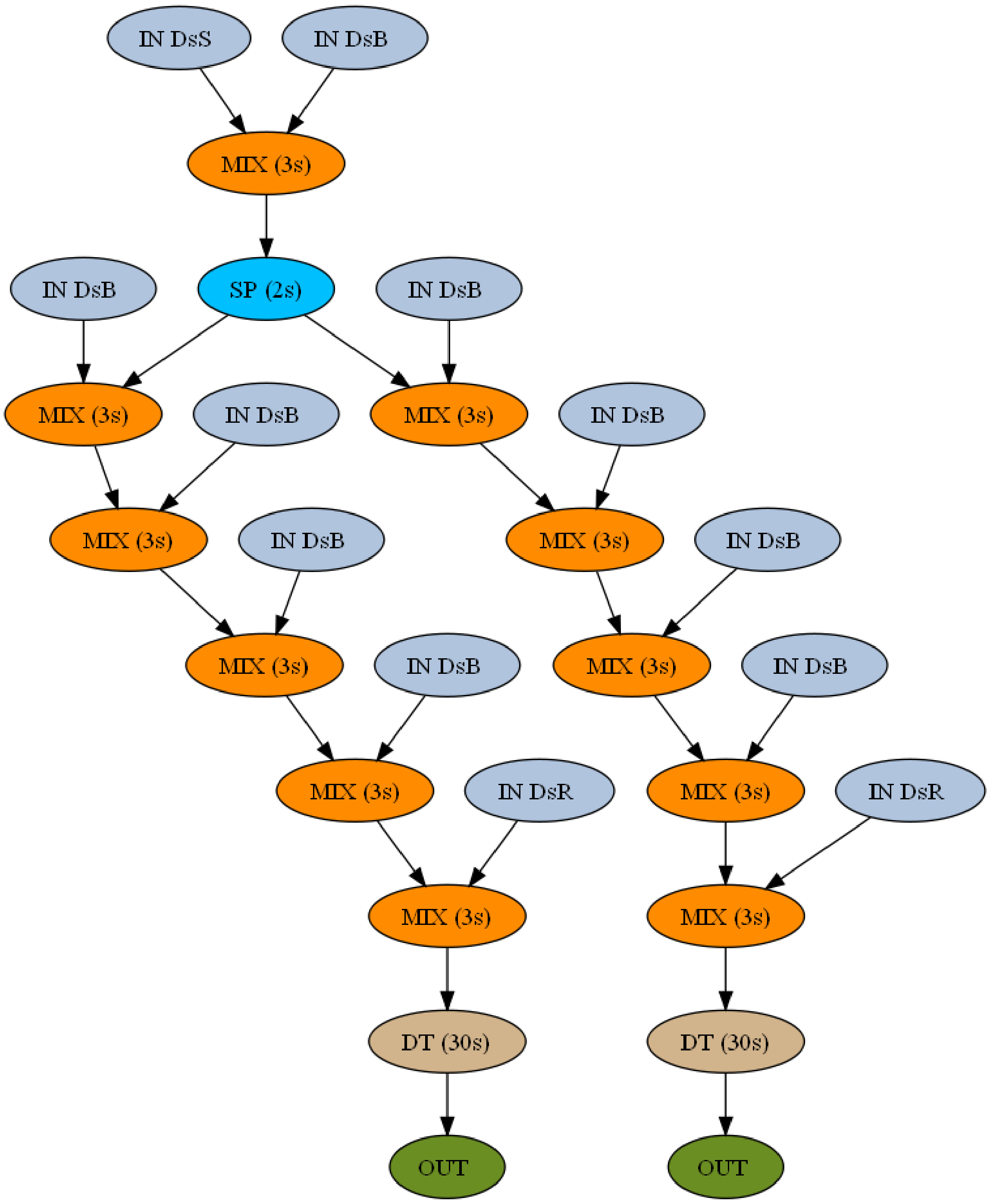


Figure A - 7: *ProteinSplit1* Benchmark; DAG for a protein synthesis assay in which the solution is split 1 time. Dilution operations (5s) are represented by a 2s split (bright blue, SP) operations followed by a 3s mix; all mixes (orange) are 3s (assuming a 4x2 module for mixes) and all input operations are 7s. Detect operations (tan, DT) are 30s and outputs (green) are instantaneous. NOTE: Each of the bottom 5 mixes in each of the 2^1=2 output paths should have a split child which causes one droplet to continue down the path and one to be output to waste so the droplet does not grow too large to be actuated (not shown for simplicity sake).

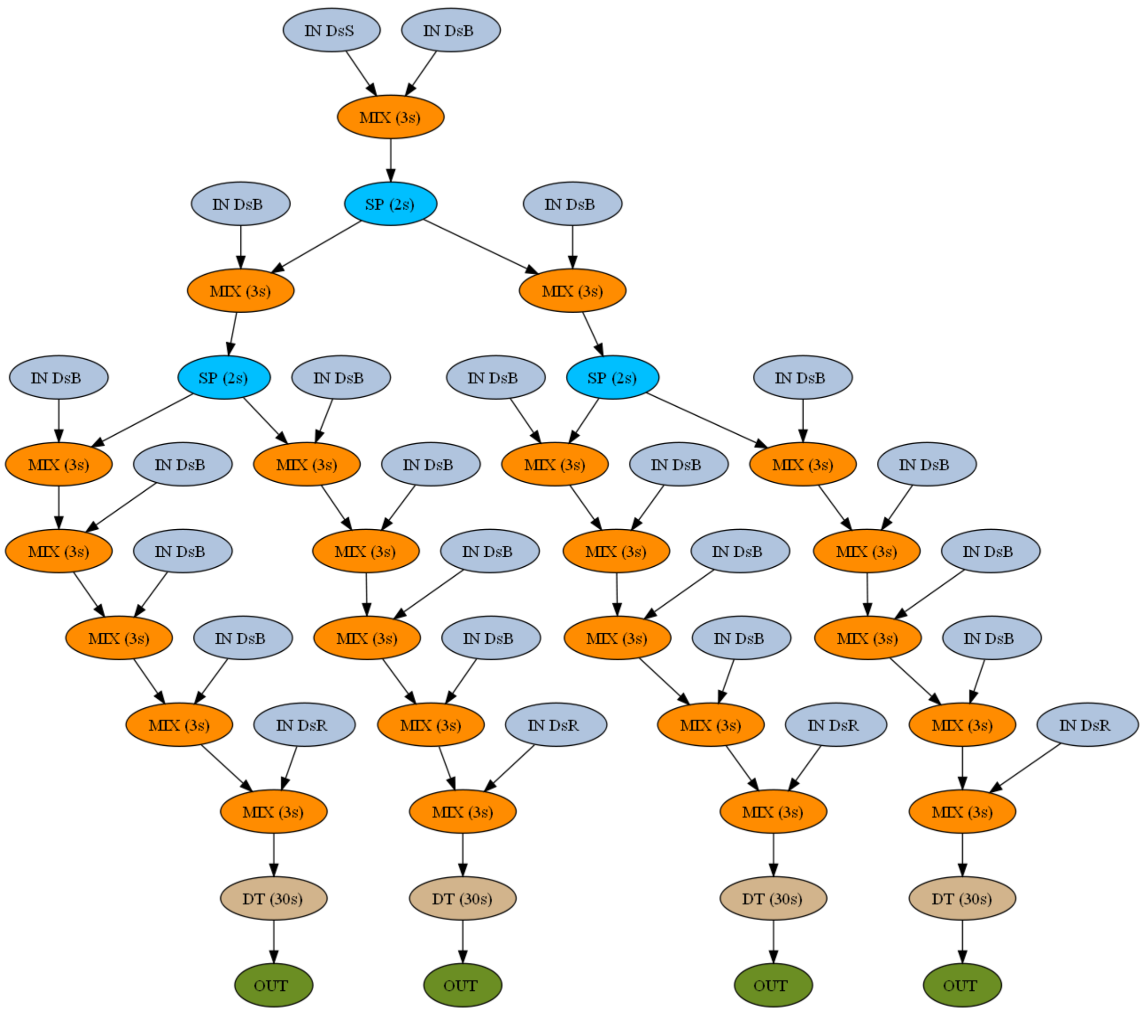


Figure A - 8: *ProteinSplit2* Benchmark; DAG for a protein synthesis assay in which the solution is split 2 times. Dilution operations (5s) are represented by a 2s split (bright blue, SP) operations followed by a 3s mix; all mixes (orange) are 3s (assuming a 4x2 module for mixes) and all input operations are 7s. Detect operations (tan, DT) are 30s and outputs (green) are instantaneous. NOTE: Each of the bottom 5 mixes in each of the 2^1=2 output paths should have a split child which causes one droplet to continue down the path and one to be output to waste so the droplet does not grow too large to be actuated (not shown for simplicity sake).

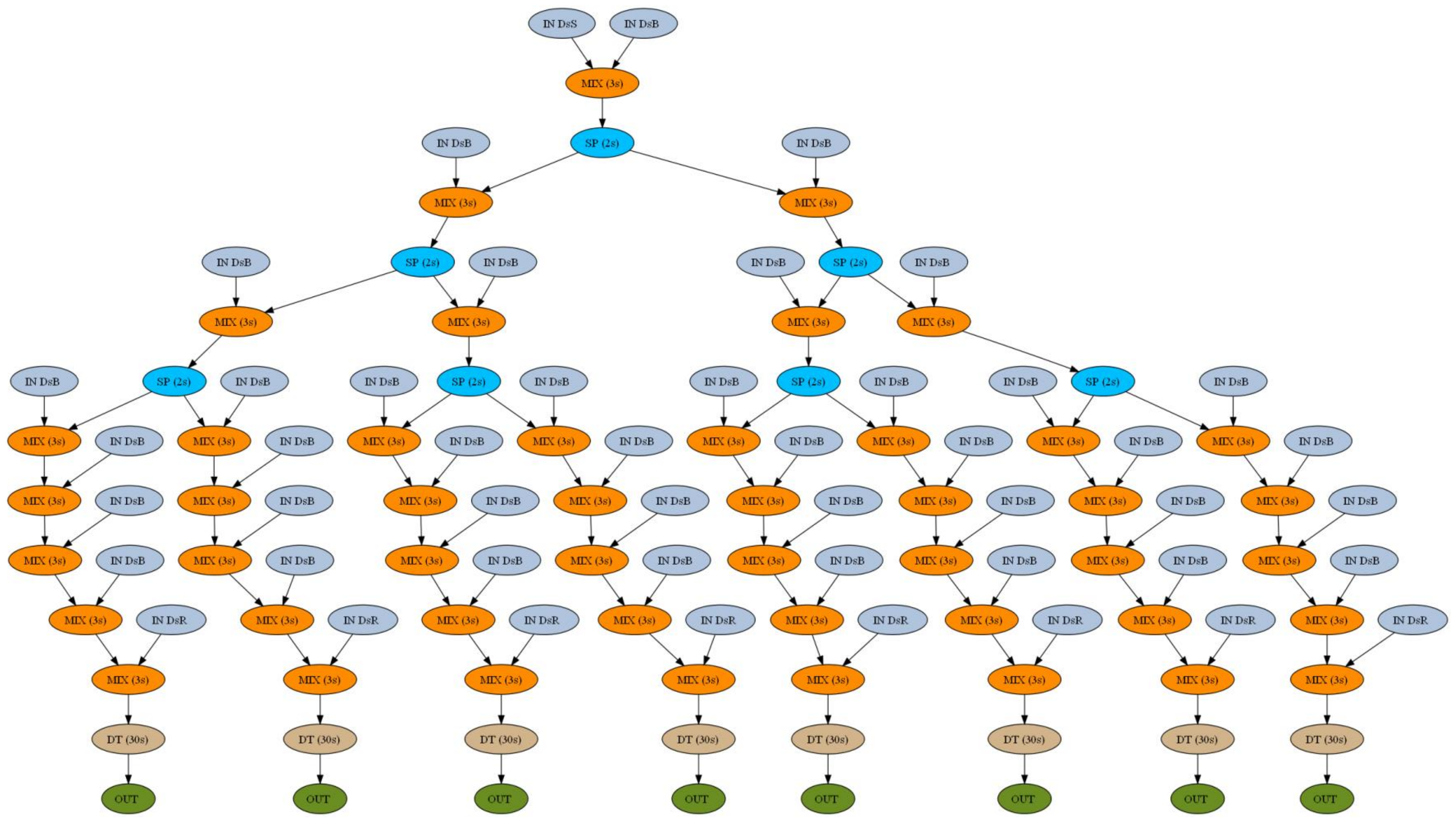


Figure A - 9: ***ProteinSplit3*** **Benchmark; DAG for a protein synthesis assay in which the solution is split 3 times. Dilution operations (5s) are represented by a 2s split (bright blue, SP) operations followed by a 3s mix; all mixes (orange) are 3s (assuming a 4x2 module for mixes) and all input operations are 7s. Detect operations (tan, DT) are 30s and outputs (green) are instantaneous. NOTE: Each of the bottom 5 mixes in each of the 2^1=2 output paths should have a split child which causes one droplet to continue down the path and one to be output to waste so the droplet does not grow too large to be actuated (not shown for simplicity sake).**

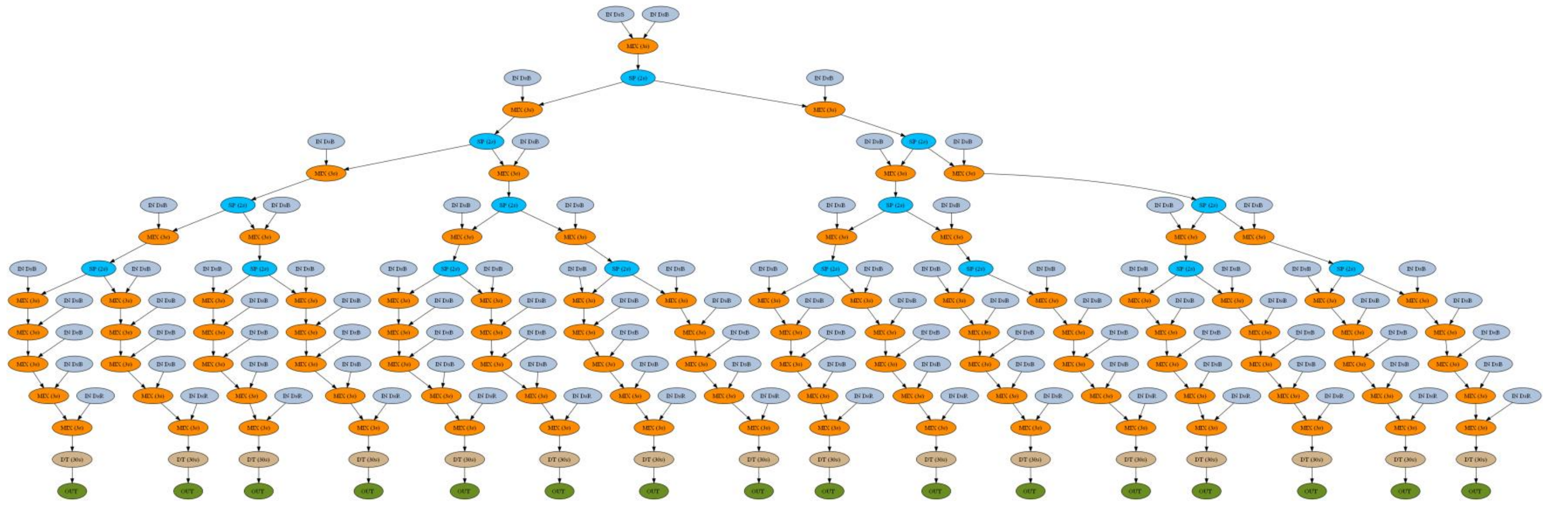


Figure A - 10: *ProteinSplit4* Benchmark; DAG for a protein synthesis assay in which the solution is split 4 times. Dilution operations (5s) are represented by a 2s split (bright blue, SP) operations followed by a 3s mix; all mixes (orange) are 3s (assuming a 4x2 module for mixes) and all input operations are 7s. Detect operations (tan, DT) are 30s and outputs (green) are instantaneous. NOTE: Each of the bottom 5 mixes in each of the 2^1=2 output paths should have a split child which causes one droplet to continue down the path and one to be output to waste so the droplet does not grow too large to be actuated (not shown for simplicity sake).

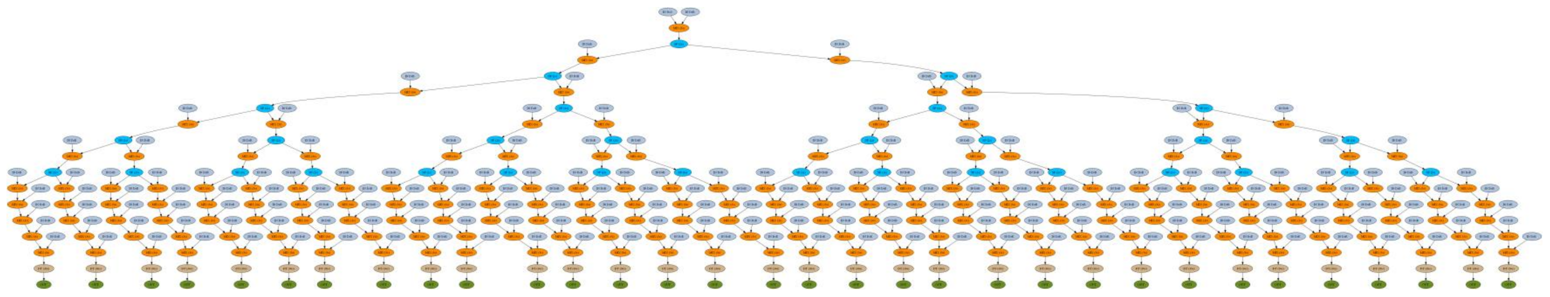


Figure A - 11: ***ProteinSplit5*** **Benchmark; DAG for a protein synthesis assay in which the solution is split 5 times. Dilution operations (5s) are represented by a 2s split (bright blue, SP) operations followed by a 3s mix; all mixes (orange) are 3s (assuming a 4x2 module for mixes) and all input operations are 7s. Detect operations (tan, DT) are 30s and outputs (green) are instantaneous. NOTE: Each of the bottom 5 mixes in each of the 2^1=2 output paths should have a split child which causes one droplet to continue down the path and one to be output to waste so the droplet does not grow too large to be actuated (not shown for simplicity sake).**

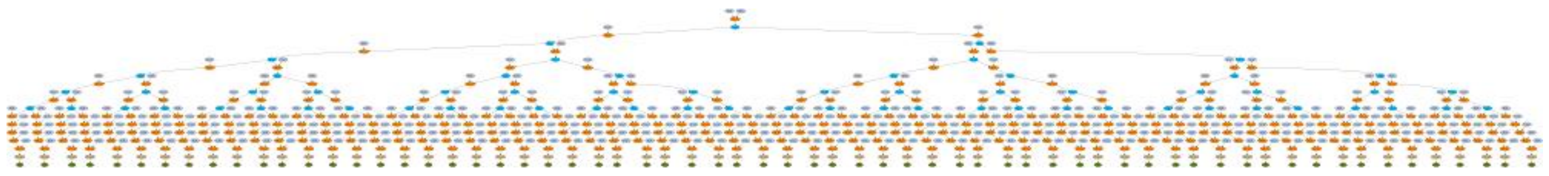


Figure A - 12: ***ProteinSplit6*** **Benchmark; DAG for a protein synthesis assay in which the solution is split 6 times. Dilution operations (5s) are represented by a 2s split (bright blue, SP) operations followed by a 3s mix; all mixes (orange) are 3s (assuming a 4x2 module for mixes) and all input operations are 7s. Detect operations (tan, DT) are 30s and outputs (green) are instantaneous. NOTE: Each of the bottom 5 mixes in each of the 2^1=2 output paths should have a split child which causes one droplet to continue down the path and one to be output to waste so the droplet does not grow too large to be actuated (not shown for simplicity sake).**

Figure A - 13: *ProteinSplit7* Benchmark; DAG for a protein synthesis assay in which the solution is split 7 times. Dilution operations (5s) are represented by a 2s split (bright blue, SP) operations followed by a 3s mix; all mixes (orange) are 3s (assuming a 4x2 module for mixes) and all input operations are 7s. Detect operations (tan, DT) are 30s and outputs (green) are instantaneous. NOTE: Each of the bottom 5 mixes in each of the 2^1=2 output paths should have a split child which causes one droplet to continue down the path and one to be output to waste so the droplet does not grow too large to be actuated (not shown for simplicity sake).