# Interpreting Assays with Control Flow on Digital Microfluidic Biochips

DANIEL GRISSOM, CHRISTOPHER CURTIS, and PHILIP BRISK,
University of California, Riverside

BioCoder is a C++ library developed at Microsoft Research, India, for the unambiguous specification of biochemical assays. This article describes language extensions to BioCoder along with a compiler and runtime system that translate and execute assays specified using BioCoder on a software simulator. The simulator mimics the behavior of laboratories-on-a-chip (LoCs) based on a droplet actuation technology called electrowetting on dielectric (EWoD). To date, prior compilers targeting similar EWoD devices are limited to assays specified as directed acyclic graphs (DAGs) and cannot handle arbitrary control flow or feedback from the LoC. The framework presented herein addresses these challenges through dynamic interpretation, thereby enlarging the space of assays that can be compiled onto EWoD devices.

Categories and Subject Descriptors: B.7.2 [**Integrated Circuits**]: Design Aids; B.8.2 [**Performance and Reliability**]: Performance Analysis and Design Aids; J.3 [**Computer Applications**]: Life and Medical Sciences—*Biology and genetics, Health*

General Terms: Performance, Design

Additional Key Words and Phrases: Digital microfluidic biochip (DMFB), BioCoder

**ACM Reference Format:**
Daniel Grissom, Christopher Curtis, and Philip Brisk. 2014. Interpreting assays with control flow on digital microfluidic biochips. ACM J. Emerg. Technol. Comput. Syst. 10, 3, Article 24 (April 2014), 30 pages.
DOI: http://dx.doi.org/10.1145/2567669

## 1. INTRODUCTION

Microfluidics is an interdisciplinary field of science and engineering that controls and manipulates liquids at the micro- to nano-liter scale. Microfluidics enables the integration of chemical laboratory functions onto *laboratories-on-chip* (LoCs) which have the potential to miniaturize and automate the execution of biochemical assays, which are currently performed by hand using traditional benchtop chemistry equipment and methods. An LoC is a passive device; it requires a computing device (e.g., a microcontroller, processor, FPGA, ASIC, etc.) to execute a program that sends signals to the LoC over time in order to actuate fluidic motion. LoCs may be either assay-specific [Xu et al. 2010] or programmable [Pollack et al. 2002; Amin et al. 2007]; 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, for example, to assays that can be represented as *directed acyclic graphs* (DAGs) without control flow and without the ability to take action based on feedback provided by the device.



This article is an extension of a conference article published in *Proceedings of the Great Lakes Symposium on VLSI (GLSVLSI'12)* [Grissom and Brisk 2012c].
This work is supported by the National Science Foundation under grant CNS-1035603 and an NSF Graduate Research Fellowship awarded to Daniel Grissom.
Authors' addresses: D. Grissom (corresponding author), C. Curtis, and P. Brisk, University of California, Riverside, Department of Computer Science and Engineering, 900 University Ave., Riverside, CA 92521; email: {grissomd, ccurt002, philip}@cs.ucr.edu.

DOI: http://dx.doi.org/10.1145/2567669

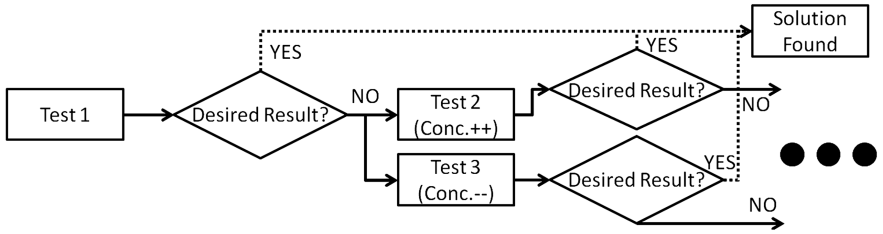



Fig. 1. A control-flow graph for a simple drug-discovery assay that increases (++) or decreases (–) concentrations based on the results of previous tests.

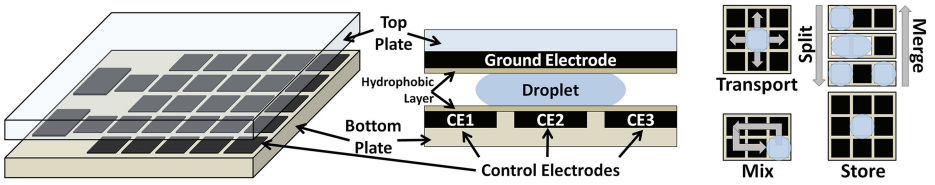



Fig. 2. (a) A DMFB consists of a 2D grid of control electrodes [Grissom and Brisk 2012b]; (b) a cross-sectional view. Applying voltages to the control electrodes in the vicinity of a droplet can hold it in place (CE2), or initiate droplet movement to the left (CE1) or right (CE3), or split the droplet in two (applying voltages to CE1 and CE3 at the same time) [Grissom and Brisk 2012b]. (c) Illustration of fundamental DMFB assay operations being performed on the 2D array: transport, splitting, merging, mixing, and storage.

This article introduces a software interpreter that performs online execution of assays featuring control flow, allowing them to be scheduled and executed immediately based on live feedback for the device. 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 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 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. Instead, an online interpreter could leverage control flow to instantly schedule and dispatch new assays (in Figure 1, the boxes labeled Test 1–3) upon detection of any terminating dependent assays.

### 1.1. Digital Microfluidic Biochip Overview

A *Digital Microfluidic BioChip* (DMFB) is a type of LoC that manipulates discrete droplets of fluid on a two-dimensional array through an enabling technology called *electrowetting on dielectric (EWoD)*. A DMFB is a 2D grid of electrodes, as shown in Figure 2(a) [Pollack et al. 2002]. The term *cell* refers to the space on top of each electrode. The DMFB actuates discrete droplets of fluid whose diameter is slightly larger than the area of each cell, as seen in Figure 2(b). Activating the control electrode beneath a droplet holds it in place. Activating a control electrode in an adjacent cell induces

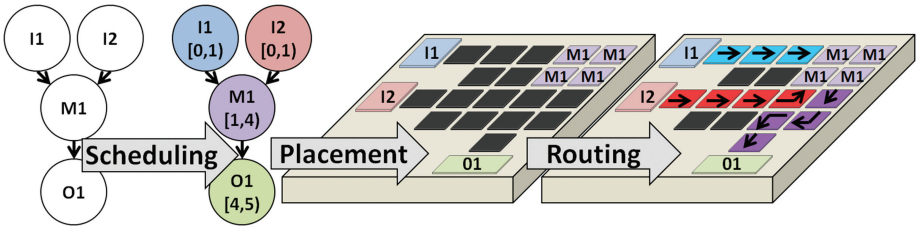



Fig. 3. DMFB compilation involves three primary steps: scheduling assay operations, placing operations at each time-step onto the 2D array, and routing droplets to their destinations [Grissom and Brisk 2012a].

droplet motion. By appropriately activating electrodes in the vicinity of a droplet in sequence, a DMFB can perform basic operations, such as droplet transporting, splitting, merging, mixing, and in-place storage, as shown in Figure 2(c). Additionally, external devices such as sensors or heaters can be affixed to specific DMFB locations; similarly, an external imaging system can be focused on a specific cell or group of cells. To use such an external device, one or more droplets are moved to the specified location and are stored in place; the device itself is activated appropriately under control of a human user or a computational device that controls the entire system.

These basic operations have proven sufficient to perform a wide variety of assays, such as DNA polymerase chain reactions (PCR), in-vitro diagnostics for clinical pathology, immunoassays [Chakrabarty 2010], protein crystallization [Xu et al. 2010], and others. Altogether, assay execution on a DMFB can be viewed as a form of reconfigurable computing, as different cells or groups of cells can be reconfigured to perform different operations throughout the duration of assay execution.

### 1.2. High-Level Synthesis Background

Historically, assays have been specified as DAGs, without control flow. A typical compilation sequence is shown in Figure 3. The DAG is first scheduled [Ding et al. 2001; Ricketts et al. 2006; Su and Chakrabarty 2008; Grissom and Brisk 2012b; O'Neal et al. 2012]; dimensions for each operational module are selected [Su and Chakrabarty 2005; Xu and Chakrabarty 2007]; scheduled operations are then placed onto the 2D grid, ensuring that no concurrently executing operations overlap to prevent interference [Su and Chakrabarty 2006b; Yuh et al. 2007; Liao and Hu 2011]; lastly, non-interfering droplet routes are computed to deliver droplets to the appropriate DMFB locations at appropriate times [Su et al. 2006; Bohringer 2006; Cho and Pan 2008; Yuh et al. 2008; Huang and Ho 2009; L'Orsa et al. 2009; Roy et al. 2010, 2012; Singha et al. 2010], in some cases, all of these problems can be solved in conjunction with one another [Maftei et al. 2010].

The control program that is generated by the compiler is a linear state machine, as shown in Figure 4. Each state specifies a subset of electrodes that will be activated; it typically takes 10ms 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 [Alistar et al. 2010]. An alternative approach, which can accommodate assay operations that fail and require partial recomputation, is to pause assay execution temporarily and recompile the assay on the fly [Zhao et al. 2010; Luo et al. 2012; Alistar et al. 2012]. Although this overall approach results in the execution of a nonlinear state machine, it essentially uses dynamic recompilation to replace one linear state machine with another, as shown in Figure 5.

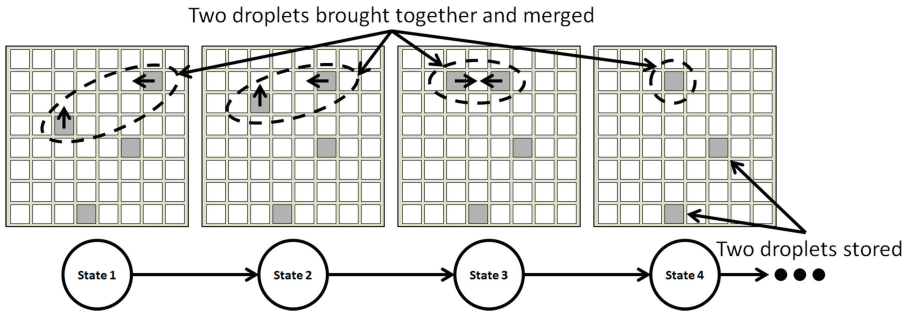



Fig. 4. Example of the linear state machine model of DMFB control. The output of each state is the subset of electrodes in the DMFB that will be activated during each time step (shown in gray). The state machine is timed, based on the activation frequency, typically 100 Hz [Yuh et al. 2008]. In this example, two droplets are transported to a common location so that they can be merged, and two droplets are stored in-place.

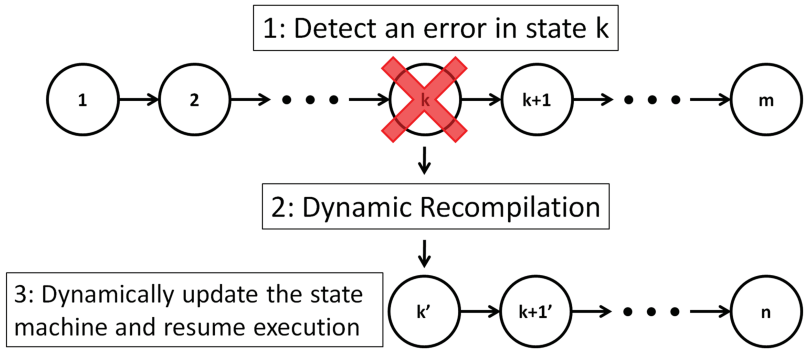



Fig. 5. In response to an error detected in state k, the assay is paused and recompiled, which includes the insertion of new states to recompute fluids that have been lost due to erroneous processing, which may execute concurrently with other ongoing assay operations that were not adversely affected. The output is a new linear state machine that compensates for and corrects the errors that occurred.

### 1.3. 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 [Ananthanarayanan and Thies 2010], 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 and can use feedback from the DMFB to make control-flow decisions, and does not rely on complex resynthesis methods that dynamically recompile the assay when control flow becomes unpredictable [Zhao et al. 2010; Alistar et al. 2010, 2012; Luo et al. 2012]. This represents a significant improvement over existing DMFB compilation techniques, as it enables the execution of a much wider range of assays compared to what has been supported previously.

## 2. VIRTUAL ARCHITECTURE

The system outlined in this article interprets assays dynamically, rather than compiling them statically and then (possibly) recompiling dynamically. Similar to prior work by Griffith et al. [2006], the approach taken here imposes a *virtual architecture* (Figure 6) on the DMFB that restricts the functions that different cells can perform; the

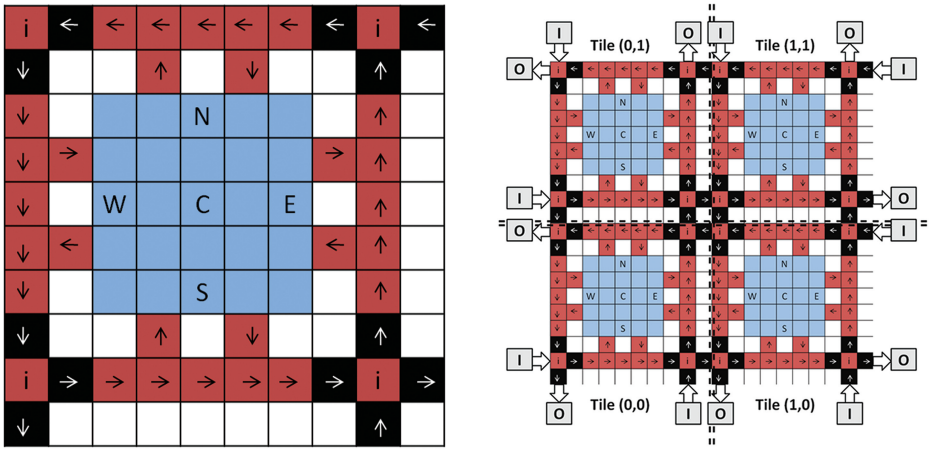



Fig. 6. (a) A tile is the fundamental building block of the virtual architecture; (b) the virtual architecture is imposed onto a DMFB by tiling the fundamental building blocks to create a 2D array of tiles.

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 6(b). The city blocks are referred to as *(work) chambers*, because all non-I/O assay operations occur there. Each chamber 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 chamber. All streets are one-way; eight one-way streets meet together at rotaries, which offer an abstraction like a network router. Droplets travel clockwise through these rotary.

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. Each chamber and the four adjacent streets surrounding it are called a *tile*, as shown in Figure 6(a). If the chamber is $5 \times 5$, then a tile requires a $10 \times 10$ array of cells. Tiles are then repeated in two dimensions to form the virtual architecture. For example, Figure 6(b) shows a virtual architecture that is a $2 \times 2$ array of tiles.

### 2.1. Synthesis Simplifications

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

*Placement.* Rather than placing operations at any location on the DMFB, the interpreter dynamically *binds* operations to chambers, as shown in Figure 7. In principle, any available work chamber can be chosen; when multiple chambers 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 architecture, 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 [Dally and Towles

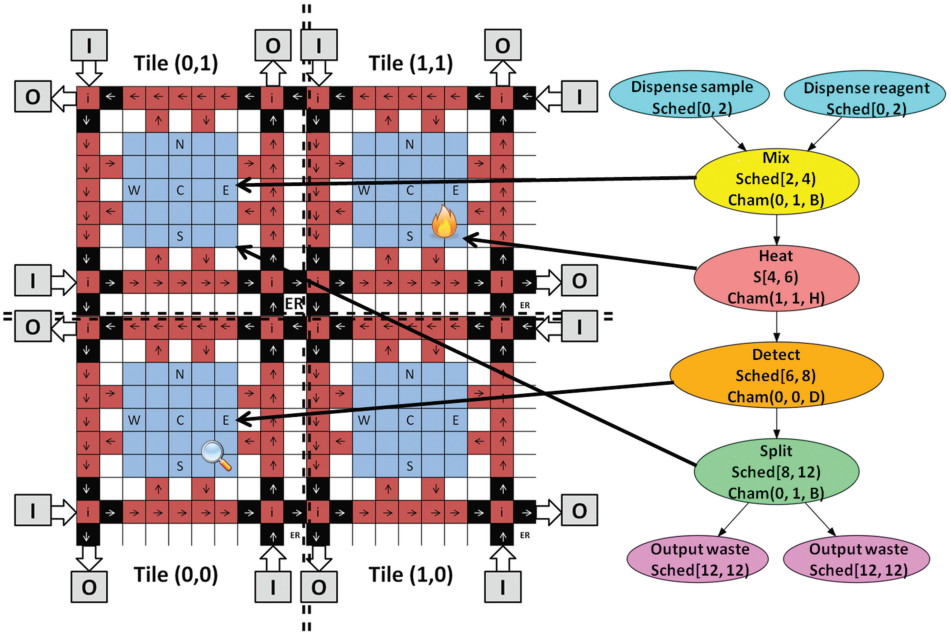



Fig. 7. The operations of a scheduled DAG are bound to the chambers indicated during the specified time steps.

2004] to be compatible with DMFB technology. Thus, droplet routing follows a simple protocol, rather than solving a challenging constrained-optimization problem.

### 2.2. Deadlock-Free 2D-Mesh Routing

The virtual architecture organizes the DMFB as a 2D-mesh network of work chambers, where the rotaries play a role akin to network routers. One contribution of this article is to adapt deadlock-free 2D-mesh routing algorithms [Dally and Towles 2004] to DMFBs using this virtual architecture. Deadlock-free routing is important in an online system where assays are scheduled and executed on the fly, because it is imperative to guarantee that droplets can reach their destination.

To motivate the design for our virtual architecture, we highlight the similarities and differences between our virtual architecture and a 2D-mesh network in the following sections. Later, Section 4.2.3 shows how 2D-mesh routing algorithms are leveraged to achieve deadlock-free routing in our system. Griffith et al. [2006], it should be noted, also achieved deadlock freedom in their virtual architecture but did so by limiting the injection rate of droplets into the system.

*2.2.1. Analogue to 2D-Mesh Topology.* As shown in Figure 6(a), a tile contains a chamber surrounded by one-way streets on each side. The chamber 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 8(a)). The four streets and intersections surrounding the chamber form a counterclockwise traffic circle called the *chamber rotary*. In Figure 8(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.

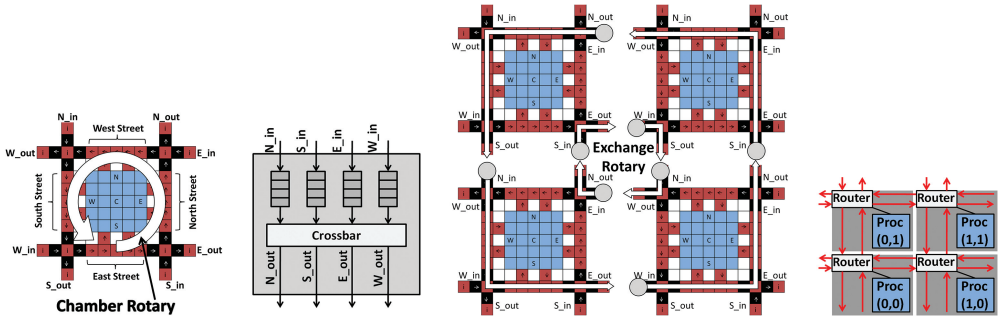



Fig. 8. (a) A chamber rotary (the cycle formed by four streets surrounding a chamber and their intersections) is similar to (b) a 2D-mesh network router; (c) an exchange rotary (the clockwise inner loop) and a long counterclockwise cycle (outer loop) of a tiled virtual architecture form equivalent connections to a (d) 2D-mesh network.

—*Chambers and I/O to Processors.* In a computer network, processors send packets to one another. Likewise, a DMFB can route droplets from one chamber to another, or between chambers and I/O reservoirs.
—*Chamber Rotaries to Routers.* Figure 8(a) depicts a chamber rotary which is essentially 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 8(b).
—*Exchange Rotaries to Wires.* Figure 8(c) depicts an exchange rotary. The cells extending from the chamber rotary (the inputs and outputs in Figure 8(a)) are similar to wires in a 2D-mesh (Figure 8(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 8(d)).
—*Streets to Buffers.* Streets hold droplets, similar to input buffers of routers (see Figures 8(a–b)).

*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 8(d)) employs a crossbar (Figure 8(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 chamber, 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 chamber executes each assay operation) can be made dynamically with minimal overhead.

## 3. BIOCODER LANGUAGE AND EXTENSIONS

BioCoder [Ananthanarayanan and Thies 2010] 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 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.

### 3.1. On the 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 here. 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 [Amin et al. 2007]. 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.

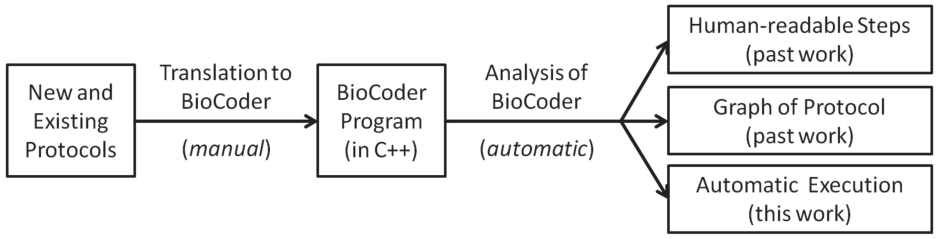
On the one hand, any language or library for specifying biological protocols must evolve as new components are developed for use: new operators (languages) or functions (libraries) to specify the usage of these components must be added; otherwise, the language or library itself will become stale over 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.

### 3.2. Object-Oriented Organization

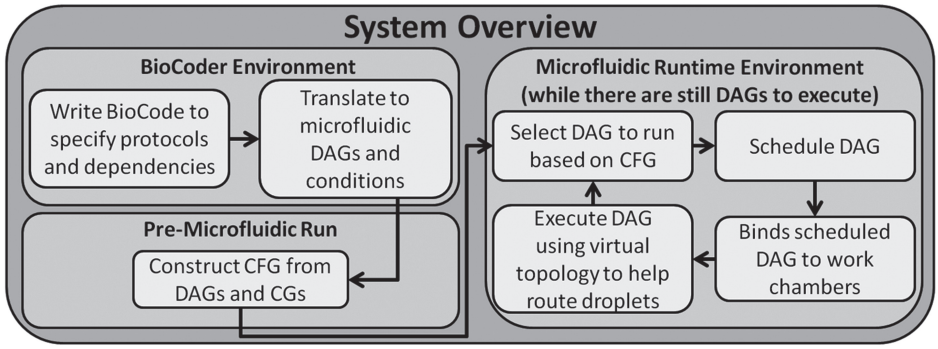
As mentioned earlier, BioCoder is a C++ library containing a variety of *structs*, global variables, and *static* functions. BioCoder's compiler creates an internal data structure that represents an assay as a DAG. The DAG is traversed to convert the assay into English language output. Assay information 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;




(a)




(b)

Fig. 9. (a) Overview of the BioCoder system and output [Ananthanarayanan and Thies 2010, Figure 1]; (b) system overview showing the BioCoder environment, the runtime environment, and the interface between them.

Table I. Prototypes for Six BioCoder Functions that are Supported by EWoD-Based LoCs

| 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) |

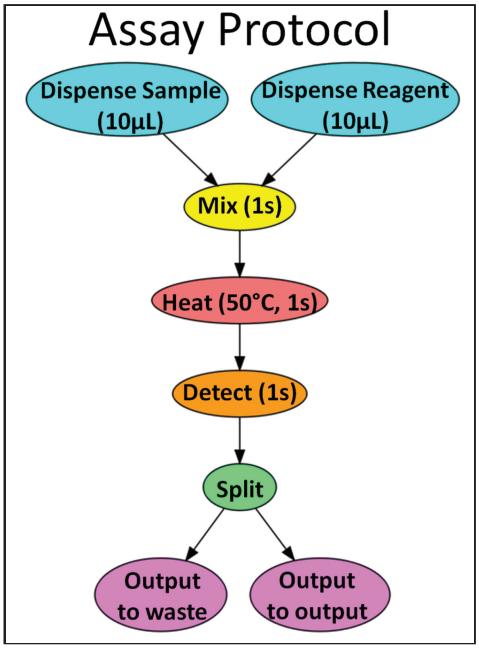
assay operations are specified as method calls. The compiler converts the program into a graph-based intermediate representation using a new class that we introduced called *AssayProtocol*, which effectively represents the CFG. *AssayProtocol*, enables the protocol to be saved, copied, and executed multiple times (if desired).

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 9(a) depicts the old and new capabilities of our system.

Table I 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 I are part of the original BioCoder specification. In traditional benchtop chemistry, the meaning of container is literal—for example, 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.

```
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)

Fig. 10. (a) BioCoder code for a sample assay and (b) the representative DAG structure. This assay does not require control flow.

### 3.3. Example

Figure 10 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 10$\mu$l of a sample and reagent for 1s, heats the mixture for 1s at 50°C, and detects the fluorescence for 1s before splitting and outputting 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, that is, the fluidic analogue of a data dependency.

### 3.4. BioCoder Extensions for Feedback and Control Flow

To support feedback and control-flow constructs, several new 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 11, a *BioConditionalGroup* can be thought of as 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.

Table II 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.

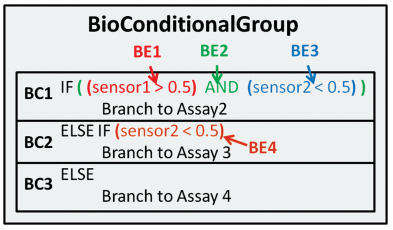



Fig. 11. A *BioConditionalGroup* contains *BioConditions* (BC1-BC3). A *BioCondition* is evaluated by its *BioExpressions* (BE1-BE4).

Table II. *BioExpressions* Create Simple or Complex, Nested Expressions for Branching Functions

| Expr. Type | C++ Style Operator | Eval. 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 III. Microfluidic Operations and BioCoder Functions that Enable Droplet Transfers between Protocols

| New Microfluidic Operation | New BioCoder Function |
|---|---|
| Transfer_In | string reuse_fluid (Container con) |
| Transfer_Out | string save_fluid (Container con) |

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 compare 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.

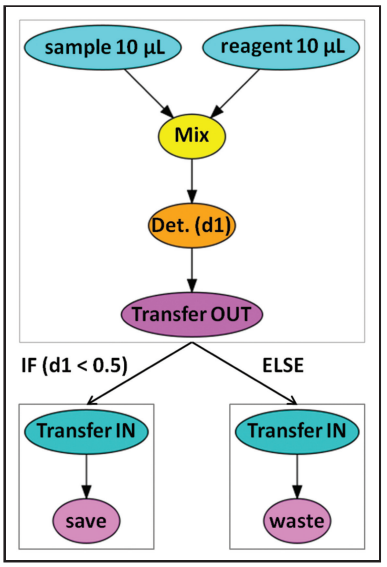
The control-flow mechanism treats each protocol as a DAG and uses control flow to determine which protocols to execute (in sequence) at runtime. In this respect, it is also necessary to transfer droplets from one protocol to the next, depending on which conditions are met. Table III lists new operations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

### 3.5. Example

Figure 12 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,

```
// 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)

Fig. 12. (a) BioCoder code illustrating the use of conditionals; (b) the resultant CFG.

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.

## 4. SYSTEM OVERVIEW AND RUNTIME ENVIRONMENT

Figure 9(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 chamber" areas of the LoC at runtime, and executed dynamically; details are described in the following sections.

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 2(c).

### 4.1. Intermediate Bytecode Format and Interpreter

Conceptually, the set of signals sent to a DMFB during each cycle can be treated like 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 recompilation approaches shown in Figure 3. The virtual architecture 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 (Figures 4 and 5).

The interpreter is a software application that accepts a partially compiled BioCoder assay, essentially 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.

*4.1.1. Bytecode Instruction Format.* The virtual architecture 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, chamber-id)*, where the opcode specifies the operation to perform, and the chamber-id specifies which chamber to perform the operation. All chambers support four basic opcodes: {*start-mix, stop-mix, split, store*} (merging is viewed as a precursor to mixing). If a chamber 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 fairly straightforward restrictions, such as limitations on the number of droplets that a work chamber can store (four), and other than multi-droplet storage, each work chamber 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 chamber stores multiple droplets, only one of which will be transported; or (2) a work chamber 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 chamber, it is stored implicitly until an O-type instruction initiates an operation.

*4.1.2. 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 [Ren et al. 2004]. 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 nonnegligible 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 essentially 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 inputed 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 [Ren et al. 2004, Figure 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.

*4.1.3. 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 chamber; this may occur as a result of a split operation or if the chamber 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.

*4.1.4. 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 chamber W for 20 seconds, the virtual machine would immediately issue the bytecode instruction *(start-mix, W)* and, at the same time, add a command *(stop-mix, W)* to the work queue, with 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 4.3.3 describes the algorithms used to perform online 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.

### 4.2. Interpreting a DAG on the Virtual Architecture

Three steps are required to compile a DAG onto the virtual architecture: scheduling, operation binding, and droplet routing. We describe each of these steps in detail.

*4.2.1. Scheduling.* We use a straightforward list scheduling algorithm for droplet-based LoCs targeting the virtual architecture; this is a simplified version of the modified list scheduling algorithm that does not involve the rescheduling step [Su and Chakrabarty 2008]. 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 architecture. 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 LoC, each chamber can store up to four droplets; however, it cannot perform any mixing operations if it stores a 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 $C$ reaction chambers, $I$ 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:

$$m_t + \left\lceil \frac{s_t}{4} \right\rceil \leq C, i_t \leq I, o_t \leq O, \text{ and } w_t \leq W. \quad (1)$$

Assay operations that require external components, such as sensors or heaters, must be scheduled onto chambers that provide those elements. Without loss of generality,

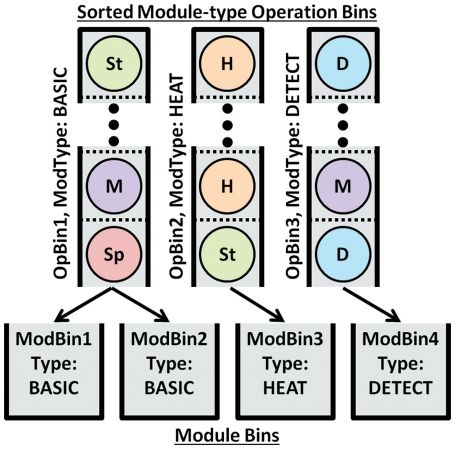



Fig. 13. Left-edge binding solution [Grissom and Brisk 2012a].

the number of heating operations scheduled concurrently cannot exceed the number of chambers 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 chambers.

*4.2.2. 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 chambers, heating chambers, and detecting chambers, and the three I/O resource types are input, output, and waste reservoirs.

The binder selects appropriate resources for each operation that has been scheduled. In general, it is a good strategy to bind operations to resources nearby to the resources that supply the inputs. For example, if we want to mix fluid inputs A and B, it may be a good idea to choose a reaction chamber that is relatively close in proximity to their two input reservoirs. At the same time, since droplet routing does not significantly affect overall assay performance, it is not of particularly great importance to compute a high-quality binding solution.

Our binding algorithm is greedy and efficient and is adapted from the Left Edge Algorithm used for dogleg routing in VLSI [Hashimoto and Stevens 1971] and register allocation in high-level synthesis [Kurdahi and Parker 1987]; Figure 13 shows an example. Operations are first separated into bins based on the chamber type they were assigned during the scheduling phase. The bins are then sorted by their starting time-step. Finally, each resource (e.g., chamber or I/O) selects a bin that matches its resource type and attempts to bind operations to itself, ensuring that no operations overlap, 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.

*4.2.3. Droplet Transportation Protocol (DTP).* DTP chooses a path in the virtual architecture for each droplet and then routes the droplets along their respective paths while satisfying all droplet interference constraints [Su and Chakrabarty 2006b]. The *interference region* of a droplet consists of the cells directly adjacent to a droplet, shown in

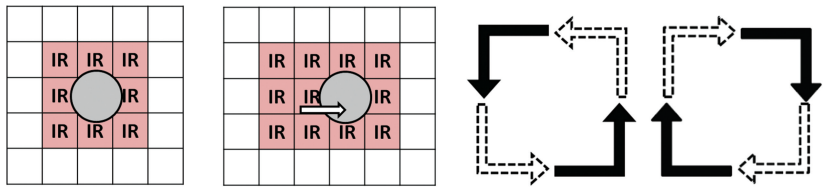



Fig. 14. (a) The interference region ('I') of a droplet at the beginning of a cycle [Grissom and Brisk 2012a]; (b) the interference region of a droplet in motion [Grissom and Brisk 2012a]; (c) legal turns (black) and prohibited turns (dashed outline) in XY routing.

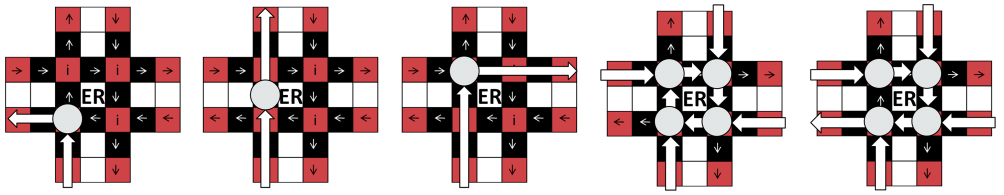
Figure 14(a). If any other droplet enters the interference region, then the two droplets will merge; if the two droplets are not intended to mix, then cross-contamination will inadvertently occur. During droplet motion, the interference region expands to include the union of the interference regions of the source and destination cells, as shown in Figure 14(b).

Deadlock prevention is the foremost responsibility of DTP. In this respect, the rotaries in the virtual architecture take on a role similar in principle to network routers; however, there are several important differences that we mention here. First, 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 architecture 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. Last, certain mechanisms, such as virtual channels [Dally and Seitz 1987], 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.

DTP adapts dimension-ordered routing (e.g., XY or YX routing in 2D) for the DMFB virtual architecture. 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 14(c). Another class of routing algorithms that can be used with the virtual architecture, based on the turn model, prevents 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 [Glass and Ni 1994]. Odd-even (OE) routing is another adaptive algorithm that prevents deadlock by prohibiting some types of turns in certain tile columns [Chiu 2000]. 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 chamber. The four streets and intersections surrounding a chamber form a counterclockwise traffic circle called a *chamber* rotary, shown in Figure 8(a). The term *exchange*




(a) (b) (c) (d) (e)

Fig. 15. (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.

*rotary* is introduced to refer to the original rotaries which allow droplets to move from a tile to one of its neighbors. Figure 8(c) shows that droplets travel clockwise through an exchange rotary and counterclockwise through one or more chamber rotaries. Groups of droplets traveling along XY paths can form cycles in chamber and exchange rotaries, and therefore preventing the formation of these cycles prevents deadlock.

Four specific rules are required.

—*Chamber Entries and Exits.* Droplets may not make prohibited turns (Figure 14(c)) when leaving source and entering destination chambers. To ensure routability in light of prohibited turns, entries and exits are placed on all four sides of the chamber.
—*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.
—*Exchange Rotaries.* In Figure 15(a), a droplet *clips* an exchange rotary if it touches one intersection before leaving. In Figures 15(b) and 15(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 15(d) depicts exchange rotary deadlock when four droplets attempt to pass through; no droplet can progress without maintaining spacing constraints (Figures 14(a–b)). In Figure 15(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.
—*Chamber Rotaries.* Figure 16(a) illustrates chamber rotary deadlock. Droplet 16 creates a dependency chain which causes deadlock; however, if it does not enter the chamber rotary, then a bubble is created which ensures that the sequence of droplets can proceed, starting with Droplet 1. To prevent deadlock in a chamber 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 chamber or input reservoir must also wait until that street has room; in Figure 16(b), Droplets 1, 2, 3, 5, and 6 must wait for this reason.

### 4.3. 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

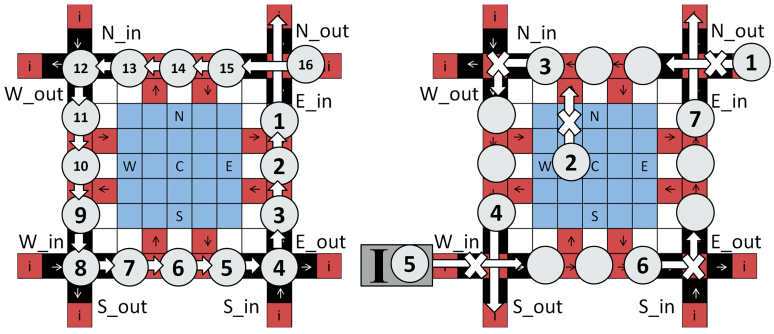



Fig. 16. (a) Deadlock in a chamber rotary; (b) chamber rotary with street capacity rules being enforced to prevent deadlock.

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 9(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.

## 5. 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 by researchers at Duke University [Su and Chakrabarty 2006a].

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 [Fair et al. 2007] 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™ E638 processor with 512MB RAM, running TimeSys 11 Linux; memory constraints were not a concern using this more powerful platform.

### 5.1. Experiment #1: Fault-Tolerant Splitting

A common assumption made when compiling assays onto EWoD-based DMFBs is that every split operation is perfect, that is, 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 remerged and the split operation repeats [Alistar et al. 2012]. This process, which is naturally probabilistic, can be expressed using a loop using the extensions to BioCoder.

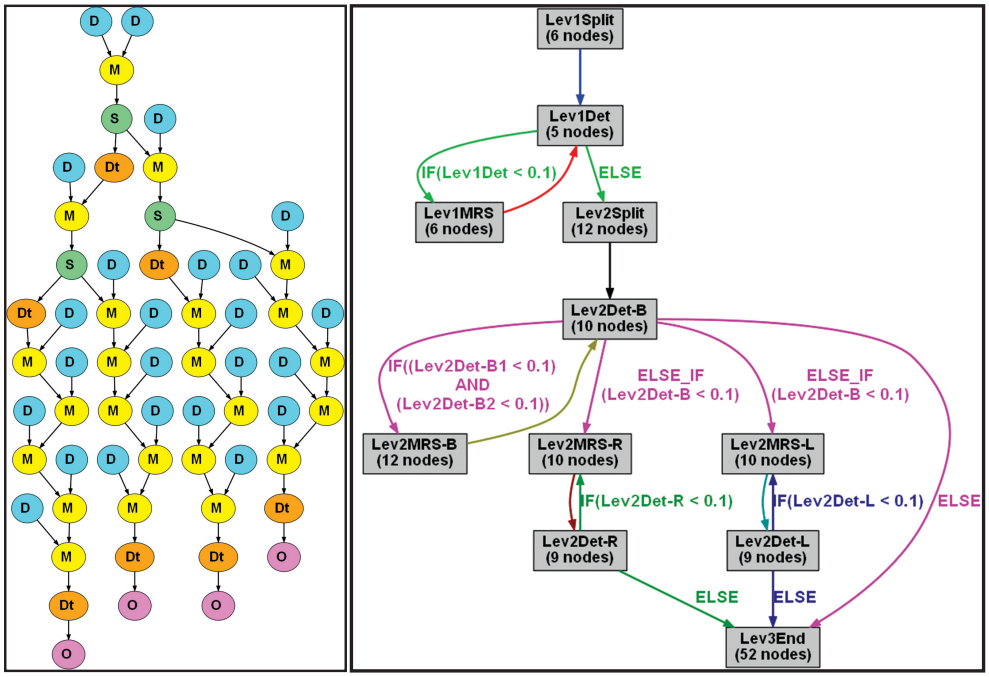



Fig. 17. Output from system showing (a) the initial two-level protein DAG executed by the online resynthesis model and (b) the control-flow graph executed by our interpreter.

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 [2010]. 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 resynthesis (e.g., Figure 5) in response to an erroneous split: when a split-error occurs, the resultant droplets are discarded and a recovery subgraph is called upon to reproduce the droplet to be split again [Alistar et al. 2012]. This eliminates the primary limitation of their former approach—fixing the number of times a split may be performed—but the resynthesis process is more complex as it reexecutes lengthy operations that may not need to be performed again.

To evaluate our interpreter and virtual architecture (INT), we compare against Alistar et al.'s online resynthesis (ORS) idea in which time-redundant graphs are resynthesized 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. First, it appears that their placer does not address the challenge of specialized modules with external devices, for example, detection operations must be placed on top of cells above or below an integrated sensor. Second, 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 [Grissom and Brisk 2012a] that accounts for specialized modules during placement and includes a routing algorithm.

We used a truncated two-level version (for the sake of clarity and demonstration) of a larger three-level protein dilution assay [Su and Chakrabarty 2006a], as shown in Figure 17(a). We do not have a dilute operation, and thus, we replaced each 5s dilute

Table IV. Recovery Synthesis Time (Averaged over 10 Runs) for a 2-Level Protein Assay with Varying Percentages of Error for Split Operations

| Average Recovery Synthesis Time With Fault-Free Baseline | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Synthesis Method | Fault-Free Baseline | 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 |

*Note*: 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 resynthesis (ORS) model and our interpreter (INT) with control-flow and virtual architecture.

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 $20 \times 20$ DMFB, allowing for four total work chambers with four detectors; ORS uses a $15 \times 19$ DMFB, allowing for six total work chambers/modules and has four detectors as well.

Figure 17(a) shows the initial sequencing graph executed by ORS, while Figure 17(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 17(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 two splits). Since there are two splits, there are detect and merge-resplit (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.

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-resplit. We performed ten runs for each of the five 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 IV 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 V shows the assay runtime (operation + routing time) for the same error probabilities as seen in Table IV. Each table has a fault-free baseline, meaning these numbers show synthesis and assay runtimes, respectively, for the two-level protein application when no errors occur. For ORS, this is essentially the DAG seen in Figure 17(a). For INT, this represents the execution path seen in Figure 17(b) of *Lev1Split* $\rightarrow$ *Lev1Det* $\rightarrow$ *Lev2Split* $\rightarrow$ *Lev2Det-B* $\rightarrow$ *Lev3End*, because this is the path that executes when no errors occur.

As shown in Table IV, 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 IV also shows that INT spends less time, on average, in response to each error

Table V. 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

| 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 |

*Note*: Results are shown for the online resynthesis (ORS) model and our interpreter (INT) with control-flow and virtual architecture.

(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 in every single run and causes the total synthesis time to be less than the synthesis-per-error time.

Table V shows the baseline assay execution time for a fault-free run and the average additional overhead incurred to reexecute 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 resplit instead of executing more complicated recovery operations [Alistar et al. 2012]. Another concern, which we did not explicitly model, is that ORS may recursively encounter further errors when reexecuting 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.

### 5.2. 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 [Su and Chakrabarty 2008].

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 18(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 architecture to preserve reagents. Figure 18(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 19 shows the BioCoder specification. The four assays are executed sequentially, and the protocol stops as soon as the first positive reading occurs.

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 VI shows the average results (over ten 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 20 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.

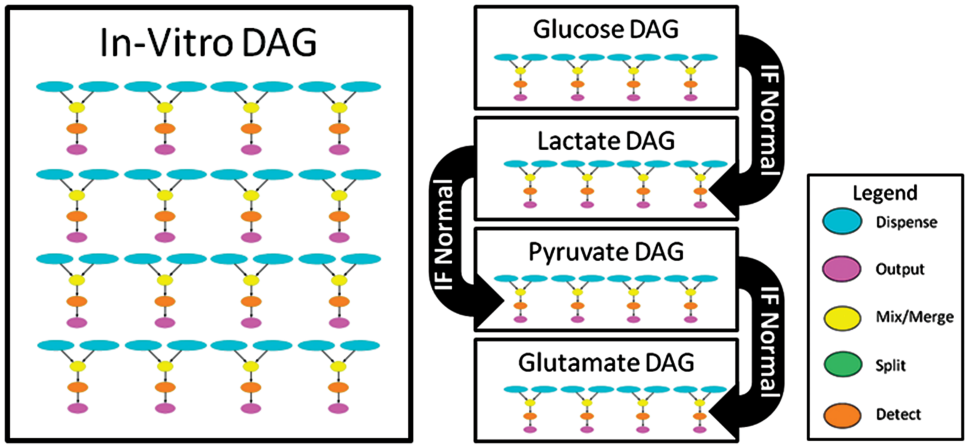



Fig. 18. (a) Parallel and (b) sequential CFG implementations of the in-vitro diagnostics assay.

```
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]);
}
```

Fig. 19. BioCoder specification of the sequential in-vitro assay for a 99% threshold.

Table VI. Average Sequential InVitro Completion Time and Sample Usage Compared to the Parallel InVitro Implementation

| 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 |

*Note*: Averages were found over 10 runs for each health/pass rate.

```
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
D=32, M=16, S=0, H=0, d=16, O=16
```

Fig. 20. Simulator output for a particular run of the sequential in-vitro assay (99% threshold).

### 5.3. Experiment #3: Baseline Assays

Finally, we perform a standard set of benchmark assays commonly reported in literature. The first assay, a polymerase chain reaction (PCR), is a technique for exponential DNA amplification used 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 [Ricketts et al. 2006; Su and Chakrabarty 2008]. Finally, we run a colorimetric protein assay based on the Bradford reaction [Su and Chakrabarty 2008]. DAGs for the PCR and protein assays, as well as the 5th in-vitro assay (four samples, four reagents), are seen in Figure 21. Each of the seven assays represents a DAG with no control flow.

All assays have dispense times of 2s. We used the module libraries provided by Duke University to choose operation timings [Su and Chakrabarty 2006a]. We chose the $2 \times 4$ mixer (3s) for all PCR mixes. For the protein assay, we chose the $2 \times 4$ mixer (3s) and $2 \times 4$ diluter (5s) for all mix and dilution operations, respectively; because we did not have an explicit dilution operation implemented, we divided the dilution operations

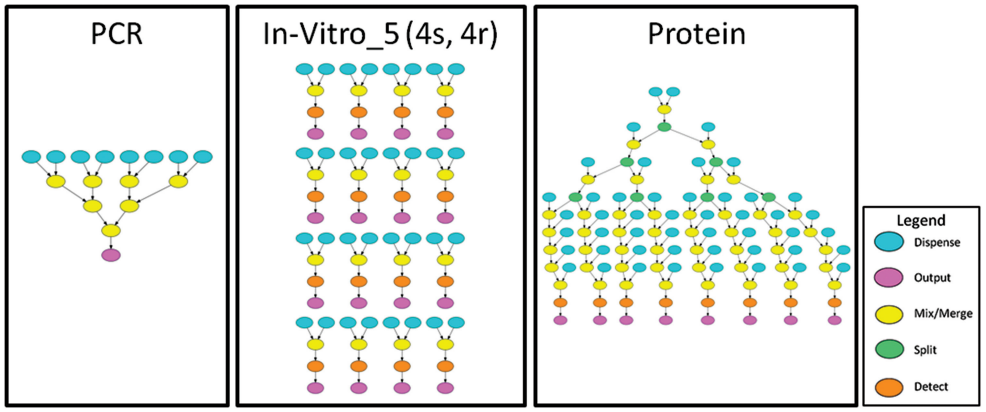



Fig. 21. PCR, In-Vitro, and Protein DAG specifications.

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 Su and Chakrabarty [2008].

Here, we compare the performance of the interpreter and virtual architecture (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 sections. We compare the schedule and route quality between LRSC and INT while simultaneously showing the trade-off that is made between computation time and assay time (schedule and route length). INT was run on a 20 $\times$ 20 DMFB (2 $\times$ 2 tile array); LRSC was able to fit the assays onto a smaller 15 $\times$ 19 DMFB. As in other works, we assume a droplet actuation frequency of 100 Hz [Yuh et al. 2008].

For our LRSC, we chose a genetic scheduling algorithm [Su and Chakrabarty 2008], a simulated annealing-based placer [Su and Chakrabarty, 2006], and a fast maze router [Roy et al. 2010] 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) [Su and Chakrabarty 2008] is also possible but may run for days or weeks on DAGs of nontrivial 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 [Su and Chakrabarty 2008]. We chose to implement a maze router [Roy et al. 2010; Grissom and Brisk 2012b] 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.

Tables VII and VIII 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

Table VII. Static Compiler Synthesis Results for 7 Deterministic Benchmarks Showing Algorithmic Computation Times (CT), the Computed Schedule Lengths (SL), and Computed Route Lengths (RL)

| 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 | 11 | 0.200 | 0.002 | 0.780 | 2.823 | 11.780 | 14.603 |
| InVitro_1 | 4.475 | 14 | 12.843 | 0.002 | 1.350 | 17.320 | 15.350 | 32.670 |
| InVitro_2 | 8.122 | 16 | 141.177 | 0.004 | 1.800 | 149.303 | 17.800 | 167.103 |
| InVitro_3 | 13.156 | 18 | 506.767 | 0.010 | 2.070 | 519.933 | 20.070 | 540.003 |
| InVitro_4 | 22.376 | 22 | 3,317.571 | 0.007 | 2.340 | 3,339.954 | 24.340 | 3,364.294 |
| InVitro_5 | 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 VIII. Online Interpreter (Using the Virtual Architecture) Synthesis Results for 7 Deterministic Benchmarks Showing Algorithmic Computation Times (CT), Computed Schedule Lengths (SL), and Computed Route Lengths (RL)

| Online Interpreter (INT): List Scheduler→Chamber 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 | 11 | 0.001 | 0.009 | 0.560 | 0.011 | 11.560 | 11.571 |
| InVitro_1 | 0.001 | 18 | 0.002 | 0.015 | 1.330 | 0.018 | 19.330 | 19.348 |
| InVitro_2 | 0.002 | 19 | 0.003 | 0.022 | 1.860 | 0.027 | 20.860 | 20.887 |
| InVitro_3 | 0.005 | 29 | 0.006 | 0.034 | 2.540 | 0.045 | 31.540 | 31.585 |
| InVitro_4 | 0.008 | 34 | 0.009 | 0.045 | 3.330 | 0.062 | 37.330 | 37.392 |
| InVitro_5 | 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 IX. Scheduling Results for Various Scheduling Methods for Large ProteinSplit Assays on a 20 × 20 DMFB with Four Work Chambers, Each Equipped with a Detector

| 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** |

*Note*: Scheduling computation times (CT) and the computed schedule lengths (SL) are shown. The best overall scheduler for each benchmark is emboldened.

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.

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 [Grissom and Brisk 2012b] and force-directed list scheduling [O'Neal et al. 2012]. 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 IX shows the results of genetic, list, force-directed list, and path schedulers on four ProteinSplit 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.

In this particular case, however, the inferior schedule quality of the interpreter is mostly due to a lack of resources. On a 20 $\times$ 20 DMFB, the interpreter can perform four concurrent mix operations, one in each of its four chambers. The static compiler, on the other hand, has sufficient room to comfortably fit six 2 $\times$ 4 mixers on a smaller 15 $\times$ 19 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 30 $\times$ 20 DMFB that could fit six mixers and reran 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 [Noh et al. 2011; Hadwen et al. 2012], 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 trade-off to make.

## 6. CONCLUSION, LIMITATIONS, AND FUTURE WORK

This article 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 prior state of the art where assays were compiled statically. The key innovation that facilitates interpretation is the imposition of a virtual architecture 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 chambers 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 recompilation methods.

The drawback of this approach is area overhead: the streets, rotaries, and separation between them occupy a significant number of cells. It is clearly possible to pack the work chambers 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. Our approach is compatible with direct addressing (which uses one control pin per electrode), cross referencing [Fan et al. 2003], and active matrix addressing [Noh et al. 2011; Hadwen et al. 2012] DMFBs; the latter two devices use M + N control pins to control M$\times$N electrodes. Our approach, however, is not compatible with pin-constrained DMFBs, which are designed to be assay-specific and allow one control pin to control multiple electrodes [Xu et al. 2007]. Pin-constrained DMFBs often use fewer than M + N 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 6 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.

Future work will extend the interpreter, including (1) the integration of wash droplet routing [Zhao and Chakrabarty 2009, 2010; Mitra et al. 2011; Huang et al. 2010], which prevents cross-contamination, into the DTP; (2) techniques to maintain the connectivity requirements of the virtual architecture in the presence of faulty cells; and (3) techniques to ensure that the DTP provides an acceptable quality-of-service and can make real-time guarantees for critical droplets traveling through the DMFB. Finally, we hope to develop more assays that can exploit the control-flow-related extensions to BioCoder that were introduced here.

## REFERENCES


M. Alistar, E. Maftei, P. Pop, and J. Madsen. 2010. 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*.

M. Alistar, P. Pop, and J. Madsen. 2012. 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*.

A. M. Amin, M. Thottedthodi, T. N. Vijaykumar, S. Wereley, and S. C. Jacobson. 2007. Aquacore: A programmable architecture for microfluidics. In *Proceedings of the International Symposium on Computer Architecture*. 254–265.

V. Ananthanarayanan and W. Thies. 2010. BioCoder: A programming language for standardizing and automating biology protocols. *J. Biol. Eng.* 4.

K. F. Bohringer. 2006. Modeling and controlling parallel tasks in droplet-based microfluidic systems. *IEEE Trans. Comput. Aid. Des. Integr. Circuits Syst.* 25, 2, 329–339.

K. Chakrabarty. 2010. Design automation and test solutions for digital microfluidic biochips. *IEEE Trans. Circuits Syst.* 57, 1, 4–17.

G.-M. Chiu. 2000. The odd-even turn model for adaptive routing. *IEEE Trans. Parallel Distrib. Syst.* 11, 7, 729–738.

M. Cho and D. Z. Pan. 2008. A high-performance droplet router algorithm for digital microfluidic biochips. *IEEE Trans. Comput. Aid. Des. Integr. Circuits Syst.* 27, 10, 1714–1724.

W. J. Dally and C. L. Seitz. 1987. Deadlock-free message routing in multiprocessor interconnection networks. *IEEE Trans. Comput.* C-36, 5, 547–553.

W. J. Dally and B. P. Towles. 2004. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann.

J. Ding, K. Chakrabarty, and R. B. Fair. 2001. Scheduling of microfluidic operations for reconfigurable two-dimensional electrowetting arrays. *IEEE Trans. Comput. Aid. Des. Integr. Circuits Syst.* 20, 12, 1463–1468.

R. B. Fair, A. Khlystov, T. D. Tailor, V. Ivanov, R. D. Evans, P. B. Griffin, V. Srinivasan, V. K. Pamula, M. G. Pollack, and J. Zhou. 2007. Chemical and biological applications of digital-microfluidic devices. *IEEE Des. Test Comput.* 24, 1, 10–24.

S. K. Fan, C. Hashi, and C. J. Kim. 2003. 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.* 694–697.

C. J. Glass and L. M. Ni. 1994. The turn model for adaptive routing. *J. ACM* 1, 874–902.

E. J. Griffith, S. Akella, and M. K. Goldberg. 2006. Performance characterization of a reconfigurable planar-array digital microfluidic systems. *IEEE Trans. Comput. Aid. Des. Integr. Circuits Syst.* 25, 2, 340–352.

D. Grissom and P. Brisk. 2012a. Fast online synthesis of generally programmable digital microfluidic biochips. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis.* 413–422.

D. Grissom and P. Brisk. 2012b. Path scheduling on digital microfluidic biochips. In *Proceedings of the Design Automation Conference.* 26–35.

D. Grissom and P. Brisk. 2012c. A high-performance online assay interpreter for digital microfluidic biochips. In *Proceedings of the Grest Lakes Symposium on VLSI (GLSVLSI'12).* 103–106.

B. Hadwen, G. R. Broder, D. Morganti, A. Jacobs, C. Brown, J. R. Hector, Y. Kubota, and H. Morgan. 2012. Programmable large area digital microfluidic array with integrated droplet sensing for bioassays. *Lab Chip* 12, 18, 3305–3318.

A. Hashimoto and J. Stevens. 1971. Wire routing by optimizing channel assignment within large apertures. In *Proceedings of the 8th Workshop on Design Automation*. 155–169.

T.-W. Huang and T.-Y. Ho. 2009. A fast routability- and performance-driven droplet routing algorithm for digital microfluidic biochips. In *Proceedings of the International Conference on Computer-Aided Design*. 445–450.

T.-W. Huang, C.-H. Lin, and T.-Y. Ho. 2010. A contamination aware droplet routing algorithm for the synthesis of digital microfluidic biochips. *IEEE Trans. Comput. Aid. Des. Integr. Circuits Syst.* 29, 11, 1682–1695.

F. J. Kurdahi and A. C. Parker. 1987. REAL: A program for REgister ALlocation. In *Proceedings of the Design Automation Conference.* 210–215.

C. Liao and S. Hu. 2011. Multiscale variation-aware techniques for high-performance digital microfluidic lab-on-a-chip component placement. *IEEE Nano Biosci.* 10, 1, 51–58.

R. L'Orsa, B. Bhattacharjee, M. Hoorfar, J. F. Holzman, and N. Homayoun. 2009. Detailed droplet routing and complexity characterization on a digital microfluidic biochip. *Proc. SPIE 7318*, 1–8.

Y. Luo, K. Chakrabarty, and T.-Y. Ho. 2012. A cyberphysical synthesis approach for error recovery in digital microfluidic biochips. In *Proceedings of Design Automation and Test in Europe.* 1239–1244.

E. Maftei, P. Pop, and J. Madsen. 2010. Tabu search-based synthesis of digital microfluidic biochips with dynamically reconfigurable non-rectangular devices. *ACM Trans. Des. Auton. Embed. Syst.* 14, 3, 287–307.

D. Mitra, S. Ghoshal, H. Rahaman, K. Chakrabarty, and B. B. Bhattacharya. 2011. On residue removal in digital microfluidic biochips. In *Proceedings of the Great Lakes Symposium on VLSI*. 391–394.

J. H. Noh, J. Noh, E. Kreit, J. Heikenfeld, and P. D. Rack. 2011. Toward active-matrix lab-on-a-chip: Programmable electrofluidic control enabled by arrayed oxide thin film transistors. *Lab Chip* 12, 2, 353–360.

K. O'neal, D. Grissom, and P. Brisk. 2012. Force-directed scheduling for digital microfluidic biochips. In *Proceedings of the IFIP/IEEE International Conference on Very Large Scale Integration*.

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

H. Ren, R. B. Fair, and M. G. Pollack. 2004. Automated on-chip droplet dispensing with volume control by electro-wetting actuation and capacitance metering. *Sens. Actuators B: Chemical* 98, 2–3, 319–327.

A. J. Ricketts, K. Irick, N. Vijaykrishnan, and M. J. Irwin. 2006. Priority scheduling in digital microfluidics-based biochips. In *Proceedings of Design Automation and Test in Europe*. 1–6.

P. Roy, H. Rahaman, and P. Dasgupta. 2010. A novel droplet routing algorithm for digital microfluidic biochips. In *Proceedings of the Great Lakes Symposium on VLSI*, 2010, 441–446.

P. Roy, H. Rahamn, and P. Dasgupta. 2012. Two-level clustering-based techniques for intelligent droplet routing in digital microfluidic biochips. *Integ. VLSI J.* 45, 3, 316–330.

K. Singha, T. Samanta, H. Rahaman, and P. Dasgupta. 2010. Method of droplet routing in digital microfluidic biochip. In *Proceedings of the IEEE/ASME International Conference on Mechatronics and Embedded Systems and Applications*, 251–256.

F. Su and K. Chakrabarty. 2005. Unified high-level synthesis and module placement for defect-tolerant microfluidic biochips. In *Proceedings of the Design Automation Conference*. 2, 825–830.

F. Su and K. Chakrabarty. 2006a. Benchmarks for digital microfluidic biochip design and synthesis. Duke University, Dept. of Electrical and Computer Engineering. http://www.ee.duke.edu/~fs/Benchmark.pdf.

F. Su and K. Chakrabarty. 2006b. Module placement for fault-tolerant microfluidics-based biochips. *ACM Trans. Des. Autom. Embed. Syst.* 11, 3, 682–710.

F. Su and K. Chakrabarty. 2008. High-level synthesis of digital microfluidic biochips. *ACM J. Emerg. Technol. Comput. Syst.* 3, 4, Article 1.

F. Su, W. Hwang, and K. Chakrabarty. 2006. Droplet routing in the synthesis of digital microfluidic biochips. In *Proceedings of Design Automation and Test in Europe*. 323–328.

T. Xu and K. Chakrabarty. 2007. Integrated droplet routing in the synthesis of microfluidic biochips. In *Proceedings of the Design Automation Conference*. 948–953.

T. Xu, K. Chakrbarty, and V. K. Pamula. 2010. Defect-tolerant design and optimization of a digital microfluidic biochip for protein crystallization. *IEEE Trans. Comput. Aid. Des. Integr. Circuits Syst.* 29, 4, 552–565.

T. Xu, W. L. Hwang, F. Su, and K. Chakrabarty. 2007. Automated design of pin-constrained digital microfluidic biochips under droplet-interference constraints. *ACM J. Emerg. Technol. Comput. Syst.* 3, 3, Article 14.

P.-H. Yuh, C.-L. Yang, and Y.-W. Chang. 2007. Placement of defect-tolerant digital microfluidic biochips using the T-tree formulation. *ACM J. Emerg. Technol. Comput. Syst.* 3, 3, Article 13.

P.-H. Yuh, C.-L. Yang, and Y.-W. Chang. 2008. BioRouter: A network-flow-based routing algorithm for the synthesis of digital microfluidic biochips. *IEEE Trans. Comput. Aid. Des. Integr. Circuits Syst.* 27, 11, 1928–1941.

Y. Zhao and K. Chakrabarty. 2009. Cross-contamination avoidance for droplet routing in digital microfluidic biochips. In *Proceedings of Design Automation and Test in Europe*. 2, 1290–1295.

Y. Zhao and K. Chakrabarty. 2010. Synchronization of washing operations with droplet routing for cross-contamination avoidance in digital microfluidic biochips. In *Proceedings of the Design Automation Conference*. 635–640.

Y. Zhao, T. Xu, and K. Chakrabarty. 2010. Integrated control-path design and error recovery in the synthesis of digital microfluidic lab-on-chip. *ACM J. Emerg. Technol. Comput. Syst.* 6, 3, Article 11.

Received October 2012; revised January 2013; accepted May 2013