INTEGRATION, the VLSI journal 51 (2015) 169–193

Contents lists available at ScienceDirect

INTEGRATION, the VLSI journal

journal homepage: www.elsevier.com/locate/vlsi

An open-source compiler and PCB synthesis tool for digital microfluidic biochips

Daniel Grissom [a], Christopher Curtis [b], Skyler Windh [b], Calvin Phung [b], Navin Kumar [c], Zachary Zimmerman [a], Kenneth O'Neal [b], Jeffrey McDaniel [b], Nick Liao [d], Philip Brisk [b,*]

[a] Department of Computer Science, Azusa Pacific University, Azusa, CA 91702, USA
[b] Department of Computer Science and Engineering, University of California, Riverside, CA 92507, USA
[c] Samsung Research America, Inc., Mountain View, CA 94043, USA
[d] School of Computer Science, Georgia Institute of Technology, Atlanta, GA 30332-0765, USA

ARTICLE INFO

Available online 17 April 2015

Keywords:
Digital microfluidic biochip
Compiler
PCB synthesis
Compression

ABSTRACT

This paper describes a publicly available, open source software framework designed to support research efforts on algorithms and control for digital microfluidic biochips (DMFBs), an emerging laboratory-on-a-chip (LoC) technology. The framework consists of two parts: a compiler, which converts an assay, specified using the BioCoder language, into a sequence of electrode activations that execute out the assay on the DMFB; and a printed circuit board (PCB) layout tool, which includes algorithms to reduce the number of control pins and PCB layers required to drive the chip from an external source. The framework also includes a suite of visualization tools for debugging, and a collection of front-end algorithms that generate mixing/dilution trees for sample preparation.

1. Introduction

A digital microfluidic biochip (DMFB) is an emerging microfluidic technology that manipulates discrete droplets of liquid on a 2-dimensional electrode grid [62]. DMFBs are poised to enable fully integrated laboratories-on-a-chip (LoCs), which have the potential to miniaturize and automate many chemical and biochemical reactions, with applications to fields such as drug discovery, heathcare and public health, environmental monitoring, and many others. Over the past decade, there has been a significant effort to develop DMFB technology and fabrication processes [14,27,38,55,58]; meanwhile, a small cadre of computer scientists and engineers have worked to develop techniques to program, control, and automatically optimize DMFB architectures.

Despite these fundamental research efforts, very little software has been publicly released. Researchers who work on DMFB compilation introduce new algorithms every year, but these algorithms are developed in isolation, and there appears to be no interoperability between the implementations. Ideally, the algorithms would be compatible with one another and widely disseminated, which would eliminate the need for research groups to implement each other's algorithms to facilitate experimental comparisons. Moreover, it is difficult to identify synergies between algorithms published by different research groups that have not been tested together. Further, both paper writing and software development is prone to human error; and pseudocode in one paper may not precisely match the implementation, or may omit key details. Lastly, researchers who implement algorithms based on pseudocode written by others may inadvertently implement certain steps incorrectly, potentially skewing their experimental results.

To address these concerns, this paper describes in detail an open source DMFB compiler and printed circuit board (PCB) synthesis tool suite developed at UC Riverside. The framework employs well-defined internal data structures and interfaces between each step of the compilation and synthesis process. Several algorithms have been implemented and tested for each stage. The software described in this paper has been released publicly and is available for non-commercial use at [26,73].

The compiler converts biochemical reactions specified using a domain-specific programming language into an executable format appropriate for DMFBs, while the PCB synthesis tool optimizes the control interface and PCB layout. In addition to the core algorithmic features listed above, the framework also includes:

- A software tool that generates mixing and dilution trees for sample preparation.
- Extensive visualization tools that create high-quality graphical output after each step; in our experience, these tools have

* Corresponding author.

http://dx.doi.org/10.1016/j.vlsi.2015.01.004
0167-9260/

proven invaluable for debugging and comparison, and we believe that they will provide similar value to other researchers as well.
- An execution interface to control a direct addressing [62] or active matrix [27,58] DMFB through a microcontroller that communicates with a host PC via USB. This enables direct execution of biochemical reactions on the chosen target device.

We envision three classes of users for the proposed framework: (1) DMFB designers will leverage the back-end synthesis tools to produce optimized PCB layouts, thereby reducing the design, fabrication, and testing costs for their devices; (2) computer scientists and engineers will disseminate and compare optimization algorithms; and (3) biochemists and bioengineers will compile and execute biochemical reactions in their laboratories, thereby increasing scientific productivity through automation. We expect these contributions to be invaluable as DMFB technology evolves over the next decade.

2. DMFB technology overview

Fig. 1 illustrates DMFB technology and the process of droplet transport via electrode actuation. The underlying micromechanical phenomenon that enables droplet transport is called elecrowetting-on-dielectric (EWoD), in which a hydrophobic dielectric layer insulates the liquid droplet from the electrodes. Fig. 2 shows the basic instruction set of a DMFB, which consists of five operations: droplet transport, splitting, merging, mixing, and storage. Additional operations can be realized by adding external devices to the device such as heaters [45], optical detectors [47,87], integrated sensors [13,57,68], etc.

Fig. 3 illustrates the algorithmic stages of the DMFB compiler and PCB synthesis toolflow. The input to the framework is a biochemical reaction to perform, which is specified as a directed acyclic graph (DAG). One fundamental limitation of this input, as well as the framework in its current form, is that control flow operations are not supported. Future work will integrate support for real-time decision-making based on sensory feedback, fault tolerance and error recovery, etc.

The compiler's output (Fig. 4) is a linear state machine (a Moore machine): each state outputs a bit-vector, where the bits that are set to '1' correspond to the subset of electrodes that are activated during each actuation cycle. A typical actuation cycle lasts for 10 ms, as reported in previous literature [60].

The compiler must solve three interdependent NP-complete problems: scheduling [69], placement [70], and routing [7]. The scheduler determines the time steps at which each biochemical operation occurs, while satisfying droplet dependency constraints as well as physical resource constraints of the target device. The placer determines the location on the 2D electrode array where each operation is performed as the reaction progresses over time. The router ensures that droplets are transported from their start/stop points at appropriate times during execution of the chemical reaction, while ensuring that droplets undergoing transport do not inadvertently collide with one another or any other ongoing operations on the chip. The router may also introduce wash droplets to remove residue left by droplets that travel over the surface of the chip; this ensures that droplets that enter the same region at different times do not accidentally contaminate one another and spoil the results of the reaction.

PCB synthesis is composed of two steps: pin-assignment and wire routing. The simplest pin assignment is direct addressing, in which a dedicated control input drives each electrode, as shown in Fig. 5(a). An optimized pin assignment establishes a one-to-many assignment from control inputs to electrodes, as shown in Fig. 5(b); moreover, some electrodes may be left unaddressed. Reducing the number of control inputs reduces the two-dimensional area of the PCB, which in turn, reduces cost. Both application-specific [9,33–37,40,41,56,76,79,80,84–86] and general-purpose [20,25,46] pin assignments have been reported in the literature. Wire routing connects each control input to the one or more electrodes that it drives, and determines the number of PCB layers required to realize the chip; reducing the number of layers also reduces cost. PCB routing for pin-constrained DMFBs is a form of escape routing. Fig. 6 shows application-specific and general-purpose pin assignment and wire routing solutions.

3. Framework input specification

As shown in Fig. 3, the input to the framework is a DAG, in which vertices represent chemical operations and edges represent the transfer of droplets between operations. In the preliminary version of the framework, the user specifies the DAG by writing a text file, which is somewhat tedious and cannot effectively scale to large protocols with hundreds or thousands of operations. To ease the burden of programmability, we introduce two new input specification options here.

3.1. BioCoder language and compiler

BioCoder is a domain-specific language for biochemical reaction specification developed at Microsoft Research, India [1]. BioCoder's original purpose was to standardize the specification and dissemination of chemical reactions in the scientific literature. Historically, chemists and biologists describe the written account of a reaction that they have performed using English language prose, which may be ambiguous or incorrect, depending on the aptitude of the writer. This stands in stark contrast to computer science and computer engineering, in which pseudocode of one form or another is standard for the widespread dissemination of algorithms.

BioCoder is a C++ library that provides functions for a wide variety of operations performed in chemistry (e.g., measurement, incubation, etc.). The programmer specifies a reaction using the

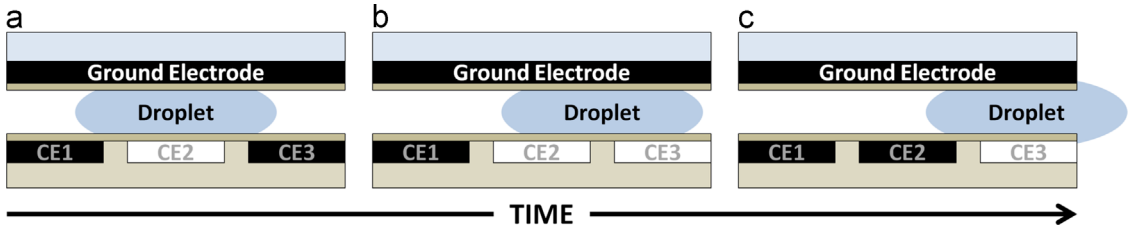


Fig. 1. A DMFB is a 2-dimensional planar array of electrodes; (b) a cross-sectional view of a DMFB, showing the ground electrode on top and a droplet sandwiched in-between; (c) illustration of droplet transport by a sequence of activated electrodes (activation is shown in white).

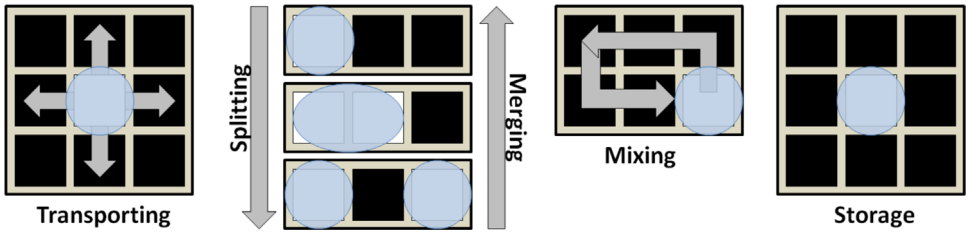


Fig. 2. The instruction set of a DMFB consists of five basic operations.

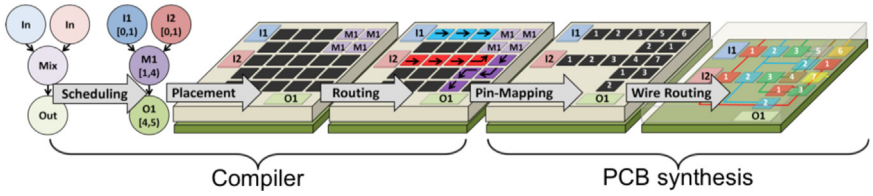


Fig. 3. The DMFB compiler consists of three stages: scheduling, placement, and droplet routing; the PCB synthesis flow adds pin-mapping and wire routing stages.

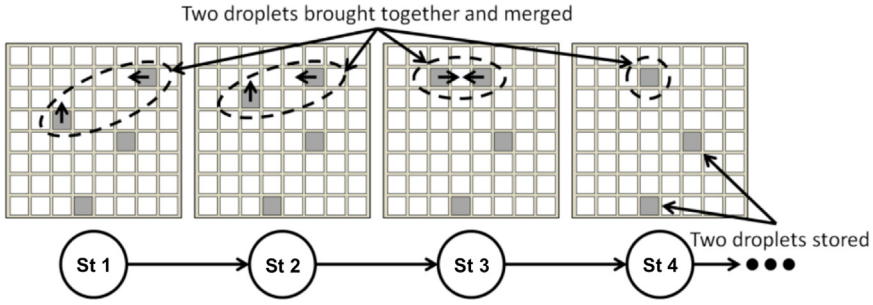


Fig. 4. The output of a DMFB compiler is a linear state machine in which each state specifies a subset of electrodes to activate at a given time.

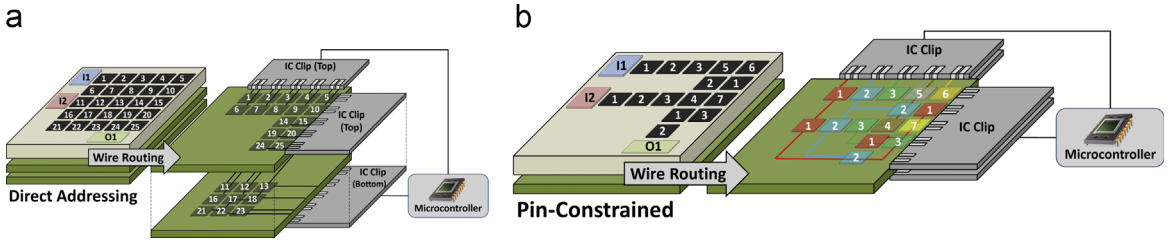


Fig. 5. A DMFB is typically mounted on top of a PCB that delivers signals provided by a microcontroller via one or more IC clips. The PCB may also include shift registers that latch the signals provided by the microcontroller. (a) A direct addressing DMFB presumably requires multiple PCB layers due to the large number of control signals that need to be routed; (b) a pin-constrained DMFB may require fewer PCB layers due to the reduced number of control inputs.

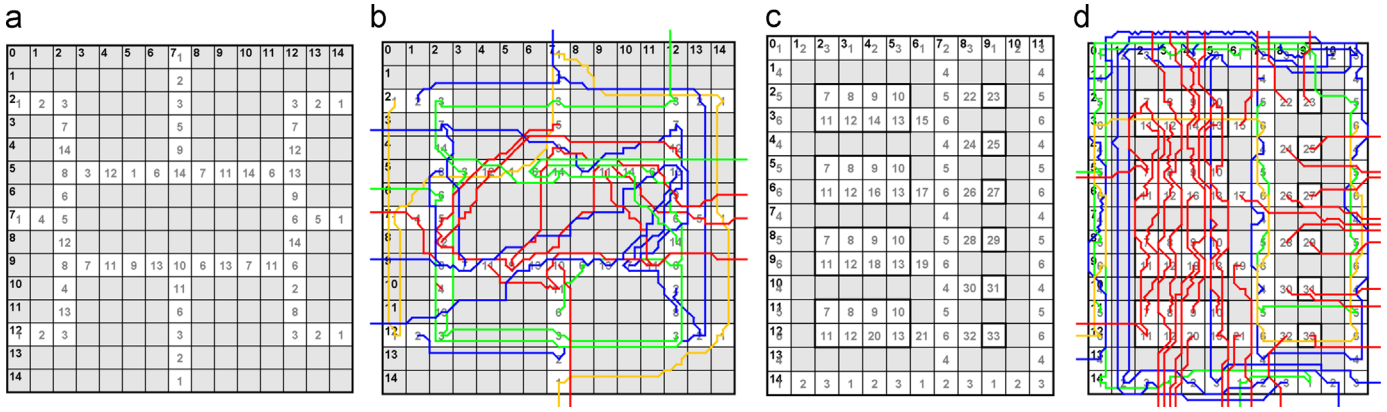


Fig. 6. (a) Pin assignment and (b) wire routing solutions for an application-specific DMFB designed for the polymerase chain reaction (PCR) [46,86]. (c) Pin assignment and (d) PCB escape routing solutions for a general-purpose pin-constrained DMFB that can execute and biochemical reaction that can fit within the space provided [52]. Each PCB routing layer is represented by a different color. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

library; the compiler then converts it into a DAG-based intermediate representation, and outputs an unambiguous step-by-step English-language description of the reaction, which has the look and feel of a cookbook recipe. This description can then be copied into a scientific paper or published online for widespread dissemination.

We modified BioCoder for use with our framework [24]. We removed all library functions that are incompatible with DMFB technology (e.g., declaration of solid materials, centrifugation, etc.). We added a droplet split operation, which was not supported by BioCoder in its original form. Lastly, we modified the compiler to convert its internal DAG representation into our text file format for use with the framework; we did not break the English-text back-end so dissemination of chemical reactions targeting DMFBs remains possible. Fig. 7 shows an illustrative example of a DMFB-compatible BioCoder biochemical reaction and its internal DAG representation.

We expect to continuously update the DMFB-compatible BioCoder library to support new integrated sensing technologies that will be developed in the future.

3.2. Automated sample preparation

Sample preparation is an important step in many analytical chemistry techniques, which are often non-responsive to the analyte in its in-situ form. Often a sample may need to be diluted to a desired concentration (or set of concentrations) and/or possibly mixed with several reagents while doing so. Griffith et al. [19] and Thies et al. [72] introduced the first sample preparation algorithms tailored for DMFBs; since 2010, there has been an explosion of literature on the topic. We have implemented many, but not all, of these algorithms in a standalone software package that outputs a DAG in our framework's text file format. The user specifies the input(s) (e.g., the desired concentration of one or more output droplets) using a graphical user interface.

Table 1 lists the algorithms that we have implemented to date. In Table 1, the term CV is shorthand for *concentration value*, i.e., the desired concentration of a target droplet. Six of the algorithms that we have implemented generate a dilution tree that produces one droplet of a desired CV, using different optimization strategies [11,19,29,65,66,72]; four more algorithms generate multi-output DAGs (sometimes, but not always, a forest of trees) that produce multiple droplets of desired CVs [4,28,30,54]. The aforementioned algorithms all assume that one sample fluid is diluted with a buffer (a non-reactive fluid that maintains the pH of the sample fluid during dilution).

The Generalized Dilution Algorithm (GDA) [67] assumes that the user requests a sequence of droplets with varying CVs in an online fashion, i.e., the CV values are not known in advance. Each time the algorithm produces a droplet with a new target CV, intermediate droplets that are not used may be generated. In other algorithms, these droplets would be treated as waste; in contrast, GDA stores these intermediate droplets opportunistically on-chip, as they can possibly be used to assist future dilution operations.

The algorithm of Bhattacharjee et al. [6] is optimized to produce multiple droplets having a linear dilution gradient, i.e., a sequence of CVs that differ by a constant factor, e.g., 5%, 10%, 15%, 20%, 25%, etc. This algorithm is optimized specifically for linear dilution gradients, but is otherwise ineffective for arbitrary sets of target CVs.

Lastly, two algorithms—MinMix [72] and Common Dilution Operation Sharing (CoDOS) [42]—produce a single droplet of a desired target CV using multiple reactants for dilution, rather than just buffer exclusively; the authors of CoDOS also introduced a multi-target CV variation as well.

4. Framework implementation

This section describes the compiler and synthesis algorithms that we have implemented and made publicly available within the framework. We have not implemented all algorithms that have been published in the literature to date, so this discussion is not meant to be a comprehensive literature survey. We focus instead on the software architecture of the framework and briefly summarize the algorithms that are available for use, modification, and experimental comparison by others.

4.1. Software architecture

Fig. 8 illustrates the software architecture of the microfluidic framework. Black boxes represent software modules (compiler/synthesis tasks) and/or visualization tools; the document boxes represent human-readable plaintext files that are produced and/or consumed by

```
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");
```

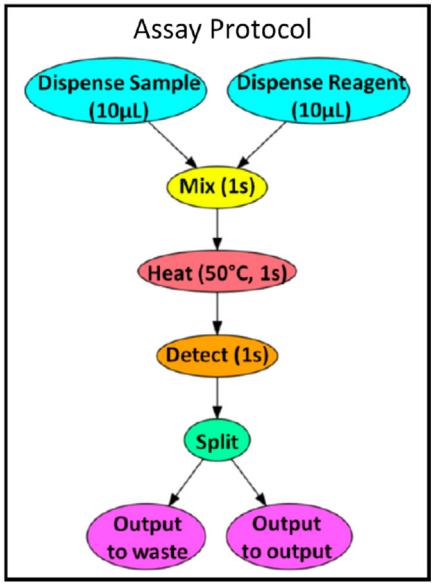


Fig. 7. A chemical reaction specified using our modified version of the BioCoder language (left) converted to a DAG (right).

Table 1
Sample preparation algorithms implemented using our framework.

Algorithm	Reference	Target problem	Optimization goal
Binary search (BIN)	Griffith et al. [19]	Single target CV	Reduce waste
Bit scanning (BS)	Thies et al. [72]	Single target CV	Reduce #mix-split steps
Dilution and mixing w/reduced wastage (DMRW)	Roy et al. [65]	Single target CV	Reduce #mix-split steps
Improved dilution/mixing Algorithm (IDMA)	Roy et al. [66]	Single target CV	Reduce reagent usage and waste
Reactant minimization Algorithm (REMIA)	Huang et al. [29]	Single target CV	Reduce reagent usage
Graph-based optimal reactant Minimization algorithm (GORMA)	Chiang et al. [11]	Single target CV	Reduce reagent usage and waste
Min–Mix (MM)	Thies et al. [72]	Single target CV (multiple reactants)	Reduce # mix-split steps
Common Dilution Operation Sharing (CoDOS)	Liu et al. [42]	Single or multi-target CV (multiple reactants)	Reduce waste
Generalized Dilution Algorithm (GDA)	Roy et al. [67]	Single target CV from multiple CVs of the same fluid	(i) #Mix-split steps, or (ii) Height of mixing/dilution tree, or (iii) Number of mixers used
Zero waste linear dilution Gradient	Bhattacharjee et al. [6]	Linear dilution gradient	Reduce waste
Multi-target Min–Mix	Bhattacharjee et al. [4]	Multiple target CV	Reduce #mix-split steps and waste
Reagent saving mixing Algorithm (RSMA)	Hsieh et al. [28]	Multiple target CV	Reduce reagent usage and waste
de Bruijn Graph Traversal	Mitra et al. [54]	Multiple target CV	No intermediate storage
Waste recycling algorithm (WARA)	Huang et al. [30]	Multiple target CV	Reduce reagent usage and waste

the software modules that comprise the simulator. Compiler and synthesis algorithms are implemented in C++, while visualization tools (described in Section 4.4) are implemented in Java. Each stage of the framework outputs a human-readable text file, which can be read as an input to the next stage or a visualization tool. An externally available, graph-drawing tool, GraphViz [18], can visualize the assay specification (as a DAG) as it is modified and annotated by the intermediate steps of the synthesis flow.

The synthesis flow is modular. Each step can execute as a standalone command-line program, or all steps can execute atomically; in the latter case, output of intermediate files for visualization purposes is optional, as each stage can propagate its internal data structures to its successor. When one or more steps executes as a standalone program, the synthesis engine uses a built-in utility class to perform I/O and to construct and destruct all internal data structures. The utility function performs file I/O in a manner that is transparent to the user; thus, we do not describe the syntactical structure of the interface files; our source code documentation describes the interface files' syntax in detail [73].

Schedules are computed on the granularity of time-steps, which are typically 1 s for most assays, under the assumption that routing overhead is negligible. An assay operation must start at the start of a time-step, and must finish at the end of a time-step. After scheduling, the assay completion time is not exactly known, because routing incurs some small, non-negligible overhead. A *routing sub-problem* may occur between every pair of time-steps; within each sub-problem, a subset of the droplets on the device may need to be transferred from one location to another [71]; the time required to perform these transfers must be added to the assay completion time in order to achieve a more accurate result. If the schedule produces T time steps, then there are $T+1$ routing sub-problems (the first routing sub-problem involves routing droplets from input reservoirs to the locations where the first operations are performed; the last routing sub-problem removes all remaining droplets from the chip upon completion of the assay). Let $L(t)$ denote the latency of each time-step, and $L(r_i)$ denote the latency of the solution to routing sub-problem r_i. Then the total assay completion time is $TL(t)+L(r_0)+\ldots+L(r_T)$.

Unlike Fig. 3, the pin-mapper is not an official stage of the framework akin to scheduling, placement, routing, etc. The data structure that represents assignment of control pins to electrodes in the array is an internal data structure that is "owned" by the architecture. The literature on pin-constrained DMFB design is vast, and includes: (1) pre-computed pin mappings that require specialized scheduling, placement, and routing algorithms [20,25,46]; (2) "pre-synthesis" array partitioning methods that update the pin-map during scheduling, placement, and routing [9,33,40,41,56,84–86]; and (3) "post-synthesis" pin assignment algorithms that derive a pin-map (and, in some cases, a wire routing solution) starting with a fully-compiled assay [34,36,37,76,79,80,86]. Rather than extending the framework to include every possible ordering of tasks vis-à-vis pin assignment algorithms, we set up an initial direct addressing pin-map before scheduling. Other stages may modify the pin map as desired, converting the target chip into a pin-constrained design.

4.2. Internal data structures and workflow

When the framework executes the complete compiler and synthesis flow atomically as a standalone program, the internal data structures are modified and passed along from one stage to the next. Each stage has a clearly defined interface with common I/O format. As shown in Fig. 9, the synthesis engine (class) maintains an instance of a scheduler, placer, (droplet) router, and (PCB) wire router. These instances inherit from global classes that force all algorithms that implement each step to use the same functions, parameters, and internal data structures for interfaces. This minimizes the number of framework-level changes that a user must make to implement a new algorithm.

In Table 2, the synthesis engine includes six internal data structures, which are passed between stages. Table 2 shows which synthesis steps read (R) and write (W) these data structures, and which methods do not access them at all (–). A discussed above, the pin-mapper (PM) is not an explicit stage of the framework, but is instead internal to the data structure representing the DMFB architecture. Pin mapping can modify the architecture (i.e., any algorithmic decision that alters the assignment of control pins to electrodes) or the electrode activation sequence (i.e., altering the pin-map will change which electrodes are activated when a control pin is activated).

All interface files are human-readable, well-structured text files. A utility class called *Util* handles all file I/O, according to

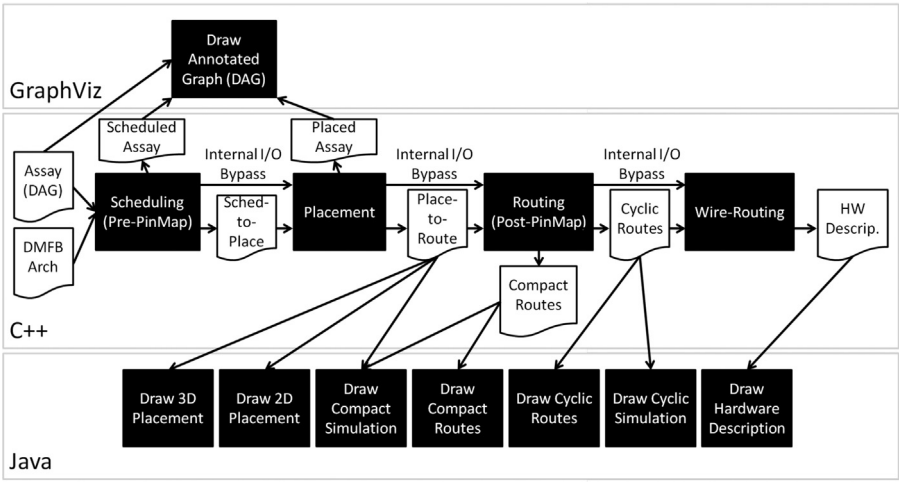


Fig. 8. Software architecture of the microfluidic framework, including compiler and synthesis algorithms (written in C++) and visualization tools.

the I/O specifications (please refer to our publicly available source code [73] for details). *Util* reads the appropriate file to populate all internal data structures when a stage of the flow begins, and outputs its result to a text file when the stage is finished.

4.2.1. Scheduler input interface

The scheduler accepts two input files: an assay specification (a DAG), and a DMFB architectural description, (Fig. 8). The assay specification contains information needed to construct the DAG; the *Util* class reads the text file and creates a data structure of type *DAG* called *dag* to represent the assay. *Util* annotates nodes with the operation type, length, and other relevant information to help understand the assay; the nodes do not (yet) contain any information about start times, stop times, or module placement. The architecture description contains: the dimensions of the chip; the locations of any I/O reservoirs (on the periphery); any fixed resources (e.g. heaters and detectors); an initial pin-map (if any; the default is direct addressing); the droplet actuation frequency; the time-step length in seconds; and two parameters (*hcap* and *vcap*), which describe relevant aspects of the PCB technology to the wire router. *Util* creates a data structure of type *DmfbArch* called *arch* and populates it with this information.

The user may optionally instruct the framework (primarily the router) to perform wash droplet routing to prevent cross-contamination. To support wash droplet routing, the architecture description must specify at least one input port dedicated to wash droplets; waste ports dedicated to wash droplets are optional.

4.2.2. Scheduler

The scheduler is called with *dag* and *arch* passed as parameters, as shown in Table 2. The scheduler computes the start and stop times of each node, and the module type to which it is bound (e.g., a mixer, heater, detector, I/O, etc.).

4.2.3. Scheduler-to-placer interface

After scheduling, the *Util* class creates an output file that is provided to the placer; it does this by flattening *dag* and *arch* into a single text file, including all information added to the nodes by the scheduler. Prior to placement, an empty vector of reconfigurable modules, *rModules*, is allocated, and *dag* and *arch* are either recreated by *Util* (if the placer runs in standalone mode), or passed from the scheduler to the placer.

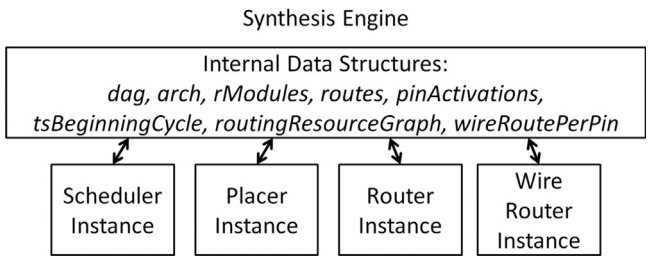


Fig. 9. The synthesis engine contains an instance of a scheduler, placer, (droplet) router, and (PCB) wire router, as well as internal data structures passed between synthesis steps.

4.2.4. Placement

The placer performs two tasks: (1) it creates a reconfigurable module, *rMod*, with a unique id number for each non-I/O node in *dag* and adds it to *rModules*; and (2) it binds *rMod* to a node in *dag* via pointers (e.g., *dag*→*node*→*module*=*rMod*); it also binds each I/O operation to a valid port (e.g., *dag*→*node*→*ioPort*=*port*).

4.2.5. Placer-to-droplet router interface

Once placement completes, *Util* flattens *dag* (which now contains references to specific modules in *rModules*), *arch* and *rModules* to produce a text file to be passed to the router. Prior to invoking the router, *Util* recreates *dag*, *arch* and *rModules* from the interface file, along with two new empty data structures. The first data structure is a list of droplet routes called *routes*, as shown in Table 2. The *RoutePoint* structure represents the (*x*, *y*)-coordinate of a droplet, the cycle number representing when the droplet is at that given coordinate, and the droplet's status (e.g. waiting, processing, etc.), which can be useful for debugging. The second data structure is a vector of cycles called *tsBeginningCycle*, which dictates the cycle at which each timestep begins. Starting at time-step 0, a cycle should be added to *tsBeginningCycle* for each time-step. This information is determined in the router because a time-step cannot officially begin until all droplets have been routed to their destinations.

4.2.6. Droplet router

The router is called with 7 parameters: *dag* (scheduled and placed), *arch*, *rModules* (populated by the placer), *routes* (empty), *pinActivations* (empty), *tsBeginningCycle* (empty) and *performWash*, a boolean flag which dictates if wash droplet routing is performed to clean residue left behind by previous droplets.

Table 2
List of internal data structures used by the framework, indicating whether the scheduler (S), placer (P), (droplet) router (R), pin-mapper (PM), or (PCB) wire router (WR) reads (R) or reads and writes (W) the data structure.

C++ type and name	S	P	R	PM	WR
DAG *dag	W	W	R	–	–
DmfbArch *arch	R	R	R	W	W
vector < ReconfigModule* > *rModules	–	W	R	–	–
map < Droplet*, vector < RoutePoint* > * > *routes	–	–	W	–	–
vector < vector < **int** > * > *pinActivations;	–	–	W	W	W
vector < **unsigned long long** > *tsBeginningCycle	–	–	W	–	–
WireRoutingModel* routingResourceGraph	–	–	–	–	W
vector < vector < WireSegment * > * > *wireRoutesPerPin	–	–	–	–	W

The router instantiates a droplet–route pair, *pair*(*Droplet*∗, *vector* < *RoutePoints*∗ > ∗), for each droplet, which is added to *routes*. The router adds a new *RoutePoint* for each actuation cycle that the droplet is on the DMFB. Thus, the router maps each droplet to a vector of *RoutePoints*, which wholly characterizes the droplet's route. A droplet must have a *RoutePoint* for each actuation cycle along its route. This is not the most time- and space-efficient representation of a droplet's route; however, it is easy to use and understand, and has proven useful for algorithm development and debugging.

Wash droplet routes are represented using the same data structures as a normal assay droplet's route, as described in the last paragraph. The *Droplet* class has a Boolean member called *isWash*; the only difference between a wash droplet route and a normal assay route is the status of the *isWash* variable in the Droplet class associated with the route. When creating new droplets, the router sets the *isWash* member of any droplets originating from a wash port as *true* for proper recognition by the simulator and visualizer. When routing wash droplets, the router may track contaminated cells that require cleaning in any way that is convenient to the particular algorithm; routes are independently analyzed afterward to determine if cross-contamination occurs.

The *tsBeginningCycle* data structure contains the exact time at which each time-step of the schedule begins. Initially, the schedule is computed under the assumption that droplet routing times are zero [69]; the router computes actual routing times after scheduling and placement [71]. The exact time at which scheduling time-step $t+1$ begins is equal to the time at which scheduling time-step t ends, plus the time taken to route all droplets, which the router computes for the $(t+1)$st routing sub-problem (i.e., the routing sub-problem between time-steps t and $t+1$).

Lastly, the router generates a pin actuation sequence, which is encapsulated in a vector of bit-vectors called *pinActivations*. The vector *pinActivations*[i] represents the set of control pins that are activated during the ith actuation cycle.

4.2.7. Droplet router–wire router interface

Once routing completes, *Util* flattens *dag*, *arch*, *rModules*, *routes*, *tsBeginningCycle*, and *pinActivations* to produce a text file to be passed to the wire router. Prior to invoking the wire router, *Util* recreates the same set of data structures from the interface file. The wire router does not read or write any of the data structures generated by the droplet router; however, it still unpacks them, as it generates additional data structures that will be propagated to its output interface.

4.2.8. Wire router

To compute the PCB wire routes shown in Fig. 6, the wire router constructs and maintains a routing graph [49,52,78]. The wire graph is generated automatically from two user-specified parameters: the *horizontal capacity* (*hcap*) and the *vertical capacity* (*vcap*). Without loss of generality, the hcap is the number of PCB wires that can be safely routed between two electrodes in the horizontal direction; vertical capacity is defined analogously. The *diagonal capacity* (*dcap*), which is typically higher than the hcap or vcap [78], is derived from the hcap and vcap values and particular underlying wire routing structure that is employed.

The routing graph is generated deterministically from hcap and vcap. It has a repeatable structure that scales directly to the size of the array. Fig. 10 shows an example where $hcap = vcap = 3$ and $dcap = 7$.

The routed PCB is a netlist that describes the routing graph resources (vertices and edges) that route each net. In Table 1, this netlist is represented by the *wireRoutesPerPin* data structure, which is a vector of vectors of *WireSegments* (a simple data structure with a beginning and end node, each of which represents a specific XY location on the physical PCB). The *wireRoutesPerPin* data structure is created as an empty structure and contained by the base *WireRouter* structure/class. The outer vector is indexed by the pin number; thus, *wireRoutesPerPin.at*(4) returns the *WireSegments* contained in Pin 4's netlist. Each net is routed to exactly one control pin on the perimeter of the device. In a direct addressing chip, each net terminates at exactly one electrode; in a pin-constrained chip, each net fans out to multiple electrodes that are controlled by the same pin. The same routing graph can be reused for each layer to support a multi-layer PCB process. All of the nets that are routed on the same layer are grouped together for visualization.

4.2.9. Wire router–output interface

Once wire routing completes, *Util* flattens *dag*, *arch*, *rModules*, *routes*, *tsBeginningCycle*, *pinActivations*, *wireRoutesPerPin* and the dimensions of *routingResourceGraph*, to produce a text file. The data structures are unpacked later and provided to the visualization tool suite, as described in Section 4.5.

4.2.10. Interface bypass

The interface files created by each framework stage are used as input to the visualization suite, which was written in Java. In practice, visualization is optional, and the compiler/synthesis flow can run as one atomic program, passing all internal data structures between stages; suppressing file I/O reduces runtime and file system clutter.

4.3. (Implemented) compiler and PCB synthesis algorithms

This section summarizes the compiler and PCB synthesis algorithms that have been implemented within the framework; this discussion is not meant to be a comprehensive literature survey, as many published algorithms have not yet been implemented.

4.3.1. Scheduling

We have implemented most of the scheduling algorithms that have been reported in the literature, including greedy heuristics (list

scheduling [22,69], force-directed list scheduling [59], and path scheduling [23]), as well as two genetic algorithms that repeated call list scheduling with randomly-generated operation priorities [63,69]. Our papers describing force-directed list scheduling [59] and path scheduling [23] include comparisons with the other algorithms using the framework.

Two algorithms that have not (yet) been implemented within the framework include optimal scheduling via integer linear programming (ILP) [15,69] and a heuristic called *Latency Optimal Scheduling with MOdule Selection* (*LOSMOS*) [43]. The ILPs that have been published thus far are specific to two different assay topologies, and we have not yet tried to generalize them. LOSMOS includes a module selection step that alters the number of electrodes required to perform mixing; using more electrodes speeds up mixing time, but consumes more on-chip resources, thus limiting concurrency.

4.3.2. Placement

We have implemented and released three different placement algorithms: one based on *simulated annealing* [70], an algorithm originally proposed for dynamically reconfigurable FPGAs called *Keep All Maximum Empty Rectangles* (*KAMER*) [2] (using a data structure introduced by Lu et al. [44] to represent free space on the chip), and a pre-placement technique that we refer to as a *virtual topology* [21,22]. Fig. 11(a) illustrates the basic principle of KAMER, while Fig. 11(b) illustrates a virtual topology.

When a new operation needs to be placed, KAMER queries a data structure the stores all of the *Maximum Empty Rectangles* (*MERs*), representing free space on the chip, and selects a MER in which to place the operation; it then updates the data structure and places the next operation. When an operation completes, it is removed from the chip and the MER is reconstructed. The virtual topology, in contrast, pre-determines the regions of the chip where operations can occur (called *modules*) and lays them out in a way that a routing path between every pair of modules is guaranteed; placement is then converted to a *binding* problem, in which any free module that can support an operation is chosen (e.g., any module can mix or store droplets; however, to perform detection or heating, a module with an associated detector or heater is required).

Placement algorithms that have not yet been implemented include a variation-aware ILP [39] and a reliability-aware placer that uses 3D deferred decision-making [10]. We also have not (yet) implemented several algorithms that perform scheduling in conjunction with placement using iterative improvement algorithms [50,75,77]. At present, we do not support algorithms that modify the DMFB architecture (other than pin-mapping). Consequently, we have not implemented a 3D placer that subsumes scheduling and placement (the third dimension being time) [82], because its internal data structure (the T-Tree) is incompatible with DMFBs where external devices (detectors, heaters, etc.) are pre-placed on the chip; likewise, we have not implemented a contamination-aware placer that tries to reduce the number of crossing droplet routes [40], as this particular

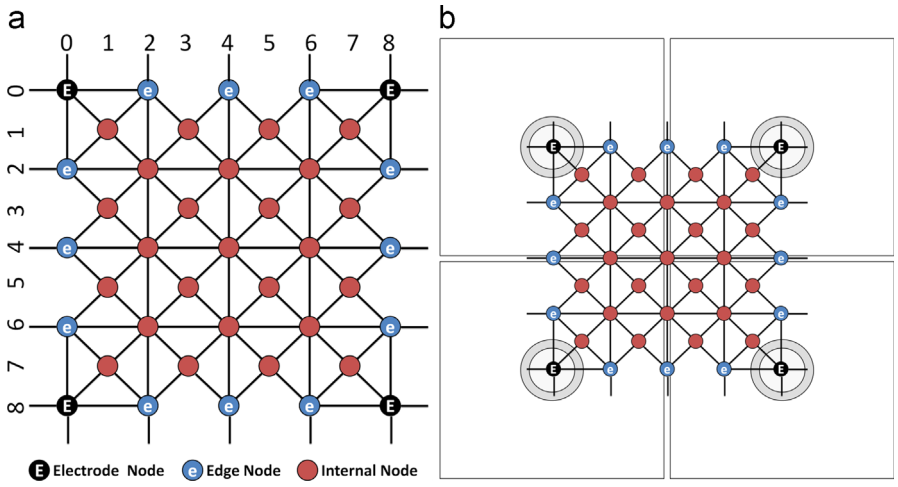


Fig. 10. (a) A routing graph tile used for PCB escape routing along with its planar coordinate system; (b) an overlaid routing graph on top of a 2×2 electrode grid representing physical locations on the PCB.

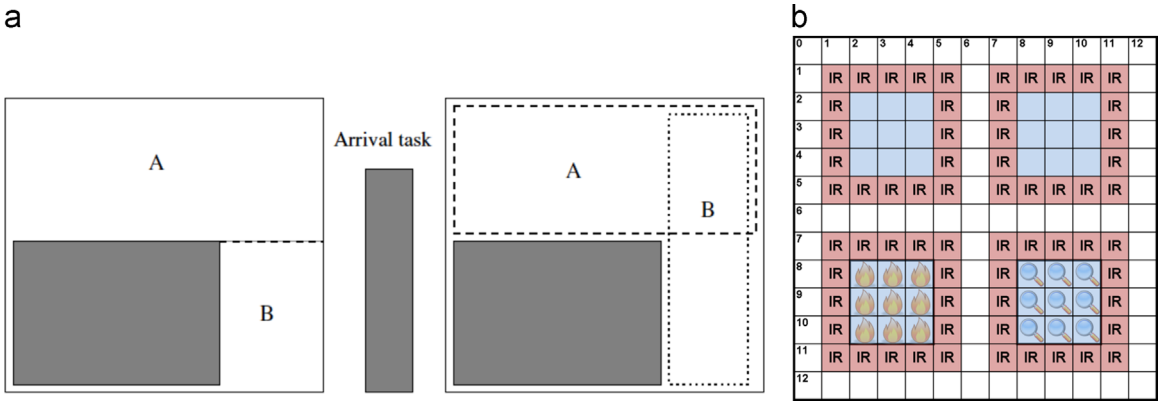


Fig. 11. (a) Illustration of the way that the KAMER algorithm represents free-space on the chip using (possibly overlapping) maximum empty rectangles (MERs) [2,44]. (b) Illustration of a 4-module virtual topology [21,22], where one of the four modules has a heater and another has a detector.

algorithm modifies the location and ordering of I/O reservoirs on the chip's perimeter.

4.3.3. Droplet routing

Many papers on DMFB routing have been published in recent years [7,12,19,21,22,32,35,64,71,81,83]; since assay operations (mixing, detection, etc.) are several orders of magnitude slower than droplet transport latencies, we have not prioritized a thorough implementation of all known droplet routing algorithms. The best and most stable router invokes Soukup's grid routing algorithm to compute individual paths for each droplet, under the assumption that they will be routed one-at-a-time [21,22,64]; a greedy *compaction* procedure then introduces concurrent transport while ensuring that all droplets follow their routes.

We have implemented several other droplet routing algorithms as well, including a high-performance droplet router described by Cho and Pan [12], and a compaction algorithm based on dynamic programming described by Huang et al. [32]. We have also implemented several droplet routing algorithms that are specific to a general-purpose pin-map that we call a field-programmable pin-constrained DMFB (FPPC-DMFB) [20,25].

4.3.4. Routing-based synthesis

Routing-based synthesis is an alternative to the traditional approach of scheduling/placement/routing, in which droplets that mix travel around the board in arbitrary directions, as opposed to rotation around a pivot [51]. This eliminates the traditional notion of rectangular modules that must be placed with a separate droplet routing stage. All routes are generated probabilistically, and the exact routes taken by each operation depend on the random number seed. Additionally, mixing operation latencies now depend on the taken route, as different movements (straight, turn left/right, reverse) have different contributions to the total mixing time [51,60].

Under the traditional model of DMFB scheduling [15,22,23,43,59,63,69], the latency of each operation is known statically. Under routing-based synthesis, the latency of operations such as mixing and dilution is not determined until the routes are known, thus scheduling and routing must be performed together [51]. Our implementation of routing-based synthesis is implemented in the routing stage of the framework, i.e., the scheduler and placer take no actions; our implementation supports wash droplets for cross-contamination removal as an option [51]. We have integrated (unpublished) extensions to reduce the likelihood of deadlock and livelock and to support non-reconfigurable operations such as heating, detection, I/O, etc. Under routing-based synthesis, the chip is partitioned, and each wash droplet is tasked to proactively remove contamination within each partition. Our implementation improves the efficiency of wash droplet routing when multiple droplets simultaneously contaminate a partition.

4.3.5. Pin mapping

Thus far, we have implemented four pin mappers within the PCB wire routing stage. The first pin mapper processes the electrode activation sequence produced by the router, constructs a compatibility graph between electrodes that could potentially share a pin, and computes a pin mapping solution by partitioning the compatibility graph into cliques [86]. Several subsequent papers extended this approach to further optimize for power and/or reliability [34,36,79,80], and/or to incrementally compute a wire routing solution in conjunction with pin mapping [37,79,80]. We have implemented three of these algorithms: one power-aware pin mapper [36], and two reliability-aware pin mappers [34,80], the second of which [80] also performs PCB wire routing.

4.3.6. PCB wire routing

Wire routing for DMFBs is a variation of the escape routing problem for general PCB routing, in which a route from each pin to the perimeter of the chip must be found; it is not required to connect the route directly to an external control pin. For direct-addressing DMFBs, the problem is identical; for pin-constrained DMFBs, the only conceptual difference is that a single escape route must be computed that connects to all the electrodes driven by the same control pin. Issues such as critical path length or total wirelength are not optimization criteria for escape routing.

We have implemented one PCB wire routing algorithm, which works well enough for our purposes [52]. The algorithm is based on the paradigm of negotiated congestion, which was originally applied to FPGA routing [53]. The original negotiated congestion-based escape router considered only single-terminal nets [49]; we modified it to support multi-terminal nets using the approach taken by the FPGA routers [52].

One paper has described a DMFB wire router based on ILP, which can also handle blockages due to external devices [8]; our router can also handle blockages, as the corresponding vertices are simply omitted from the routing resource graph. We have not yet implemented this ILP-based router.

4.4. Verification and validation

Each stage of the compiler and PCB synthesis flow is followed by a short verification routine, which ensures that the output of the stage is correct. The verification routine is particularly helpful for software developers who are integrating new algorithms into the framework for comparison and new rules can easily be added by future developers.

4.4.1. Scheduling

A number of rules have been incorporated into the schedule analysis tool:

- Each DAG vertex must have the proper number of parent/child nodes (e.g., a dispense operation should have exactly 0 parents and 1 child).
- Each vertex must be bound to a resource (i.e., I/O port or module).
- The scheduler may not allocate more resources than available at any time-step (e.g., 3 simultaneous detects should not be scheduled if there are only 2 detectors)
- Each vertex in the DAG must be scheduled to begin after the completion time of all of its predecessors.
- There may be no gaps in between parent and child vertices. For example, if vertex v completes at the end of time-step 4, all of its children must start at the beginning of time-step 5. The scheduler must insert single-cycle storage operations (which must be bound to physical on-chip resources) if a droplet is consumed at a later time-step.

4.4.2. Placement

Placement rules have been included to analyze the validity of the placement or binding solution:

- Two modules cannot be placed on the same cell(s) at the same time.
- Two modules cannot be placed on directly adjacent cells at the same time (i.e., there must be an interference region of at least 1 cell between adjacent modules).
- All assay operations must be bound to a module or I/O port; all modules must be bound to an assay operation.

- At most one module can be bound to a single assay operation at any given time-step.

In addition, the analyzer reports warnings in cases where a rule violation may be acceptable in certain cases, but is often detrimental:

- Generally, a module should not be placed in a location that causes its interference region to overlap with a cell adjacent to an I/O port; however, this may be acceptable if the I/O port is not used at any point while the module is active.

4.4.3. Droplet routing

Several rules are included to analyze the validity of the droplet routes for proper simulation and visualization:

- Droplet routing points marked as "OUTPUT" must be adjacent to an output port.
- Consecutive routing points must describe an orthogonal move (UP, DOWN, LEFT, or RIGHT). Droplets may not make diagonal moves in one cycle, and all movements must be to an adjacent electrode; "jumping" over electrodes is not permitted.
- For each droplet route, there should be no gaps in consecutive routing points.
- No two droplets may enter one another's interference regions (Fig. 12) unless they are splitting or merging.

4.4.4. Droplet routing with wash droplets

Post-routing verification tracks when electrodes are contaminated (by an assay droplet passing over the surface) and cleaned (a wash droplet passes over a contaminated electrode).

- No assay droplet may cross a contaminated electrode, unless it will mix with the droplet that caused the contamination.

4.4.5. Assay correctness

After routing, the framework validates the following two rules:

Droplet Conservation: any droplet that enters the system must terminate with a status of MERGED or OUTPUT. (When two droplets are merged, the resulting droplet receives the lower ID number of the two droplets. The droplet with the higher ID number terminates with MERGED status.)

The total volume of fluid injected into the system from input reservoirs must be equal to the total volume of fluid output. The volume and composition of each droplet is updated upon each merge and split operation, in order to keep track of this property. For example, if 15 10 μL droplets (150 μL total) of fluid A is input, 150 μL of fluid A must be output, although the total quantity may be distributed among a larger number of droplets due to mixing and dilution).

4.4.6. Pin mapping

A legal pin mapping must satisfy the following straightforward criteria:

- Each electrode must be driven by at most one control pin.
- Each control pin must drive at least one electrode.

Any electrode that is not driven by a control pin can be eliminated and replaced with white space; it is perfectly legal to do this, as the additional space is usable by the PCB wire router. Any control pin that addresses no electrodes should be removed as well; although it is not an issue of correctness, per se, unused control pins should be removed to reduce the I/O cost of the chip.

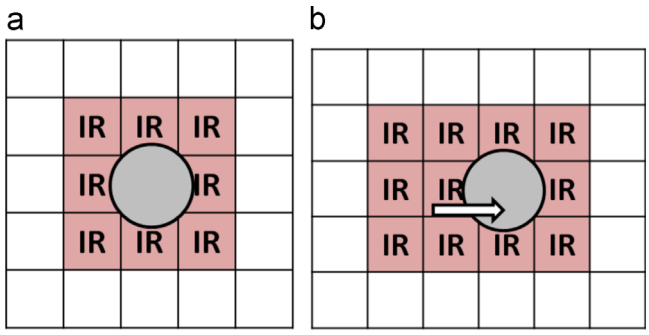


Fig. 12. The interference region of a droplet (a) at rest and (b) moving. If any other droplet enters the interference region, the two will inadvertently merge.

It is important to note that pin mapping can be performed before, during, or after compilation. Performing pin mapping after compilation essentially converts a direct addressing electrode actuation sequence into a pin-constrained electrode actuation sequence; in this case, it is a good idea to verify that the pin mapper does not alter any of the droplet movements; however, we do not enforce this check, because it is irrelevant to pin mappers that are performed before or during compilation.

4.4.7. PCB wire routing

A PCB wire routing solution may use multiple PCB layers. Each PCB layer is represented by the routing graph, and the set of nets that are routed on it. A legal PCB routing solution satisfies the following criteria:

- The set of nets must have a bijective correspondence to the pin mapping solution; each net (p, E_p) must connect a control pin p to the set of electrodes E_p that it drives.
- Each net must be routed on exactly one PCB layer.
- The nets routed on a PCB layer must be disjoint (i.e., shared routing resources between distinct nets is not permitted).
- Net (p, E_p) must route through control pin p and no other control pins, and electrode set E_p and no other electrodes.
- Each PCB layer must contain at least one routed net (although not technically needed to ensure correctness, this does prevent the instantiation of redundant PCB layers).

4.5. Visualization tools

The framework includes an extensive suite of visualization tools written in Java, as shown in Fig. 3. Visualization provides two key capabilities that will assist framework users: debugging, and high quality visual output that can be integrated into papers and presentations. This subsection highlights these capabilities. We use the Polymerase Chain Reaction (PCR) mixing stage as a running example.

4.5.1. Assay visualization

The assay specification (a DAG) is converted into the *.dot* file format, which facilitates display via *GraphViz* [18]. As shown in Fig. 13(a), each node in the DAG is annotated with information, including its operation type (e.g., dispense, mix, etc.), length of duration, and its name (if available). The scheduler computes the start and stop top for each operation in the DAG, and outputs a .dot file, as shown in Fig. 13(b); the scheduler also inserts storage nodes when a droplet is produced by an operation but not consumed immediately. The placer determines the specific location on the DMFB where each assay operation is performed within the (start, stop) timer-interval computed by the

scheduler. The placer adds this information to each node in the DAG and outputs a .dot file, as shown in Fig. 13(c).

4.5.2. Placement visualization

We have implemented Java applications to depict placement in 2- and 3-dimensions. The 2D placement visualizer draws an image of the DMFB, with modules placed, for each time-step. Fig. 14(a) shows an example. The "TS 5" label in the upper left hand corner indicates that this placement occurs at the fifth time-step of the assay. The light blue cells depict two concurrent mixing operations. The dark ovals above each mixer provide information about the two mixing operations with respect to their location in the DAG. The light red cells depict the interference region (IR) [71] of the mixing operations.

Any droplet that inadvertently enters the interference region of an operation will mix with the droplet(s) engaged in the operation, which could result in contamination. Similarly, two operations that overlap with one another at the same time-step and location will cause inadvertent mixing. The visualization tool is thereby useful for debugging placement algorithms. Fig. 14(a) also depicts I/O reservoirs on the periphery of the DMFB, each of which displays the type of fluid it contains; the output reservoir is simply labeled "output." The cells with insignias—fire and magnifying glasses—indicate that external devices that perform heating and detection are available at those locations. The heater is essentially a physical element that is placed above or below the chip. The detector, for example, could be an infrared camera, completely external to the chip, but focused directly on those specific cells.

Fig. 14(b) depicts a 3-dimensional visualization, in which the third dimension (vertical axis) is time. This allows the user to view the scheduled and placed assay operations at each time step, as the assay proceeds. The DMFB is shown at the bottom; a red plane above the DMFB is drawn for each time-step, and time-steps are clearly labeled. Operations that have been placed on the DMFB stretch vertically above the cells that execute them. Modules are labeled with a module number, which can be used to find the corresponding assay operation in the DAG, e.g. Fig. 13(c). The placement rotates so that it can be viewed from all angles; the user can also "fly" through the 3-dimensional space using keyboard controls to view the placement from any desired perspective.

4.5.3. Routing visualization

The Java graphics suite uses two different approaches to display droplet routes. As shown in Fig. 15(a), the cyclic-route view draws an image for each droplet actuation cycle that droplets are in motion. The droplets are numbered, and the light red cells surrounding each droplet represent its interference region (IR).

Similar to modules, two droplets will inadvertently mix if one enters the interference region of another. The droplet color in Fig. 15(a) indicates whether the droplet is free to move forward along its route (green), is waiting for another droplet to move out of its way (yellow), or has reached its destination and has stopped (red). Other colors not shown in Fig. 15(a) indicate droplet merging or I/O operations. The tool draws an image for each actuation cycle of each "*routing phase*" (the droplet routes computed for each timestep in the schedule). For each routing phase, the frames can be compacted into a movie, where each frame equals 10 ms, creating a real-time video for a 100 Hz DMFB.

Lower-end machines, such as netbooks and tablets, may not be able to handle the computational complexity of drawing images for each droplet actuation cycle and creating a movie. Thus, the visualization suite includes a compact-route view, shown in Fig. 15(b), which draws a single image for each route. The electrodes are numbered so that the user can see the exact path the droplet is taking. Although the DMFB may contain multiple droplets at once, only a single droplet's path is drawn in any image; otherwise, the image would be too chaotic and aesthetically displeasing to the human user. Debugging a router using this tool would be tedious; however, it remains a useful tool for visualization on low-end computing devices.

4.5.4. Simulation visualization

The graphics suite has two packages that can display the entire simulation process to the user.

The *cyclic simulator* draws an image for each droplet actuation cycle; it includes all images drawn by the cyclic routing visualizer, as described in the preceding subsection. It also adds images for the cycles between routing phases where assay operations occur; it shows droplets being processed inside the modules, as shown in Fig. 16. The software can stitch the images together to form a movie. This is the

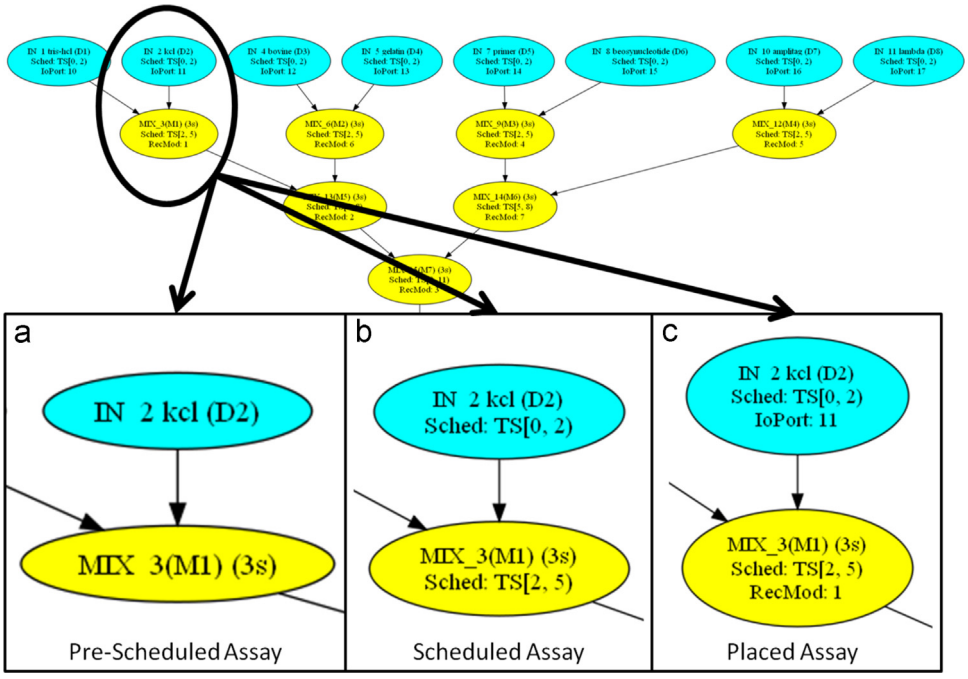


Fig. 13. GraphViz visualization for the assay prior to synthesis (a), scheduled (b), and placed (c). All text is legible in native output files.

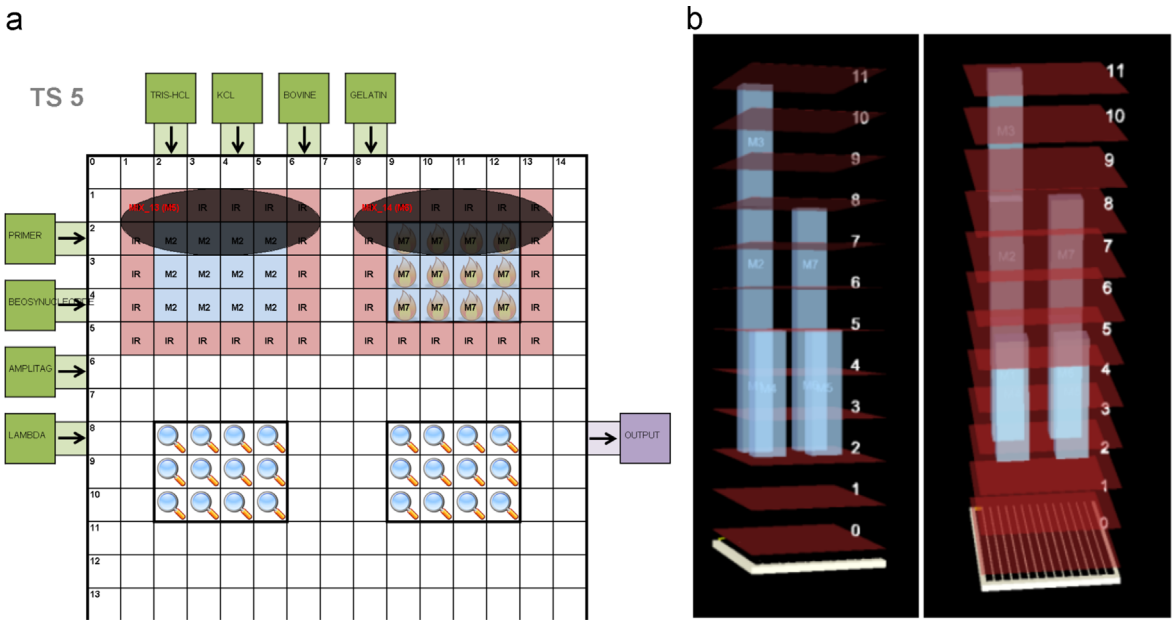


Fig. 14. (a) Sample 2-dimensional placement visualization (the bottom half of the DMFB is clipped for space). All text is legible in native output files. (b) Sample 3-dimensional visualization. (For interpretation of the references to color in this figure, the reader is referred to the web version of this article.)

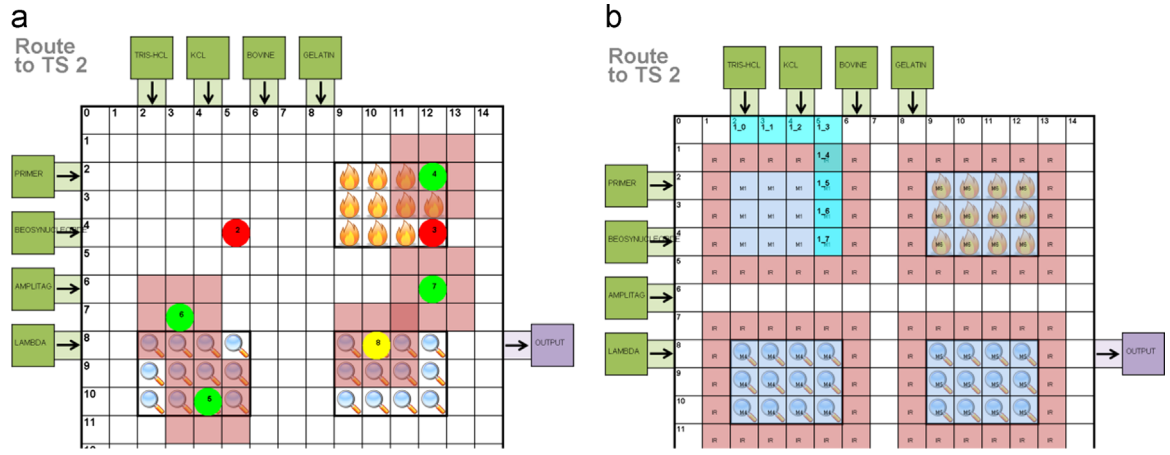


Fig. 15. (a) Cyclic-routing visualization depicting where each droplet is located at a particular droplet actuation cycle. Droplet interference regions are shown in transparent red, while the droplet colors indicate its status (green=free to move; yellow=waiting to avoid droplet interference; red=droplet has reached its destination). All text is legible in native output files. (b) Compact-routing image showing the path a single droplet takes. All text is legible in native output files. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

most complete representation, as all executing operations are visually shown.

The *compact simulator* is designed for low-end machines where the time required to draw images for all droplet actuation cycles may be prohibitive. This simulator interleaves the 2D placement images, e.g., Fig. 14(a), with the compact routing images, e.g., Fig. 15(b). For each routing phase, one image for each droplet is included. This provides a quick and efficient representation of the simulation in progress, but at a coarser granularity of detail than the cyclic simulator.

4.5.5. *Wash droplet routing visualization*

Fig. 17 illustrates wash droplet visualization. Dirty electrodes are filled with a brown color and labeled with the ID of the droplet that contaminated them. Wash droplets (blue; labeled with a 'W') clean contaminated cells, removing the brown shading, and have special I/O ports, also blue with a small image of a broom to represent "cleaning."

5. Experimental comparison

To illustrate the overall utility of our framework, we performed a set of experimental case studies to evaluate the performance of different DMFB synthesis and PCB layout algorithms that have been implemented within the framework. The objective of these studies is to highlight important algorithmic differences, to the greatest extent possible, that have not been reported in prior literature.

5.1. *Benchmarks*

Table 3 lists 10 benchmark DAGs that were used in our experiments, along with their origins. The first four benchmarks, PCR, In vitro 4×4, Protein, and Protein Split 5 have been widely used in past literature on DMFB scheduling [15,22,23,43,59,63,69]; the remaining six are DAGs generated by four of the different sample preparation algorithms listed in Table 1: to the best of our knowledge, this is the first time that DAGs generated by any of these

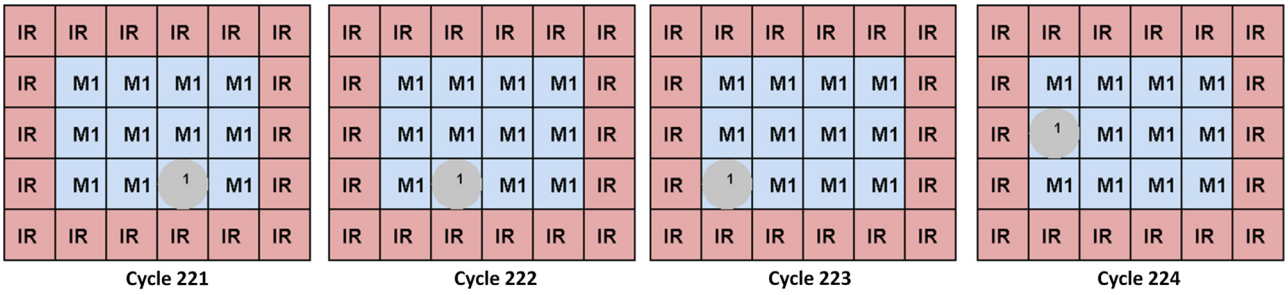


Fig. 16. Illustration of droplet movement within a module.

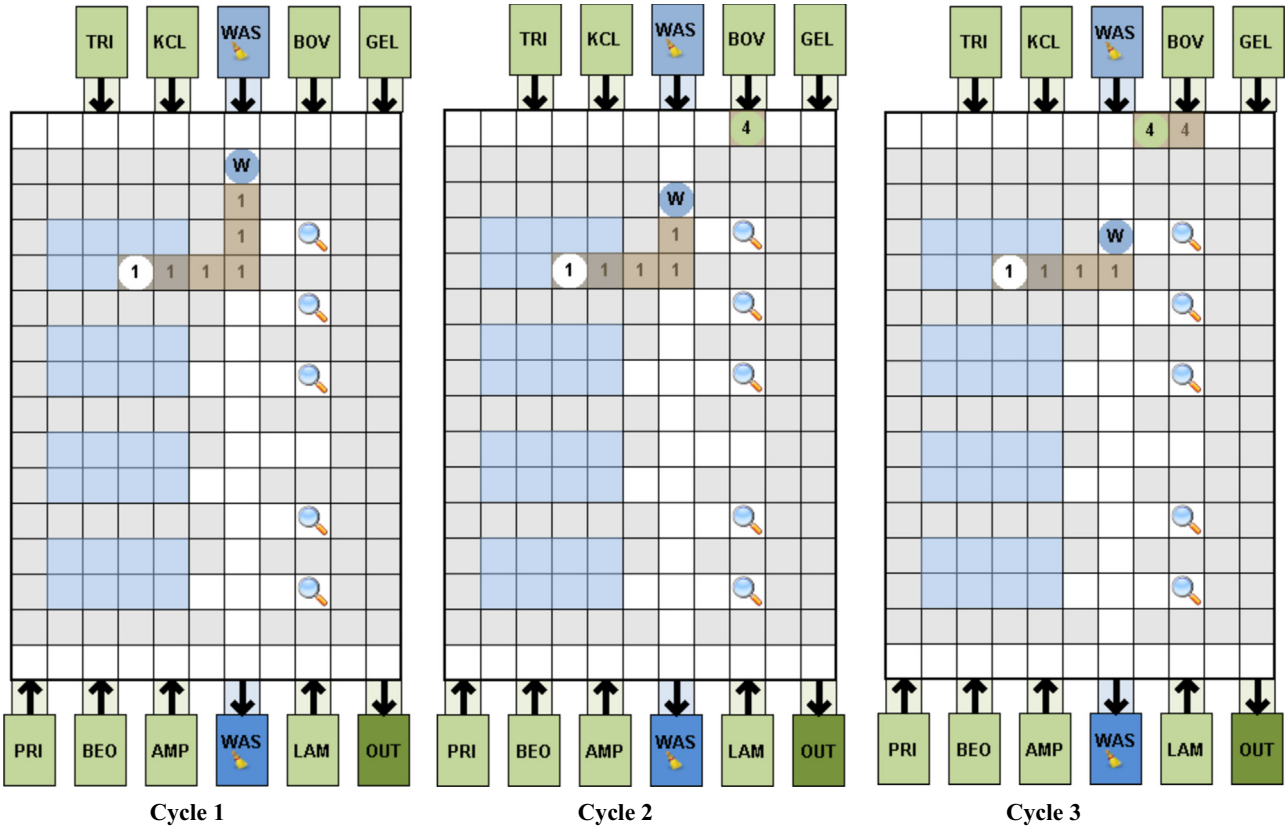


Fig. 17. Three cycles of an assay showing dirty cells (brown cells) marked by the ID of the droplet which made it dirty. Wash droplets (blue, labeled 'W') have specific input and output reservoirs (blue I/O ports with broom labeled 'WAS') and clean dirty cells (removing the brown shading). (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

sample preparation algorithms have been used to evaluate the performance of the scheduling algorithms listed above.

All experiments reported here target a 15×19 DMFB. Experiments were performed on a 2.3 GHz Intel Core i7 processor with 8 GB RAM running 64-bit Windows 7.

5.2. Scheduling results

Fig. 18 reports scheduling results for five different algorithms on the 10 benchmarks listed in Table 3: *List Scheduling* (*LS*) [22,69] *Force-Directed List Scheduling* (*FDLS*) [59], *Path Scheduling* (*PS*) [23], and two *Genetic Algorithms* (*GA*-1, *GA*-2) [63,69]. Fig. 18(a) reports the length of the computed schedules.

PS is most effective when targeting resource-constrained DMFBs relative to the size of the assay that will execute on them [23,59]; PS tries to limit the number of droplets stored on-chip by finding sequences of dependent operations that can be scheduled contiguously. This can be a drawback where there exists ample spatial parallelism, but some operations are delayed to ensure contiguous execution. Another limitation of PS is that it can only schedule trees and forests of trees, so it could not produce results for the sample preparation benchmarks listed in Table 3. LS and FDLS are greedy algorithms that compute priority functions to determine which available vertex to schedule at the current time-step. GA-1 and GA-2 are iterative improvement algorithms also based on LS. They randomly rearrange the priorities of the vertices relative to one another; after each perturbation, they call LS to compute a new schedule with the updated priorities.

The only benchmark in this experiment that is resource constrained is Protein Split 5. Only LS and PS find legal schedules, and the schedule computed by PS is approximately 35% shorter, benefiting significantly from superior spatial resource management. In contrast, PS computes inferior schedules for the small benchmarks such as In Vitro 4×4 and Protein, which can easily fit onto the chip.

LS computes a shorter schedule than FDLS for only one benchmark, Protein, suggesting than FDLS is perhaps the most effective heuristic in situations where resource constraints are not stringent. GA-1 and GA-2 often obtain marginal improvements over LS

Table 3
Benchmark DAGs used for the experiments reported in this section.

Benchmark	Description
PCR	PCR Mixing Tree [15]
In Vitro 4×4	Multiplexed in vitro diagnostics with 4 samples and 4 reagents [69]
Protein	Protein interpolating serial dilution assay [69]
Protein Split 5	5-level generalization of the Protein assay [23]
BIN (13/128)	Single droplet with target CV (13/128) using the BIN algorithm [19]
REMIA (191/1024)	Single droplet with target CV (191/1024) using the REMIA algorithm [29]
REMIA (641/1024)	Single droplet with target CV (641/1024) using the REMIA algorithm [29]
REMIA (850/1024)	Single droplet with target CV (850/1024) using the REMIA algorithm [29]
GORMA (23/256)	Single droplet with target CV (23/256) using the GORMA algorithm [11]
CoDOS $<2, 45, 23, 67, 93>$	Single droplet from 5 reactants (R1 - R5) composed as follows from CoDOS [42] 2 parts R1 45 parts R2 23 parts R3 67 parts R4 93 parts R5

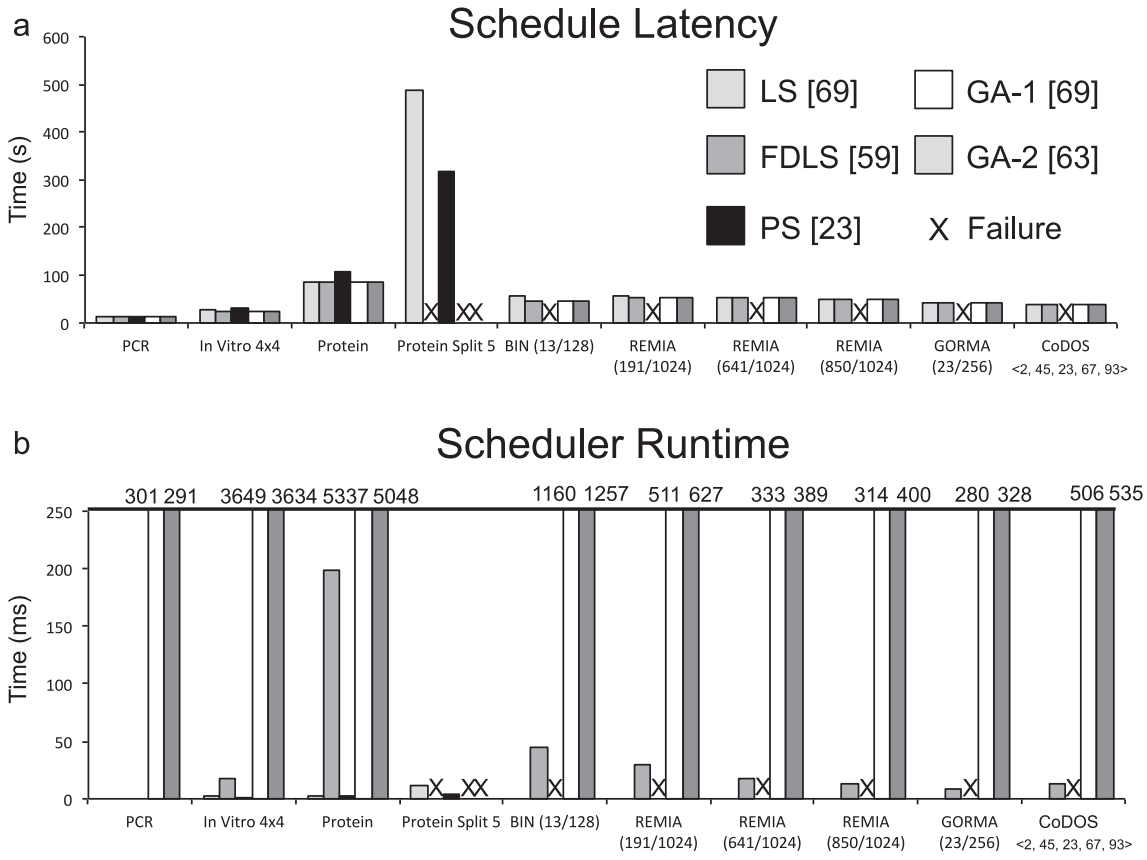


Fig. 18. Comparison of five scheduling algorithms for the 10 benchmark DAGs in Table 3: (a) schedule lengths and (b) scheduler runtimes.

and FDLS (except for Protein Split 5), due to their iterative improvement nature: LS and FDLS compute one schedule each, while GA-1 and GA-2 compute a large number of randomly-generated schedules and choose the best.

Fig. 18(b) reports the runtime of the scheduling algorithms. The fastest algorithm is PS, because it makes scheduling decisions on the granularity of paths, not vertices. FDLS is slower than LS because it requires a pre-processing step to compute vertex priorities, which is not required for LS. GA-1 and GA-2 run several orders of magnitude slower than the other three, which is expected for iterative improvement heuristics; in practice, the runtime can be tuned indirectly by varying a collection of user-specified parameters.

In all subsequent experiments, we use the LS scheduling result for all benchmarks except for Protein Split 5. For Protein Split 5 we use the PS scheduling result instead, due to its dramatically superior quality in comparison to the schedule produced by LS.

5.3. Placement results

Fig. 19 reports placement results for the *KAMER* [2,44] and *Virtual Topology* (*VT*) [21,22] placement heuristics depicted in Fig. 11(a) and (b). VT finds legal placements for all ten benchmarks, while KAMER finds legal placements for seven of them, echoing previously reported results [22]. VT clearly articulates the available on-chip resources to the scheduler; with KAMER, the scheduler

must estimate which operations may execute concurrently but cannot assess the legality of the resulting placement a-priori.

The metric for comparison reported in Fig. 19 is the total number of DMFB cells allocated at least once by each placer. This metric is a proxy for space utilization, but does not directly correlate with performance. VT achieves more efficient space utilization than KAMER for five of the seven benchmarks that KAMER could legally place. KAMER, which is based on a reconfigurable computing paradigm, typically begins with an orderly placement similar to VT; however, as operations start and stop at later scheduling time steps, fragmentation may occur, which negative impacts utilization; in the worst case, fragmentation may lead to failures as it becomes more challenging to place scheduled operations in a highly fragmented spatial 2D plane.

In terms of performance, the best placements yield the shortest droplet routes; however, routing latencies cannot be characterized until after both placement and routing complete. We discuss this interplay in detail in the following subsection.

The runtime of both placers was at most 1 ms for all benchmarks; detailed results are not reported here, as the scheduling and routing algorithms dominate runtime.

5.4. Routing results

For each benchmark, we compare four different routing algorithm configurations using placements produced by KAMER and VT (eight data points, per benchmark). Two configurations are based on a Maze Routing algorithm described by Roy et al. [64]. The Maze Router uses Soukup's Algorithm to compute the path that each droplet will take, under the assumption that droplets are routed one-at-a-time. A post-processing step called "compaction" is then performed to enable concurrent droplet routing while preventing interference between droplets. Roy's implementation uses a greedy compaction algorithm, a configuration that we refer to as *Maze-Greedy*. Huang et al. [32] introduced a more intricate compaction algorithm based on dynamic programming; however, their paper only describes how to compact two droplet routes. In our implementation, if k droplets have been compacted, we must solve k dynamic programming problem instances in-sequence to compact the $(k+1)$st droplet; we call this configuration *Maze-DP*.

We also implemented a *High-Performance* (*HP*) droplet router introduced by Cho and Pan [12]. HP prioritizes droplets for routes based on the concepts of bypassability and concession. Bypassability estimates the ability of a droplet to route around other droplets in the chip, which may block it; concession is used to break deadlocks, where some droplets back off to allow other droplets to proceed. Similar to the Maze Router, route computation is followed by compaction. We have implemented two versions of HP: one using the same greedy compaction algorithm as the Maze Router (*HP-Greedy*), and one using a compaction step described in Cho and Pan's paper (*HP–HP*).

Fig. 20 reports the aggregate latencies computed by the three routers for all droplet routing sub-problems. The choice of placer (KAMER or VT) has a much greater impact on droplet routing latency than the choice of droplet routing algorithm: in all cases where KAMER found a legal placement, VT yielded lower latency droplet routes than KAMER, regardless of which router was used. The best performing router was Maze-Greedy, followed by Maze-DP, HP-Greedy, and HP–HP, although there are a few exceptions (e.g., Protein under VT, where HP-Greedy yielded the shortest routes). The greedy compaction algorithm, in most cases, yielded shorter routes than the DP or HP compactors, regardless of whether Maze or HP routing was used. DP's weakness is the need to perform multiple compactions against all of the droplets undergoing transport, while the HP compactor sacrifices route lengths in order to improve fault tolerance by utilizing fewer cells.

Fig. 21 reports the runtime of the different routing algorithms. Across all benchmarks, Maze-Greedy is the fastest running routing algorithm while HP–HP is the slowest. In almost all cases, each of the four routing algorithms runs faster under VT than KAMER.

Comparing the results reported in Figs. 20 and 21, Maze-Greedy is generally the best performing and fastest converging routing algorithm among the four evaluated here

The Maze-Greedy router is used in all subsequent experiments.

5.5. Pin mapping and wire routing results

We compare three different pin mapping and PCB routing algorithm combinations that have been previously published by others. We report the number of PCB layers and the number of control pins required to realize each chip; as reported by Grissom et al. [25], these two factors are the primary drivers for total chip cost, and the number of PCB layers plays a more significant role.

The *RAU-Aware* pin mapper [36] attempts to minimize the number of *Redundant Activation Units* (*RAUs*). An RAU is the activation of an electrode (due to pin sharing) that does not directly act on a droplet, thereby expending excess power. Under pin sharing, some RAUs are unavoidable, but power savings can be achieved in RAUs are taken into account. After pin mapping, the PCB wire route is computed using an algorithm proposed by McDaniel et al. [52].

The *GV-Aware* pin mapper [34] judiciously inserts *ground vectors* (*GVs*) to mitigate the impact of trapped charge [3,16,74] and residual charge [61], both of which negatively affect reliability and degrade

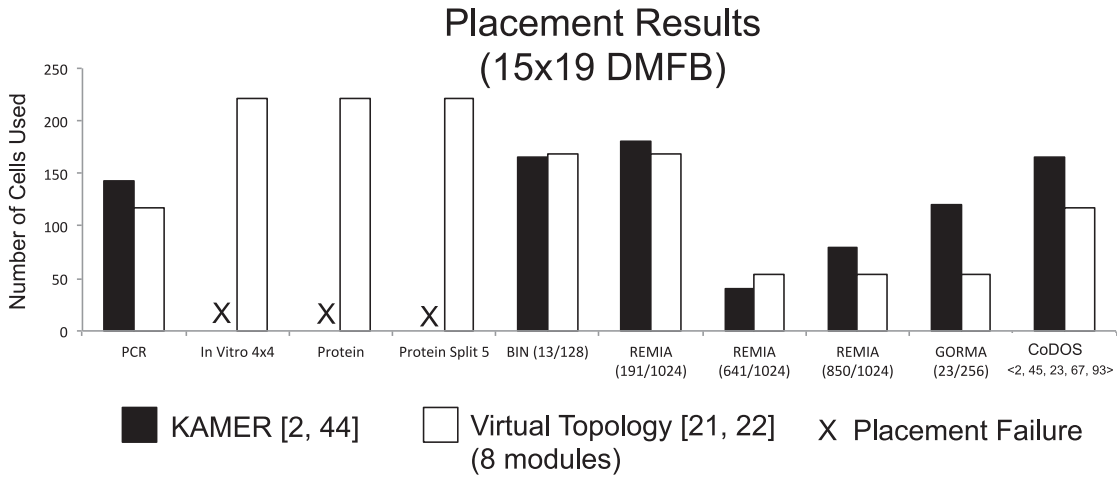


Fig. 19. Comparison of the KAMER [2,44] and Virtual Topology (VT) [21,22] placers on the 10 benchmark DAGs in Table 3.

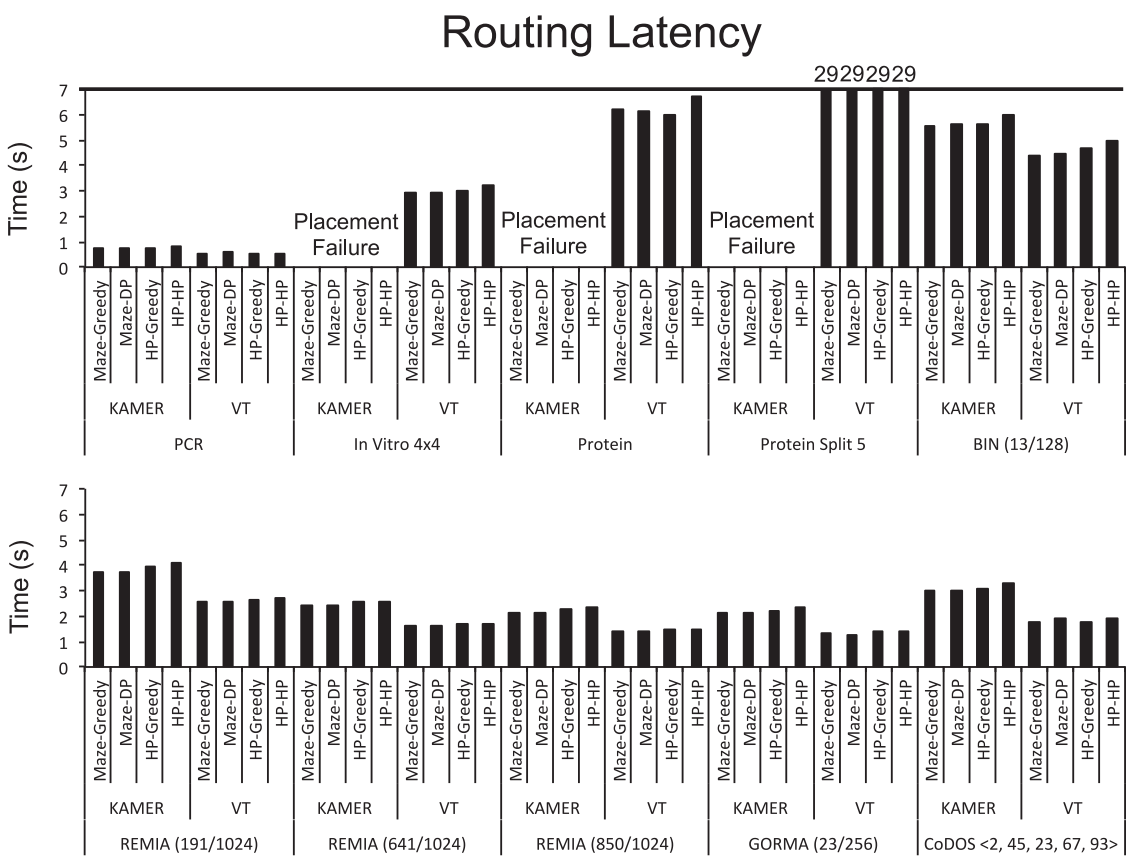


Fig. 20. Performance comparison of the Maze-Greedy [64], Maze-DP [32,64], HP-Greedy [12,64], and HP–HP [12] routing algorithms across the 10 benchmarks from Table 3 using the KAMER and Virtual Topology (VT) placement heuristics.

performance. Inserting a GV quickly deactivates an electrode that would otherwise be charged, thus reducing the accumulation of charge over time. After pin mapping, the PCB wire route is computed using an algorithm proposed by McDaniel et al. [52].

The *Toggle-Aware* pin mapper [80] tries to reduce the number of times each electrode is *toggled* on/off, which reduces the likelihood of electrolysis [31]. Unlike the other two pin mappers, Toggle-Aware performs PCB wire routing incrementally as an integrated part of the algorithm. In principle, it can call any incremental wire routing algorithm as a subroutine, as needed. Our implementation calls an incremental extension to the PCB wire routing algorithm proposed by McDaniel et al. [52].

RAU-Aware and GV-Aware employ a progressive addressing scheme in which subsets of yet-unaddressed electrodes are repeatedly selected and grouped with others to implement control pin sharing; both algorithms model the pin count expansion using a network flow problem to limit the growth in the number of control pins while reducing power consumption or increasing reliability as needed. Toggle-Aware takes a different approach, in which one unaddressed electrode is added at a time; moreover, the switching constraint is progressively relaxed, permitting the algorithm to discard unsuccessful electrode groupings and start over. Thus, Toggle-Aware is noticeably different than RAU-Aware and GV-Aware in terms of algorithmic structure; this difference may (or may not) account for the disparities in experimental results reported here.

Toggle-Aware, as described in Ref. [80], does not admit PCB routing solutions with multiple layers. Our implementation relaxes this assumption, which enables more extensive pin sharing. Since RAU-Aware and GV-Aware are not cognizant of the number of PCB layers, this approach represents the fairest possible comparison.

5.5.1. Number of PCB layers

Fig. 22 reports the number of PCB layers obtained by each algorithm using both the KAMER and VT placers. The most significant result shown in Fig. 22 is that the choice of placer often has a greater impact on the number of PCB layers than the choice of pin mapper: in all cases, the VT placer yielded fewer PCB layers than KAMER, regardless of which pin mapper was used. Among the different pin mappers, there are no clear trends. For example, GV-Aware yields the fewest PCB layers for In Vitro 4×4 under VT, and the most for BIN (13/128) under VT. In principle, more conservative pin sharing [37] could further reduce the number of PCB layers.

5.5.2. Pin sharing

Fig. 23 reports the number of control pins obtained through pin sharing by each algorithm using the KAMER and VT placers. Similar to Fig. 22, the VT placer uniformly reduces the number of control pins, although the reduction is truly dramatic for just one benchmark: BIN (13/128). Among the three pin mappers, there are no uniform trends, although Toggle-Aware has a general tendency to require a few more control pins than the others. RAU-Aware and GV-Aware require the same number of control pins in most cases, although there are a few cases whether either RAU-Aware or GV-Aware requires fewer pins.

5.5.3. Redundant activations

Fig. 24 reports the number of RAUs obtained through pin sharing by each algorithm using the KAMER and VT placers. In most of the experiments Toggle-Aware yields considerably more RAUs than either RAU-Aware or GV-Aware. Although the choice of placer does not have a significant effect on the total number of RAUs, there are no uniform trends, i.e., sometimes KAMER is better

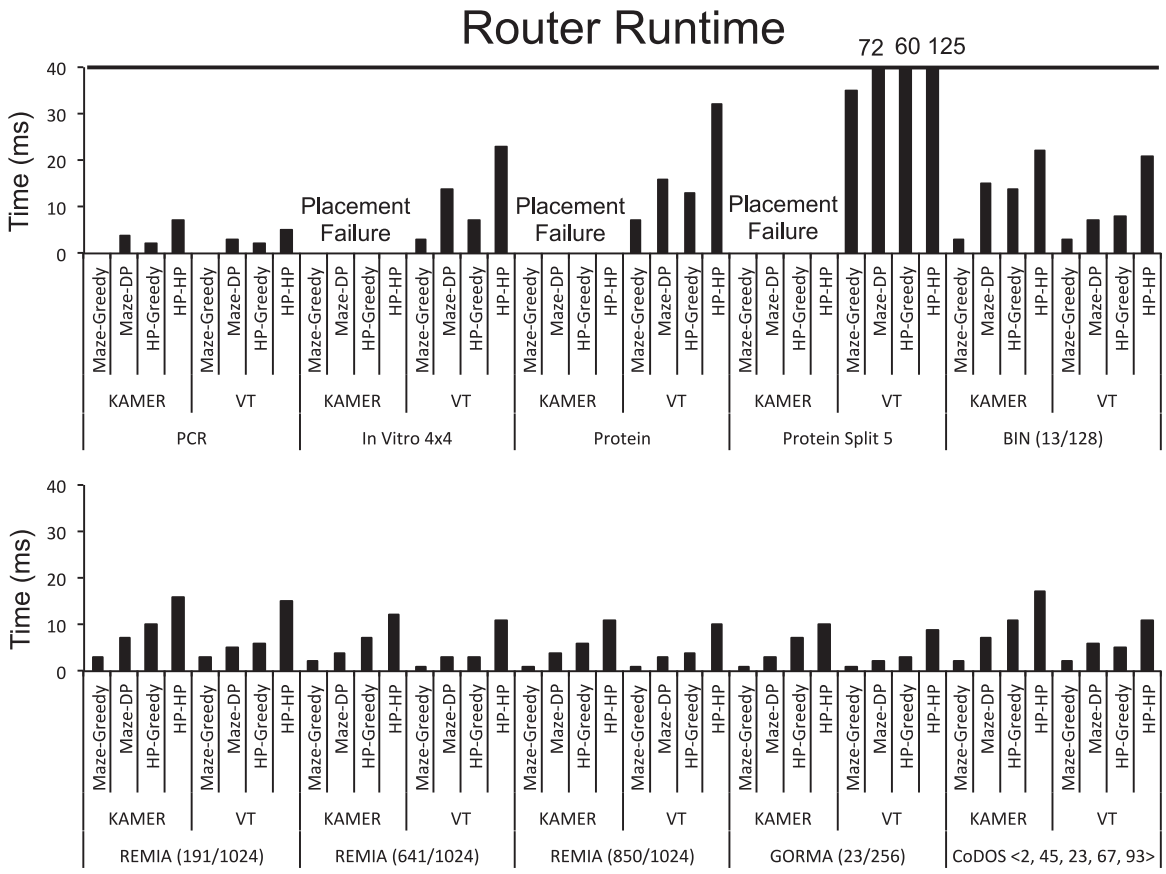


Fig. 21. Runtimes of the Maze-Greedy [64], Maze-DP [32,64], HP-Greedy [12,64], and HP–HP [12] routing algorithms across the 10 benchmarks from Table 3 using the KAMER and Virtual Topology (VT) placement heuristics.

than VT, and sometimes VT is better than KAMER. The most notable result obtained from examining these results is that GV-Aware appears to be just as good as RAU-Aware in terms of RAU minimization, despite that the later was designed precisely to solve this problem.

5.5.4. *Switching activity*

Fig. 25 reports the amount of switching activity (electrodes toggling on/off) observed through pin sharing by each algorithm using the KAMER and VT placers. The amount of switching activity seems to be near uniform across most system configurations for most benchmarks. In other words, Toggle-Aware does not seem to offer any advantages over RAU-Aware and GV-Aware in terms of reducing switching activity, despite its intended objective; additionally, the effects of different placement results on switching activity appears to be negligible.

5.5.5. *Ground vectors*

Fig. 26 reports the number of ground vectors (GVs) inserted as part of pin sharing by each pin mapper using the KAMER and VT placers. Unsurprisingly, GV-Aware tended to reduce the number of GVs inserted compared to RAU-Aware and Toggle-Aware. Using KAMER for placement in conjunction with GV-Aware pin mapping led to fewer GVs inserted compared to using VT for placement in all but two cases, BIN (13/128) and GORMA (23/256).

5.5.6. *Discussion*

Absent pin sharing, the operations most likely to necessitate the insertion of GVs are those that activate an electrode for many consecutive time-steps, i.e., storage and any operation that activates an external device for an extended period of time. Without loss of generality, electrodes involved in mixing operations cannot share the same pin as electrodes involved in storage operations that are scheduled at the same time. The electrodes that perform mixing repeatedly switch on and off, while the electrodes that perform storage remain activated (before GV insertion): a pair of electrodes that require simultaneous activation and deactivation cannot share the same pin.

KAMER and VT minimally consider the operations that they place and neither tries to optimize for pin sharing or power/reliability when making decisions. KAMER places operations that use external devices on regions of the chip that are accessible to those devices; beyond that, its objective is simply to obtain a legal placement at each time-step. VT also accounts for external devices and considers whether each work module is performing mixing or storage at a given time-step. VT may transport a droplet from one module to another during storage to free up the former module for another operation. VT makes these decisions this without considering the implications for pin sharing and power/reliability insertion. Thus, it is difficult to ascertain whether the differences between VT and KAMER reported in Figs. 24–26 are inherent to the placement algorithms, or essentially just random effects. To better answer this question, future work should focus on synergistically optimizing scheduling and placement with pin sharing while considering power reduction and/or reliability.

6. Device interface and control

Fig. 27 depicts the device interface and control software. An application written in Java runs on a host PC; it communicates

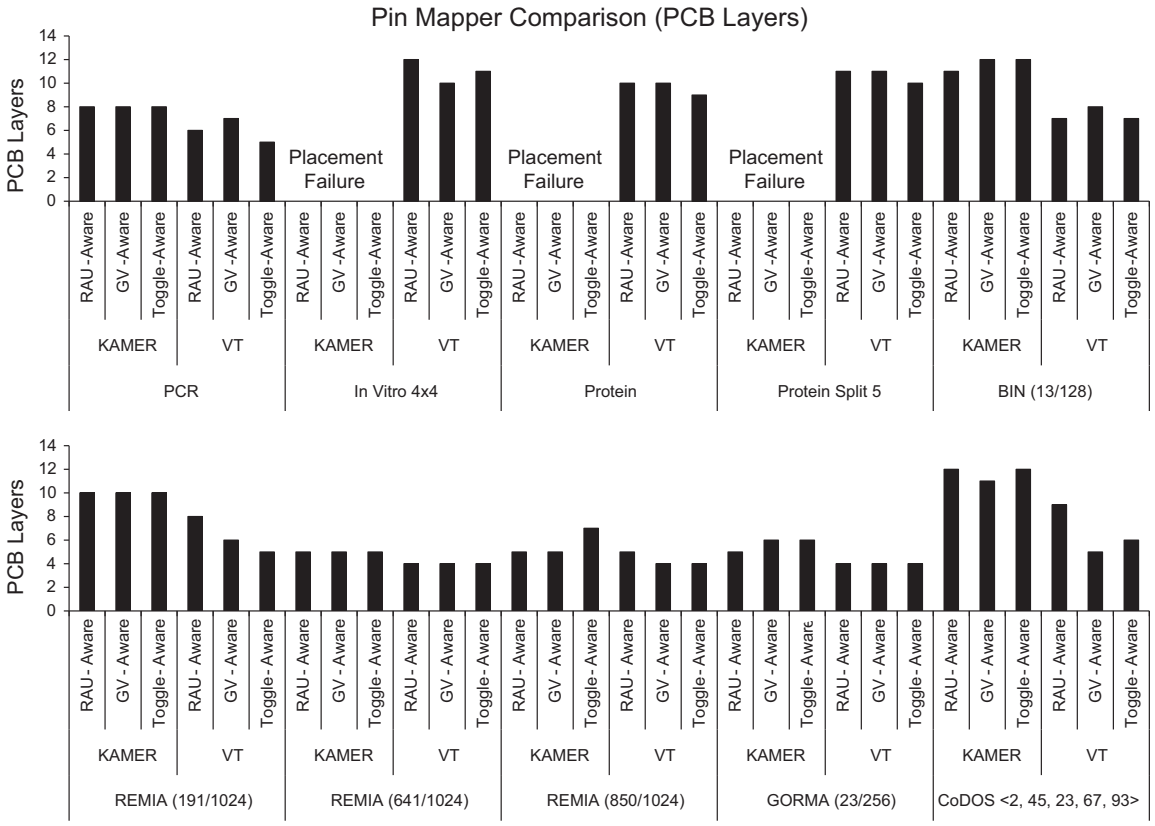


Fig. 22. The number of PCB layers obtained using the RAU-Aware [36], GV-Aware [34], and Toggle-Aware [80] pin mappers, with KAMER and VT placers.

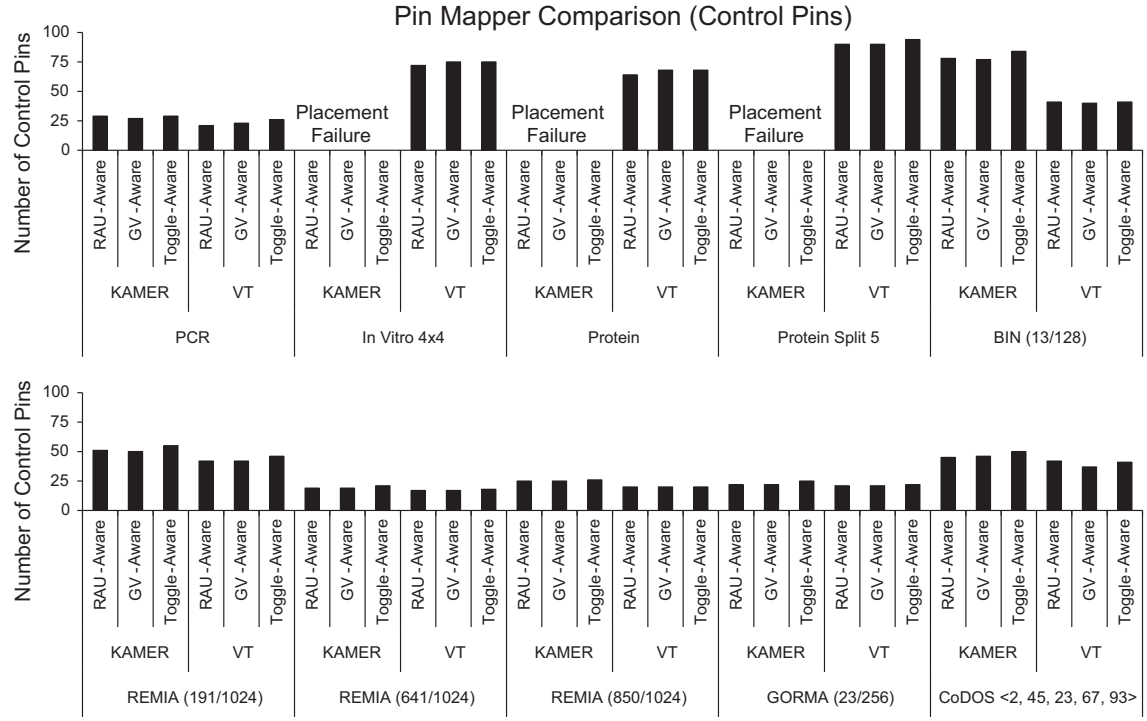


Fig. 23. The number of control pins obtained using the RAU-Aware [36], GV-Aware [34], and Toggle-Aware [80] pin mappers, with KAMER and VT placers.

with an ATmega2560 microcontroller, which interfaces directly with the DMFB. The current version of the system can control DMFBs based on the paradigm of direct [62] or active-matrix [27,58] addressing, both of which provide independent control over each electrode; the active-matrix is implemented using thin film transistors fabricated on a glass substrate, similar to liquid crystal displays. Although the control interface to an active-matrix DMFB is considerably different than to a direct-addressing chip, the fundamental property of individually addressable electrodes ensures that the same algorithms for scheduling, placement, and

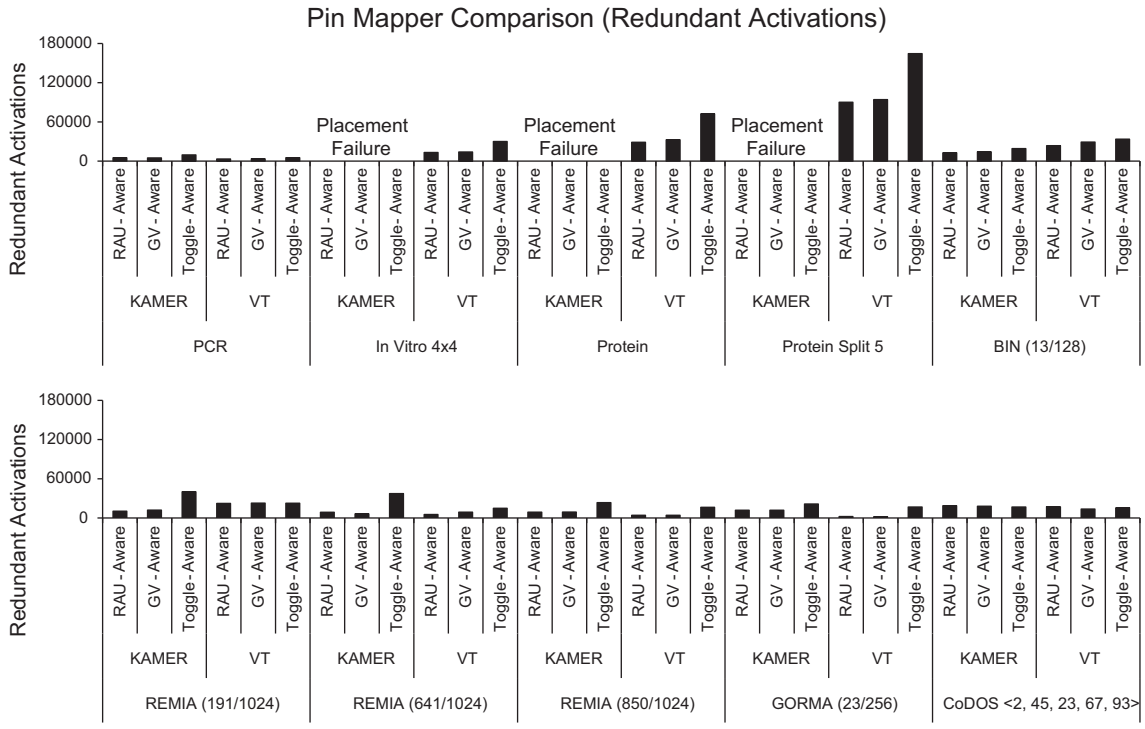


Fig. 24. The number of redundant activations (RAs) observed when using the RAU-Aware [36], GV-Aware [34], and Toggle-Aware [80] pin mappers, in conjunction with the KAMER and VT placers.

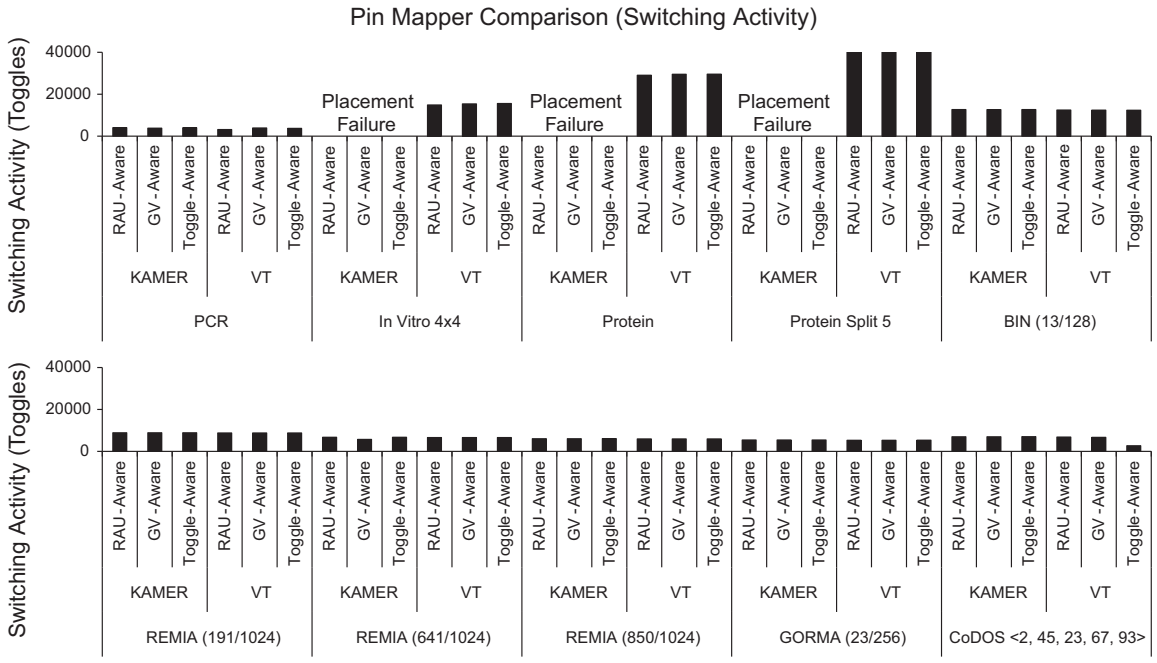


Fig. 25. The switching activities (toggles) observed when using the RAU-Aware [36], GV-Aware [34], and Toggle-Aware [80] pin mappers, in conjunction with the KAMER and VT placers.

routing can be used in a compiler that targets both devices. The following subsections describe the Java control program and the embedded control software that runs on the ATmega2560.

6.1. DMFB control software

Fig. 28 shows a screenshot of the Java application that runs on the host PC. An architecture description file, describing the array dimensions, DMFB driving technology (direct or active-matrix addressing), and microcontroller pin map (e.g., column 0 connects to general purpose I/O pin 23 on the microcontroller) initializes the program.

The control software permits either manual or automatic control of the device. For instance, the user may click on electrodes in a visual depiction of the device to activate and de-active them (in Fig. 28 yellow electrodes are selected by the user for activation during the next cycle; green electrodes are activated during the current cycle). Multiple electrodes can be selected at once by

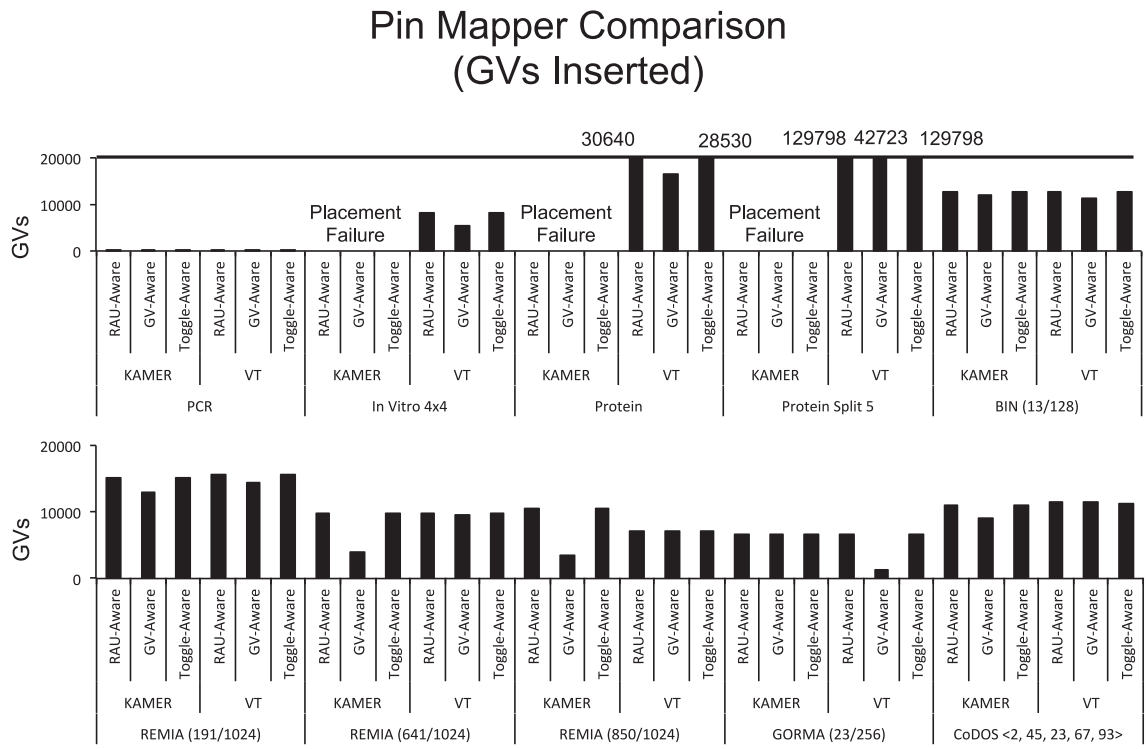


Fig. 26. The number of GVs inserted by the RAU-Aware [36], GV-Aware [34], and Toggle-Aware [80] pin mappers, in conjunction with the KAMER and VT placers.

grouping them. If one electrode is activated, the keyboard arrow keys can move the activated electrode to one of its four adjacent neighbors (e.g., this feature could manually transport a droplet). As the user "enters" new electrode activations, the sequence is recorded and saved for playback.

The control software can also load and play electrode activation sequences generated by the compiler. During playback, the cycle length (in ms) can be adjusted to change the electrode activation time. For active-matrix devices, the length of the gate pulse (in μs), i.e., the time to charge the capacitor, can be varied to adjust the refresh rate in response to physical parameters that dictate the charge retention time.

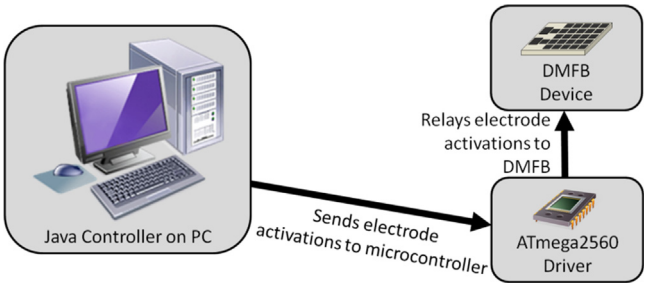


Fig. 27. Control software interface. An application written in Java and running on the host PC transmits a sequence of bit vectors representing electrode actuations to an ATmega2560 microcontroller, which interfaces directly with the DMFB.

6.2. Microcontroller software and DMFB interface

The host PC control software interfaces with the microcontroller via a serial communication protocol over a USB cable. When the board is controlled manually, the set of electrodes to be activated is sent when the user hits the "Enter" key, as described in the previous subsection. The microcontroller holds this status until another electrode pattern is received. During playback, the host PC can only transmit a portion of the overall electrode activation sequence at a time; the number of cycles that can be sent in a single bulk transfer depends on the number of electrodes and the microcontroller's memory capacity. The microcontroller plays the received subsequence at the desired frequency (as specified by the configuration in the host PC control software). When it completes, it sends a message to the host PC to ask for the next subsequence to play back. This process repeats until the activation sequence is played back in full. The serial connection between the host PC and microcontroller is 115,200 bits per second (millisecond scale), sufficient for transmission of the next state before the activation time of the current state completes.

To support a direct-addressing DMFB, each pin is set to ON or OFF and then left alone until the next bit-vector in the sequence is processed. Active-matrix DMFBs are controlled like LCD screens and require a continuous refresh. An active matrix DMFB has a thin film transistor under each electrode, which acts as a capacitor. Rows are selected one-by-one; while each row is selected, the driver steps through the columns and charges the capacitor at the row-column intersections if the electrode requires activation. Without refresh, the capacitors will dissipate their charge/state.

6.3. State machine controller

The size of a state machine is $T \times C$, where T is the number of time-steps (at the routing granularity), and C is the number of cells in the chip. The uncompressed storage requirement can be quite high [48]. By compressing the state machine, it is sometimes possible to reduce its size so that it can fit directly onto the microcontroller. Once the state machine is loaded into the microcontroller's memory, the host PC is no longer needed to execute the assay, thereby reducing cost and increasing overall portability.

Fig. 29 depicts the bit-vector sequence required for the first six states of a 2×2 mixer. The mixing process is actually a circular repetition of four electrodes activated one at a time. Without any compression, this example requires six bit vectors, each 4 bits in size; after the first four states, all subsequent bit vectors are redundant until the mixing finishes; in this example, two are

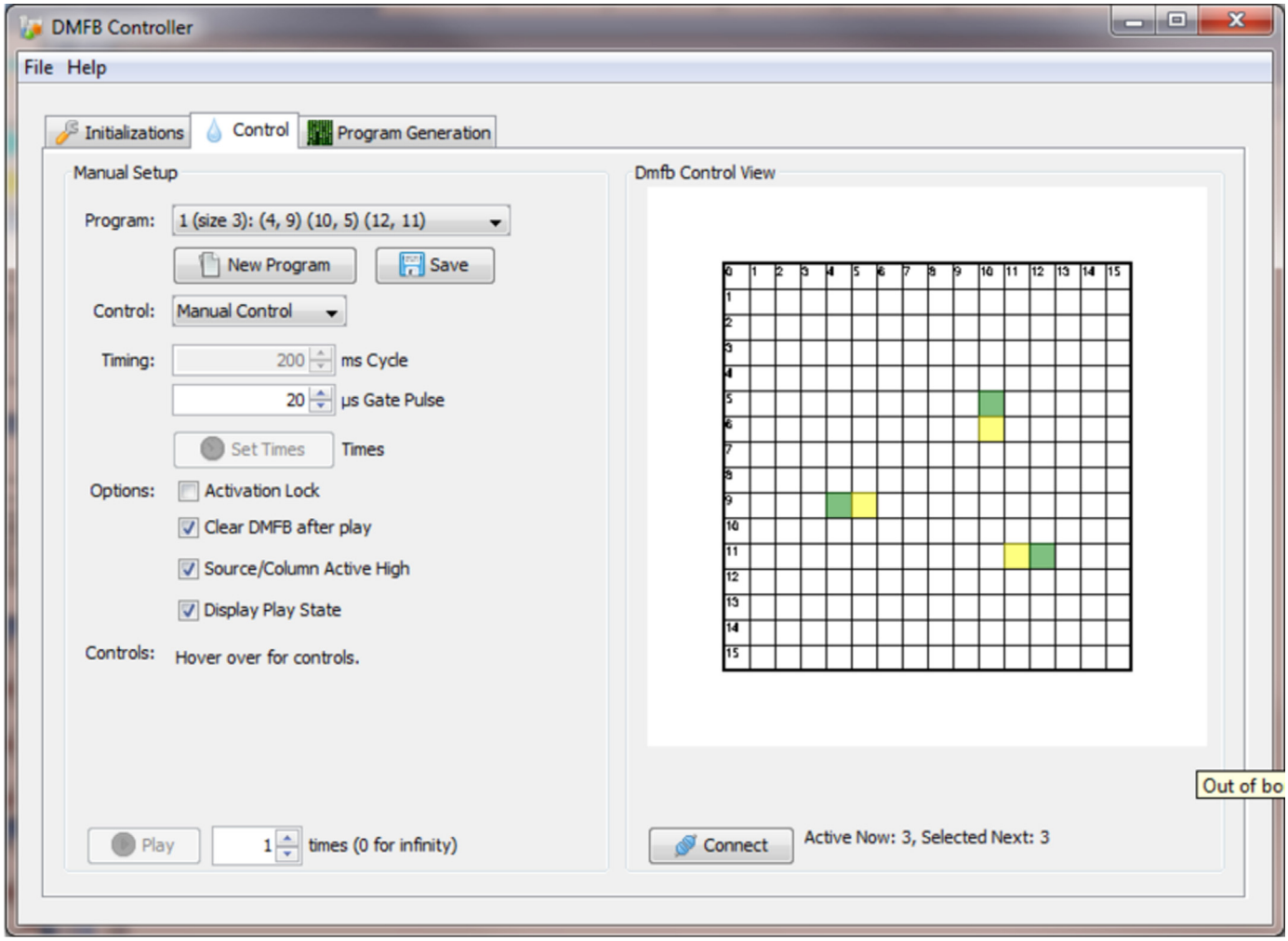


Fig. 28. Screenshot of the DMFB control software. The user can create and save manually-designed electrode activation sequences by clicking on cells in the "DMFB Control View" shown on the right; green cells are currently activated cells, while yellow represents cells that the user has selected for activation during the next cycle. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

redundant. Therefore, it is much more space efficient to put the four repeating bit vectors into a *dictionary D*, and have each state output a two-bit dictionary index *j*. The control software accesses the *j*th dictionary entry, $D[j]$, which is a bit-vector representation of the electrodes to activate at the next time-step.

There is also redundancy in the repeated cycle of four states, which is essentially an unrolled loop. A state machine minimization algorithm could identify this redundancy and convert the linear sequence to a four-state loop. This reduces the number of unique states, each of which requires a unique identifier, in the DMFB control program; our framework automatically generates the DMFB control program in the C language.

6.3.1. Dictionary generation

The first step is to identify redundant bit vectors, i.e., Moore Machine state outputs. There is no need to understand context when examining a state, i.e., whether the state is part of an operational or routing phase is irrelevant.

The process of dictionary generation is straightforward. The dictionary is represented as a vector of bit-vectors, *D*, and a hash table *H* is used to improve performance. Let S_i denote the *i*th state in the linear Moore Machine, and let B_i denote the corresponding bit-vector. Initially, both *D* and *H* are empty.

States can be processed in any order. When processing state S_i, we compute a hash table lookup $H(B_i)$, to determine if B_i is redundant, i.e., it has been encountered already. If B_i is redundant, then the result of $H(B_i)$ is a tuple (B_i, j), indicating that $D[j]=B_i$, i.e., B_i is stored in the *j*th entry of dictionary *D*. The Moore Machine output of state S_i is changed from B_i to *j*, which indicates a dictionary lookup. Since *D* is incomplete until all states have been examined, the number of bits required to express the value *j* is not yet known.

If B_i is not redundant, then B_i is appended to *D*, i.e., if $|D|$ is the current number of entries in *D*, then B_i is inserted into dictionary entry $D[|D|+1]$, and the tuple $(B_i, |D|+1)$ is inserted into *H*. This way, if B_i is encountered again, the hash table will return the correct dictionary index for B_i. After the dictionary is built, the number of bits required for each dictionary index *j* (Moore Machine state output) is $\log_2|D|$.

6.3.2. Reducing the number of States

To reduce the number of states, each operational stage is converted into a loop, while routing stages (which presumably contain no repetitions) retain the linear structure of the original state machine. The boundaries between operational stages and routing stages are determined from the scheduling, placement, and routing solutions; there is no need to extract this information from a bit-vector sequence.

The algorithm to convert each operational stage into a loop is simple. Let there be *n* concurrent operations, which may include

mixing (modules may have different sizes), storage, or usage of an external device; these operations are naturally repetitive. Let c_i be the number of cycles required for one iteration of the ith operation o_i. For example, if o_i is a 2×3 mixer, then $c_i = 6$; if o_i is storage or usage of an external device, then $c_i = 1$. The number of states in the loop is the least common multiple of all of the c_i values. For example, if operations o_1 and o_2 have $c_1 = 4$ (e.g., a 2×2 mixer) and $c_2 = 3$ (e.g., a 2×3 mixer), then the loop will have 12 states. Within one iteration of the loop, o_1 will complete three cycles and o_2 will complete four cycles. This ensures that all droplets have the same starting point at the beginning of the loop in the state machine.

6.3.3. Microcontroller implementation

The ATmega2560 microcontroller has 256 KB of programmable flash memory called Program Memory and 8 KB of SRAM called Data Memory. The Program Memory stores the dictionaries and index arrays used for compression. The Data Memory stores the state machine control code and local variables.

6.3.4. Experiments

We considered 10 assays commonly used in peer-reviewed publications on programmable microfluidics: one PCR mixing tree, five multiplexed in vitro diagnostics assays with a varying number of samples and reagents ("in Vitro xs_yr" means x samples and y reagents), and four exponential protein dilution assays with a varying number of splits. We compiled the assays using two synthesis flows with two different placers: KAMER (Fig. 11(a)) [2,44], and a virtual topology (VT) (Fig. 11(b)) [21,22]; we used the same scheduling and routing algorithms in both experiments. We expected that the VT placer would yield smaller state machines, because all mixing operations would have identical geometries and occur in the exact same set of restricted locations on the chip, while KAMER can place any operation at any location.

First, we compiled each assay into a linear state machine using both synthesis flows, similar in principle to Fig. 4; *the memory requirement of every single assay exceeded the capacity of the ATmega2560*. Next, we compressed the state machines using dictionary compression only (D) and dictionary compression in conjunction with state machine optimization (D+SM). Fig. 30(a) and (b) reports the program and data sizes for the two respective synthesis flows. The C programs generated by our synthesis flow are converted to .elf files, and the results shown in Fig. 30 report the.elf file size in KB.

Fig. 30 shows that compression fails for two benchmarks (Protein and Protein Split 3) with KAMER, but succeeds for all benchmarks for VT. The general trend is that program and data segment sizes are smaller for VT than for KAMER.

Compression using the dictionary and state machine optimization (D+SM) yielded lower overall memory requirements than dictionary compression alone (D). Luo et al. [48], it should be noted, achieved even greater reductions in dictionary size using run length encoding; however, the decoding process is complex,

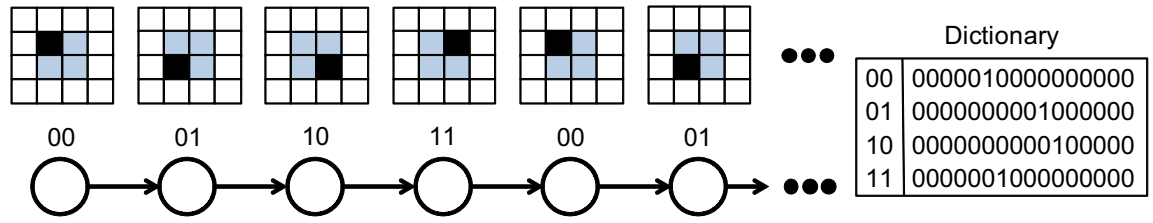


Fig. 29. Illustration of dictionary compression: redundant bit vectors are identified and put into a dictionary. Each state's output is now an index into the dictionary. The original Moore Machine had $6 \times 16 = 98$ output bits; the compressed Moore Machine has a dictionary consisting of $4 \times 16 = 64$ bits, while the state machine requires $6 \times 2 = 12$ output bits, for a total of 76 bits.

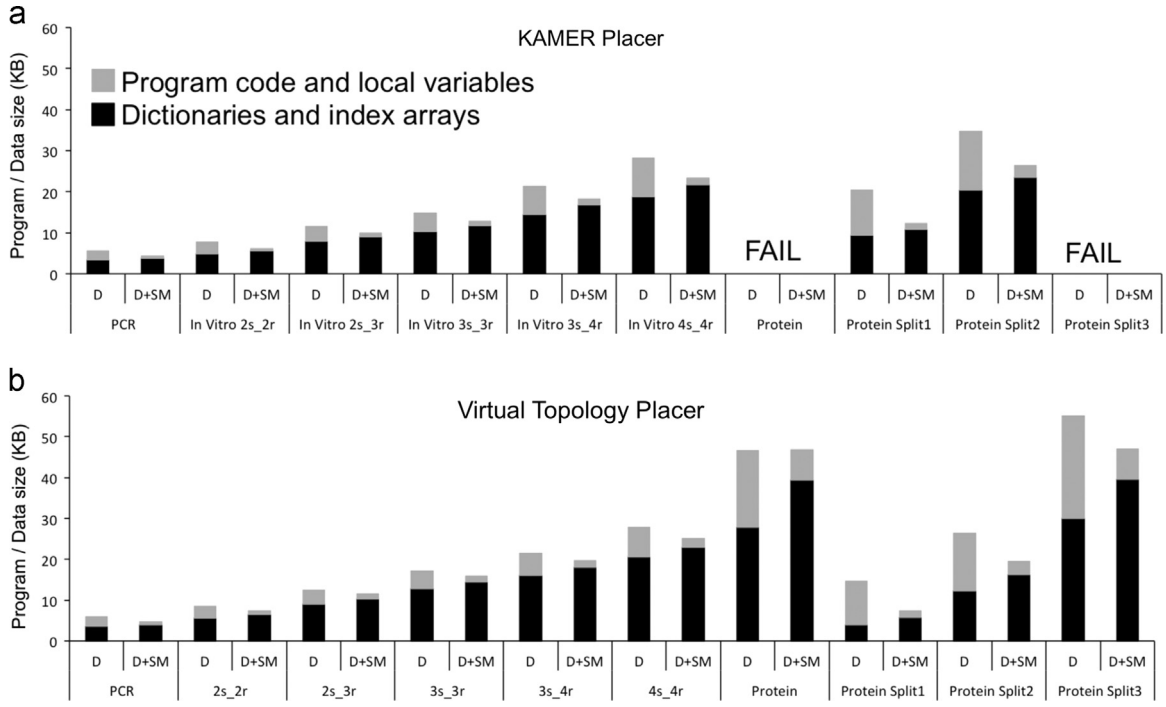


Fig. 30. Effectiveness of compression algorithms on program/data sizes using the (a) KAMER [2,44] and (b) virtual topology (VT) [21,22] placers.

and Luo et al. offloaded the task onto a small FPGA; to minimize the cost and complexity of our microcontroller interface, we avoid runlength encoding.

It remains possible to specify assays whose state machine implementations are too large to fit into microcontroller memory, even with compression. In this case, there is no tangible benefit to compression, as the PC is still required to run the system. In principle, a PC could store the state machine and dictionary as data and decompose them into chunks that can fit into the microcontroller's memory; when the state machine executing on the microcontroller enters a state or accesses a dictionary entry that is not in the microcontroller's memory, the access could be forwarded back to the PC, which then delivers a new chunk of code and data. For large chunks, the transmission latency could be significant, potentially causing the assay to pause. Rather than trying to mitigate this overhead, it is easier to execute the assay in an uncompressed form with predictable communication, as described in Section 6.2.

7. Related work

To the best of our knowledge, no comparable software platform for DMFB control has been released to the wider research community. The closest related project is an open source hardware/software system developed at the University of Toronto called DropBot [17]. DropBot provides real-time monitoring of droplet position and velocity and application of constant droplet driving forces in the presence of variations in amplifier load and device capacitance due to physical variations in the insulator. DropBot does not provide automatic compilation. The user must manually select electrodes to activate and deactivate over time; rather than clicking on an image of the chip, as in our system, the user is presented with a webcam video overlay of the chip.

Bhattacharjee et al. [5] described a formal verification checking system for DMFBs, which share some principle similarities with the verification steps in our system, as described in Section 4.4; they do not provide verification rules for pin mapping and PCB escape routing. The paper mentions a DMFB simulator called SimBioSys, and provides a screenshot; however, no such software could be located online.

8. Conclusion

We encourage researchers who want to study or otherwise use DMFB compilation and PCB synthesis to download and use our framework. We hope that researchers with a background in computer science and engineering will develop and contribute new algorithms and features to the framework, as it provides a common, standardized platform for dissemination and comparison. We also hope that practitioners who use DMFBs in their laboratories will adopt the framework, eventually switching to BioCoder as a specification language. In the long term, we intend to leverage the framework at UC Riverside to create undergraduate and graduate courses on programmable microfluidics (focusing on DMFB technology specifically) and to support undergraduate senior design projects and graduate-level projects and theses.

Acknowledgment

This work was supported in part by NSF Grant CNS-1035603. Daniel Grissom was supported by an NSF Graduate Research Fellowship and a UC Riverside Dissertation Year Fellowship. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of the NSF.

References

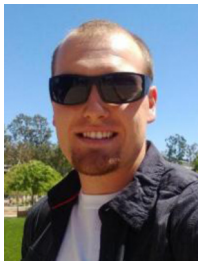
[1] V. Ananthanarayanan, W. Thies, Biocoder: a programming language for standardizing and automating biology protocols, J. Biol. Eng. 8 (13) (2010) 1–3.
[2] K. Bazargan, R. Kastner, M. Sarrafzadeh, Fast template placement for reconfigurable computing systems, IEEE Des. Test Comput. 17 (1) (2000) 68–83.
[3] S. Berry, J. Kedzierski, B. Abedian, Irreversible electrowetting on thin fluoropolymer films, Langmuir 23 (24) (2007) 12429–12435.
[4] S. Bhattacharjee, A. Banerjee, B.B. Bhattacharya, Multiple dilution sample preparation using digital microfluidic biochips, in: Proceedings of the ISED, Kolkata, India, Dec. 19–22, 2012, pp. 188–192.
[5] S. Bhattacharjee, A. Banerjee, K. Chakrabarty, B.B. Bhattacharya, Correctness checking of biochemical protocol realizations on a digital microfluidic biochip, in: Proceedings of the VLSI Design, Mumbai, India, Jan. 5–9, 2014, pp. 504–509.
[6] S. Bhattacharjee, A. Banerjee, T.-Y. Ho, K. Chakrabarty, B.B. Bhattacharya., On producing linear dilution gradient of a sample with a digital microfluidic biochip, in: Proceedings of the ISED, Singapore, Dec. 10–12, 2013, pp. 77–81.
[7] K.F. Böhringer, Modeling and controlling parallel tasks in droplet-based microfluidic systems, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 25 (2) (2006) 334–344.
[8] J.-W. Chang, S.-H. Yeh, T.-W. Huang, T.-Y. Ho., An ILP-based routing algorithm for pin-constrained EWOD chips with obstacle avoidance, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 32 (11) (2013) 1655–1667.
[9] J.-W. Chang, S.-H. Yeh, T.-W. Huang, T.-Y. Ho, Integrated fluidic-chip co-design methodology for digital microfluidic biochips, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 32 (2) (2013) 216–227.
[10] Y.-H. Chen, C.-L. Hsu, L.-C. Tsai, T.-W. Huang, T.-Y. Ho, A reliability-oriented placement algorithm for reconfigurable digital microfluidic biochips using 3-D deferred decision making technique, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 32 (8) (2013) 1151–1162.
[11] T.-W. Chiang, C.-H. Liu, J.-D. Huang, Graph-based optimal reactant minimization for sample preparation on digital microfluidic biochips, in: Proceedings of the VLSI-DAT, Hsinchu, Taiwan, Apr. 22–24, 2013, pp. 1–4.
[12] M. Cho, D.Z. Pan., A high-performance droplet routing algorithm for digital microfluidic biochips, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 27 (10) (2008) 1714–1724.
[13] K. Choi, et al., Automated digital microfluidic platform for magnetic-particle-based immunoassays with optimization by design of experiments, Anal. Chem. 85 (20) (2013) 9638–9646.
[14] M. Dhindsa, S. Kuiper, J. Heikenfeld, Reliable and low-voltage electrowetting on thin parylene films, Thin Solid Films 519 (10) (2011) 3346–3351.
[15] J. Ding, K. Chakrabarty, R.B. Fair, Scheduling of microfluidic operations for reconfigurable two-dimensional electrowetting arrays, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 20 (12) (2001) 1463–1468.
[16] A.I. Drygiannakis, A.G. Papathanasiou, A.G. Boudouvis, On the connection between dielectric breakdown strength, trapping of charge, and contact angle saturation in electrowetting, Langmuir 25 (1) (2009) 147–152.
[17] R. Fobel, C. Fobel, A.R. Wheeler., DropBot: an open-source digital microfluidic control system with precise control of electrostatic driving force and instantaneous drop velocity measurement, Appl. Phys. Lett. 102 (19) (May 2013) 193513-1–193513-4.
[18] GraphViz, ⟨http:/www.graphviz.org/⟩.
[19] E. Griffith, S. Akella, M.K. Goldberg, Performance characterization of a reconfigurable planar-array digital microfluidic system, IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. 25 (2) (2006) 345–357.
[20] D. Grissom, P. Brisk, A field-programmable pin-constrained digital microfluidic biochip, in: Proceedings of the DAC, Article 46, Austin, TX, USA, 2013.
[21] D. Grissom, P. Brisk, Fast online synthesis of digital microfluidic biochips, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 33 (3) (2014) 356–359.
[22] D. Grissom, P. Brisk, Fast online synthesis of generally programmable digital microfluidic biochips, in: Proceedings of the CODES+ISSS, Tampere, Finland, Oct. 7–12, 2012, pp. 413–422.
[23] D. Grissom, P. Brisk, Path scheduling on digital microfluidic biochips, in: Proceedings of the DAC, San Francisco, CA, USA, Jun. 3–7, 2012, pp. 26–35.
[24] D. Grissom, C. Curtis, P. Brisk, Interpreting assays with control flow on digital microfluidic biochips, ACM J. Emerg. Technol. 10 (3) (2014) 24-1–24-30.
[25] D. Grissom, J. McDaniel, P. Brisk, A low-cost field-programmable pin-constrained digital microfluidic biochip, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 33 (11) (2014) 1657–1670.
[26] D. Grissom, et al., A digital microfluidic biochip synthesis framework, in: Proceedings of the VLSI-SoC, Santa Cruz, CA, Oct. 7–10, 2012, pp. 177–182.
[27] B. Hadwen, et al., Programmable large area digital microfluidic array with integrated droplet sensing for bioassays, Lab-on-a-Chip 13 (18) (2012) 3305–3313.
[28] Y.-L. Hsieh, T.-Y. Ho, K. Chakrabarty, A reagent-saving mixing algorithm for preparing multi-target biochemical samples using digital microfluidics, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 31 (11) (2012) 1656–1669.

[29] J.-D. Huang, C.-H. Liu, T.-W. Chiang, Reactant minimization during sample preparation on digital microfluidic biochips using skewed mixing trees, in: Proceedings of the ICCAD, San Jose, CA, USA, Nov. 7–11, 2012, pp. 377–383.
[30] J.-D. Huang, C.-H. Liu, H.-S. Lin, Reactant and waste minimization in multi-target sample preparation on digital microfluidic biochips, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 32 (10) (2013) 1484–1494.
[31] L.-X. Huang, B. Koo, C.-J. Kim, Evaluation of anodic Ta_2O_5 as the dielectric layer for EWOD devices, in: Proceedings of the IEEE MEMS, Paris, France, Jan. 29–Feb. 2, 2012, pp. 428–431.
[32] T.-W. Huang, T.-Y. Ho., A fast routability- and performance-driven droplet routing algorithm for digital microfluidic biochips, in: Proceedings of the ICCD, Lake Tahoe, CA, USA, Oct. 4–7, 2009, pp. 445–450.
[33] T.-W. Huang, T.-Y. Ho, A two-stage integer linear programming-based droplet routing algorithm for pin-constrained digital microfluidic biochips, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 30 (2) (2011) 215–228.
[34] T.-W. Huang, T.-Y. Ho, K. Chakrabarty, Reliability-oriented broadcast electrode-addressing for pin-constrained digital microfluidic biochips, in: Proceedings of the ICCAD, San Jose, CA, USA, Nov. 7–11, 2011, pp. 448–455.
[35] T.-W. Huang, C.-H. Lin, T.-Y. Ho., A contamination aware droplet routing algorithm for the synthesis of digital microfluidic biochips, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 29 (11) (2010) 1682–1695.
[36] T.-W. Huang, H.-Y. Su, T.-Y. Ho, Progressive network-flow based power-aware broadcast addressing for pin-constrained digital microfluidic biochips, in: Proceedings of the DAC, San Diego, CA, USA, June 5–10, 2011, pp. 741–746.
[37] T.-W. Huang, S.-Y. Yeh, T.-Y. Ho, A network-flow based pin-count aware routing algorithm for broadcast-addressing EWOD chips, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 30 (12) (2011) 1786–1799.
[38] Y. Li, et al., Anodic Ta_2O_5 for CMOS compatible low voltage electrowetting-on-dielectric device fabrication, in: Proceedings of the ESSDERC, Munich, Germany, Sep. 11–13, 2007, pp. 446–449.
[39] C. Liao, and S. Hu. Multiscale variation-aware techniques for high-performance digital microfluidic lab-on-a-chip component placement. IEEE Trans. Nanobiosci 10(1), 2011, 51–58.
[40] C.C.-Y. Lin, Y.-W. Chang, Cross-contamination aware design methodology for pin-constrained digital microfluidic biochips, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 30 (6) (2006) 817–828.
[41] C.C.-Y. Lin, Y.-W. Chang., ILP-based pin-count aware design methodology for microfluidic biochips, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 29 (9) (2010) 1315–1327.
[42] C.-H. Liu, H.-H. Chang, T.-C. Liang, J.-D. Huang, Sample preparation for many-reactant bioassay on DMFBs using common dilution operation sharing, in: Proceedings of the ICCAD, San Jose, CA, USA, Nov. 18–21, 2013, pp. 615–621.
[43] C.-H. Liu, K.-C. Liu, J.-D. Huang, Latency-optimization synthesis with module selection for digital microfluidic biochips, in: Proceedings of the IEEE SOCC, Erlangen, Germany, Sep. 4–6, 2013, pp. 159–164.
[44] Y. Lu, T. Marconi, G. Gaydadjiev, K. Bertels., An efficient algorithm for free resources management on the FPGA, in: Proceedings of the DATE, Munich, Germany, Mar. 10–14, 2008, pp. 1095–1098.
[45] Y. Luo, B.B. Bhattacharya, T.-Y. Ho, K. Chakrabarty, Optimization of polymerase chain reaction on a cyberphysical digital microfluidic biochip, in: Proceedings of the ICCAD, San Jose, CA, USA, Nov. 18–21, 2013, pp. 622–629.
[46] Y. Luo, K. Chakrabarty, Design of pin-constrained general-purpose digital microfluidic biochips, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 32 (9) (2013) 1307–1320.
[47] Y. Luo, K. Chakrabarty, T.-Y. Ho, Error recovery in cyberphysical digital microfluidic biochips, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 32 (1) (2013) 59–72.
[48] Y. Luo, K. Chakrabarty, T.-Y. Ho, Real-time error recovery in cyberphysical digital microfluidic biochips using a compact dictionary, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 32 (12) (2013) 1839–1852.
[49] Q. Ma, T. Yan, M.D.F. Wong, A negotiated congestion based router for simultaneous escape routing, in: Proceedings of the ISQED, San Jose, CA, USA, Mar. 22–24, 2010, pp. 606–610.
[50] E. Maftei, P. Pop, J. Madsen, Tabu search-based synthesis of dynamically reconfigurable digital microfluidic biochips, in: Proceedings of CASES, Grenoble, France, Oct. 11–16, 2009, pp. 195–204.
[51] E. Maftei, P. Pop, J. Madsen, Routing-based synthesis for digital microfluidic biochips, in: Proceedings of the 2010 International Conference on Compilers, Architectures and Synthesis for Embedded Systems, 2012, pp. 41–50.
[52] J. McDaniel, D. Grissom, P. Brisk, Multi-terminal PCB escape routing for digital microfluidic biochips using negotiated congestion, in: Proceedings of the VLSI-SoC, Playa Del Carmen, Mexico, Oct. 6–8, 2014, pp. 219–224.
[53] L. McMurchie, C. Ebeling, PathFinder: a negotiation-based performance-driven router for FPGAs, in: Proceedings of FPGA, Monterey, CA, USA, Feb. 12–14, 1995, pp. 111–117.
[54] D. Mitra, S. Roy, K. Chakrabarty, B.B. Bhattacharya, On-chip sample preparation with multiple dilutions using digital microfluidics, in: Proceedings of ISVLSI, Amherst, MA, USA, Aug. 19–21, 2012, pp. 314–319.
[55] H. Moon, S.K. Cho, R.L. Garrell, C.-J. Kim, Low voltage electrowetting on dielectric, J. Appl. Phys. 92 (7) (2002) 4080–4087.
[56] R. Mukherjee, et al., A heuristic method for co-optimization of pin assignment and droplet routing in digital microfluidic biochip, in: Proceedings of VLSI Design, Hyderabad, India, Jan. 7–11, 2012, pp. 227–232.
[57] M.A. Murran, and H. Najjaran. Capacitance-based droplet position estimator for digital microfluidic devices. Lab-on-a-Chip 12(11), 2012, 2053–2059.
[58] J.H. Noh, J. Noh, E. Kreit, J. Heikenfeld, P.D. Rack, Toward active-matrix lab-on-a-chip: programmable electrofluidic control enabled by arrayed oxide thin film transistors, Lab-on-a-Chip 12 (2) (2011) 353–360.
[59] K. O'Neal, D. Grissom, P. Brisk, Force-directed list scheduling for digital microfluidic biochips, in: Proceedings of VLSI-SoC, Santa Cruz, CA, USA, Oct. 7–10, 2012, pp. 7–11.
[60] P. Paik, V. Pamula, R. Fair, Rapid droplet mixers for digital microfluidic systems, Lab-on-a-Chip 3 (4) (2003) 253–259.
[61] J.K. Park, S.J. Lee, K.H. Kang, Fast and reliable droplet transport on single-plate electrowetting on dielectrics using nonfloating switching method, J. Biomicrofluid. 4 (2) (2010) 024102-1–024102-8.
[62] M.G. Pollack, A.D. Shenderov, R.B. Fair, Electrowetting-based actuation of droplets for integrated microfluidics, Lab-on-a-Chip 2 (2) (2002) 96–101.
[63] A.J. Ricketts, K. Irick, N. Vijaykrishnan, M.J. Irwin, Priority scheduling in digital microfluidics-based biochips, in: Proceedings of DATE, Munich, Germany, Mar. 6–10, 2006, pp. 239–334.
[64] P. Roy, H. Rahaman, P. Dasgupta, A novel droplet routing algorithm for digital microfluidic biochips, in: Proceedings of the GLSVLSI, Providence, RI, USA, May 16–18, 2010, pp. 441–446.
[65] S. Roy, B.B. Bhattacharya, K. Chakrabarty, Optimization of dilution and mixing of biochemical samples using digital microfluidic biochips, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 29 (11) (2010) 1696–1708.
[66] S. Roy, B.B. Bhattacharya, K. Chakrabarty, Waste-aware dilution and mixing of biochemical samples with digital microfluidic biochips, in: Proceedings of DATE, Grenoble, France, Mar. 14–18, 2011, pp. 1059–1064.
[67] S. Roy, B.B. Bhattacharya, S. Ghoshal, K. Chakrabarty, On-chip dilution from multiple concentrations of a smple using digital microfluidics, in: Proceedings of VDAT, Jaipur, India, July 27–30, 2013, pp. 274–283.
[68] S.C.C. Shih, et al., Digital microfluidics with impedance sensing for integrated cell culture and analysis, Biosens. Bioelectron. 42 (4) (2013) 314–320.
[69] F. Su, K. Chakrabarty, High-level synthesis of digital microfluidic biochips, ACM J. Emerg. Technol. 3 (4) (2008) 16-1–16-32.
[70] F. Su, K. Chakrabarty, Module placement for fault-tolerant microfluidics-based biochips, ACM Trans. Des. Autom. Electron. Syst. 11 (3) (2006) 682–710.
[71] F. Su, W. Hwang, K. Chakrabarty, Droplet routing in the synthesis of digital microfluidic biochips, in: Proceedings of DATE, Munich, Germany, Mar. 6–10, 2006, pp. 323–328.
[72] W. Thies, J.P. Urbanski, T. Thorsen, S.P. Amarasinghe, Abstraction layers for scalable microfluidic biocomputing, Nat. Comput. 7 (2) (2008) 255–275.
[73] UC Riverside Digital Microfluidic Biochip Simulator, ⟨http://microfluidics.cs.ucr.edu⟩.
[74] H.J.J. Verheijen, M.W.J. Prins, Reversible electrowetting and trapping of charge: model and experiments, Langmuir 15 (20) (1999) 6616–6620.
[75] T. Xu, K. Chakrabarty, Integrated droplet routing and defect tolerance in the synthesis of digital microfluidic biochips, ACM J. Emerg. Technol. 4 (3) (2008) 11-1–11-24.
[76] T. Xu, W.L. Hwang, F. Su, K. Chakrabarty, Automated design of pin-constrained digital microfluidic biochips under droplet-interference constraints, ACM J. Emerg. Technol. 3 (3) (2007) 14-1–14-23.
[77] T. Xu, K. Chakrabarty, F. Su, Defect-aware high-level synthesis and module placement for microfluidic biochips, IEEE Trans. Biomed. Circuits Syst. 2 (1) (2008) 50–62.
[78] T. Yan, M.D.F. Wong, Correctly modeling the diagonal capacity in escape routing, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 31 (2) (2012) 285–293.
[79] S.-H. Yeh, J.-W. Chang, T.-W. Huang, T.-Y. Ho, Voltage-aware chip-level design for reliability-driven pin-constrained EWOD chips, in: Proceedings of ICCAD, San Jose, CA, USA, Nov. 5–8, 2012, pp. 353–360.
[80] S.-T. Yu, S.-H. Yeh, T.-Y. Ho. Reliability-driven chip-level design for high-frequency digital microfluidic biochips, in: Proceedings of ISPD, Petaluma, CA, USA, Mar. 30–Apr. 2, 2014, pp. 133–140.
[81] P.-H. Yuh, C.-L. Yang, Y.-W. Chang, BioRoute: a network flow- based routing algorithm for the synthesis of digital microfluidic biochips, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 27 (11) (2008) 1928–1941.
[82] P.-H. Yuh, C.-L. Yang, Y.-W. Chang, Placement of defect-tolerant digital microfluidic biochips using the T-tree formulation, ACM J. Emerg. Technol. 3 (3) (2007) 13-1–13-32.
[83] Y. Zhao, K. Chakrabarty, Cross-contamination avoidance for droplet routing in digital microfluidic biochips, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 31 (6) (2012) 817–830.
[84] Y. Zhao, K. Chakrabarty, Simultaneous optimization of droplet routing and control-pin mapping to electrodes in digital microfluidic biochips, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 31 (2) (2012) 242–254.
[85] Y. Zhao, K. Chakrabarty, R. Sturmer, V.K. Pamula, Optimization techniques for the synchronization of concurrent fluidic operations in pin-constrained digital microfluidic biochips, IEEE Trans. Very Large Scale Integr. 20 (6) (2012) 1132–1145.
[86] Y. Zhao, T. Xu, K. Chakrabarty, Broadcast electrode-addressing and scheduling methods for pin-constrained digital microfluidic biochips, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 30 (7) (2011) 986–999.
[87] Y. Zhao, T. Xu, K. Chakrabarty, Integrated control-path design and error recovery in the synthesis of digital microfluidic biochips, ACM J. Emerg. Technol. 6 (3) (2010) 11-1–11-28.

Daniel Grissom received the B.S. degree in Computer Engineering from the University of Cincinnati (UC), Ohio, in 2008 and the M.S. and Ph.D. degrees in Computer Science from the University of California, Riverside (UCR), in 2011 and 2014, respectively. He is now an Assistant Professor at Azusa Pacific University.

To date, he has published over a dozen papers/journals and filed one provisional patent in the field of digital microfluidics. His research interests include languages, synthesis and hardware interfacing for digital microfluidic biochips.

Christopher Curtis received the B.S. degree in Computer Science from the University of California at Riverside (UCR), in 2013. He is presently pursuing a Ph.D. degree in Computer Science at UCR. His research interests include languages, synthesis and hardware interfacing for digital microfluidic biochips.

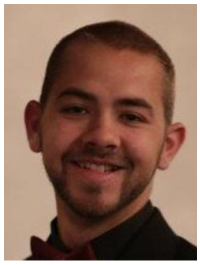


Skyler Windh received the B.S. Degree in Computer Science from the University of California, Riverside (UCR). He is presently pursuing a Ph.D. degree in Computer Science at UCR. His research interests include the usage of FPGAs and GPUs to accelerate database and data mining applications.

Calvin Phung received the B.S. Degree in Computer Science from the University of California, Riverside (UCR) in 2012. He is presently pursuing a Ph.D. degree in Computer Science at UCR. He is the leading engineer for the first brain-training game for the Brain Game Center at UCR, a collaborative project between the Computer Graphics and the Cognitive Neuroscience Labs. His research interests include machine learning to predict the performance in future sessions for general cognitive training applications to provide challenging, but not overwhelming, application experiences for the user.

Navin Kumar received the Bachelor of Engineering Degree in Computer Science from the CMR Institute of Technology, Bangalore, India, in 2011. He received the M.S. Degree in Computer Science from the University of California, Riverside (UCR) in 2014. From 2011-2012, he worked as a project engineer with Wipro Technologies, Pune, India. He worked as a research intern at Samsung Research America, Inc., in 2013, and has worked there as a software engineer since 2014. His research interests include open source technologies, distributed systems, Big Data, and Cloud Computing.

Zachary Zimmerman received the B.S. Degree in Computer Science from the University of California, Riverside (UCR) in 2015. He will begin the M.S. program in Computer Science at UCR starting in Fall 2015. His research interests include programmable microfluidics, graphics, machine learning, and computer vision.



Kenneth O'Neal received the B. S. Degree in Computer Engineering from the University of California, Riverside (UCR) in 2012. He is presently pursuing a Ph.D. degree in Computer Science at UCR. He worked as an intern at Intel Corporation in 2014 and 2015. His research interests include optimization and exploration in power and performance for heterogeneous computing platforms with an emphasis on GPUs, and algorithmic design for high-level synthesis of Digital Microfluidic Biochips.

Jeffrey McDaniel received the B.S. Degree in Computer Science and Pure Mathematics and the M.S. Degree in Computer Science from the University of California at Riverside (UCR), in 2011 and 2014 respectively. He is presently pursuing the Ph.D. degree in Computer Science at UCR. To date, he has published five conference and two journal papers. His research interests include languages, synthesis and hardware interfacing for continuous- flow microfluidic biochips.

Nicholas Liao is a Freshman studying Computer Science at the Georgia Institute of Technology. He graduated from the Bishop's School, La Jolla, California, in 2014. In high school, he studied computer programming languages, web design and computer gaming. He completed internships at the University of California, Riverside (UCR), the University of California, San Diego (UCSD), and the Jiexin Network in Beijing. His personal interests include music, computer gaming, and water polo.

Philip Brisk received the B.S., M.S., and Ph.D. Degrees, all in Computer Science, from UCLA in 2002, 2003, and 2006, respectively. From 2006-2009 he was a Postdoctoral Scholar in the Processor Architecture laboratory in the School of Computer and Communication Sciences at the École Polytechnique Fédérale de Lausanne, (EPFL), in Lausanne, Switzerland. Since 2009, he has been an Assistant Professor in the Department of Computer Science and Engineering at the University of California, Riverside (UCR), and has been promoted to Associate Professor as of July 1, 2015. His work has received the Best Paper Award at CASES 2007 and FPL 2009, and been nominated for the Best Paper Award at DAC 2007 and HiPEAC 2010. Dr. Brisk is or has been a member of the Program Committees of several international conferences and workshops including DAC, DATE, ASPDAC, VLSI-SoC, FPGA, FPL, FPT, and others. He has been the general (co-) chair of IEEE SIES 2009, IEEE SASP 2010, and IWLS 2011, and participated in the organizing committee of many other conferences and symposia as well.