Fast Online Synthesis of Generally Programmable Digital Microfluidic Biochips

Daniel Grissom, Philip Brisk
Department of Computer Science and Engineering
University of California, Riverside
{grissomd, philip}@cs.ucr.edu

ABSTRACT
We introduce an online synthesis flow for digital microfluidic biochips, which will enable real-time response to errors and control flow. The objective of this flow is to facilitate fast assay synthesis while minimally compromising the quality of results. In particular, we show that a virtual topology, which constrains the allowable locations of assay operations such as mixing, dilution, sensing, etc., in lieu of traditional placement, can significantly speed up the synthesis process without significantly lengthening assay execution time.

Categories and Subject Descriptors
B.7.2 [**Integrated Circuits**]: Design Aids – *Placement and routing*.

General Terms
Algorithms, Design, Performance.

Keywords
Microfluidics, Laboratory-on-Chip (LoC), Electrowetting-on-Dielectric (EWoD), Synthesis.

1. INTRODUCTION
With the emergence of novel, scalable, flexible digital microfluidic biochips (DMFBs) [15], new features such as end-user programmability and online synthesis will revolutionize microfluidic applications. Instead of application-specific DMFBs, low-cost, general-purpose DMFBs will provide a reusable, flexible, programmable platform. With the notion of end-user programmability being introduced to DMFBs, control-flow constructs present exciting, new possibilities for microfluidic applications. Consequently, when control-flow is introduced, synthesis (**Figure 1**) will need to be performed online since the order of *assays* (biochemical reactions) to be executed is no longer deterministic, but instead dependent on live-feedback from the DMFB [2][13].

In contrast to *offline compilers*, which synthesize assays as deterministic state-machines, an *online interpreter* will act more like a virtual machine which manages the DMFB's resources and interprets assays on-the-fly. **Figure 2** shows the tradeoffs that need to be made when moving synthesis online. During offline compilation, optimized designs are created with little concern to runtime since the synthesis process is run once and the compiled "executable binary" is packaged into an application-specific device. With a programmable DMFB, the end-user will have to wait each time a programmed assay is synthesized. Furthermore, each time a branch is taken, the user will have to wait as the target assay of the branch is interpreted online. Thus, new synthesis methods are needed that concede optimality in performance and area to reduce algorithmic runtimes from seconds/minutes to milliseconds and achieve a greater amount of flexibility [13].

CODES+ISSS'12, October 7–12, 2012, Tampere, Finland.

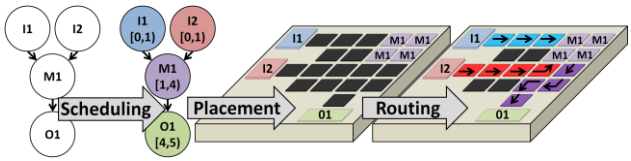


Figure 1. DMFB synthesis consists of *scheduling, placing* and *routing*.

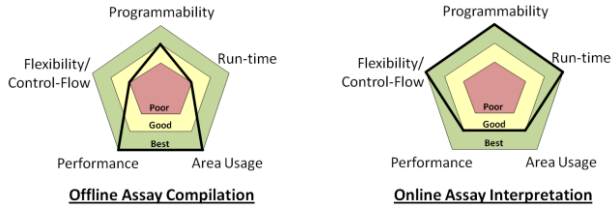


Figure 2. Offline vs. online synthesis tradeoffs.

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

With offline compilation, a single DAG must be created that details every possible execution path, which quickly becomes intractable as the compiler must account for numerous paths that will never be taken [2]. Instead, upon completion of one assay, an online interpreter could immediately interpret and execute the next assay with only milliseconds of downtime between assays.

Although synthesis has been performed entirely offline up to this point, Ho, Chakrabarty and Pop suggest that online systems are forthcoming with the development of "specialized heuristics" which can perform synthesis in milliseconds [10]; Luo Chakrabarty, and Ho [13] have implemented one such specialized heuristic for an error detection and recovery scheme based on checkpointing: at each checkpoint, a droplet is routed to a sensor that detects whether its concentration is satisfactory; if not, the assay is re-synthesized on-the-fly to repeat the sequence of operations that produced the droplet, interleaving the schedule of these newly-introduced operations with concurrent operations that do not depend on the droplet that failed the checkpoint.

1.2 Contribution

Assays are synthesized by computing solutions for three NP-complete problems, as seen in **Figure 1**. Before synthesis, an assay is modeled as a directed acyclic graph (DAG), where the nodes and edges represent operations and operation dependencies, respectively. Each assay operation is then assigned start/stop times during resource-constrained scheduling [19]. During the placement stage, the scheduled operations are assigned specific locations, called *modules*, on the array [25]. Finally, the routing stage computes droplet paths between subsequent modules and I/O ports so droplets arrive safely at each destination [27].

We present an online synthesis flow that can interpret assays and map them onto a cross-referenced, fully-addressable or active-matrix DMFB [15] in milliseconds, making it appropriate for both offline and online synthesis. Our key contribution is a virtual topology that defines distinct regions for module placement and droplet routing. With our topology in mind, list scheduling [24] is used to quickly produce schedules. Placement, which has been solved in the past by iterative improvement algorithms [25][32] or integer linear programming (ILP) [12], is simplified to a binding problem, which can be solved efficiently in polynomial-time [11]. The placement defined by the virtual topology provides dedicated routing cells which ease the router's job. We simplify an existing router [20] to compute droplet paths very quickly.

1.3 DMFB Technology Overview

A DMFB manipulates discrete droplets of fluid on a 2D array of electrodes (**Figure 3(a)**) through electrowetting, a process that induces droplet motion [18]. **Figure 3(b)** shows a droplet sandwiched between a ground electrode and a set of independent control electrodes. The droplet is centered over one electrode (CE2), but overlaps adjacent electrodes CE1 and CE3. If a voltage is applied to CE1 or CE3, the surface energy gradient induces motion and the droplet will move to the center of the newly activated electrode(s). During each droplet-actuation *cycle*, a set of electrodes is activated which moves droplets from electrode to electrode. Basic assay operations such as transport, mixing, merging and splitting are performed through the appropriate sequence of electrode activations over a number of cycles.

There are several classes of DMFBs that provide varying levels of droplet control. Typical *direct-addressing* (*fully-addressable*) DMFBs have one control pin for each electrode (i.e. $(M \times N)$ control pins for an $(M \times N)$ array of electrodes) so each electrode (droplet) can be independently controlled at all times. However, the wiring cost of independently controlled electrodes, especially as array sizes grow, has motivated cheaper designs [29].

Cross-referencing DMFBs use $(M + N)$ control pins to control an $(M \times N)$ array of electrodes [5]. In this scheme, each row and each column has a single control pin; when a particular column m and row n are activated, the electrode at (m, n) is activated. Multiple columns and rows can be simultaneously activated, but may cause superfluous electrode activation, yielding undesired droplet movement [28]. Thus, once a route for a direct-addressing DMFB is computed, each droplet-actuation cycle is serialized across multiple droplet-actuation cycles, resulting in prolonged routing times and increased algorithmic complexity.

Pin-constrained DMFBs represent another addressing scheme. An assay is first synthesized as if on a direct-addressing DMFB; then, special heuristics attempting to solve the clique partitioning problem (NP-Hard) are used to minimize the total number of control pins, based on which electrodes can be activated together without causing undesired droplet movement [29].

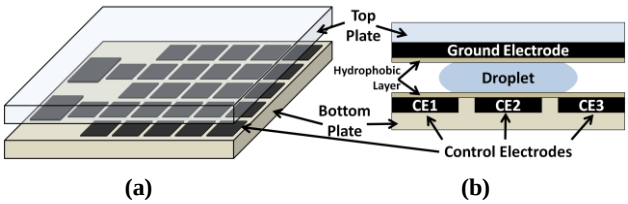


Figure 3. (a) DMFB with a 2D array of electrodes; (b) DMFB cross-section.

To summarize, pin-constrained designs offer minimal product costs, are inflexible and cannot be reprogrammed after being manufactured; cross-referencing DMFBs are reprogrammable, but add another layer of complexity that must be handled to serialize droplet motion [28].

Fortunately, *active-matrix addressing* designs are emerging which give independent control of $(M \times N)$ electrodes while using only $(M + N)$ control pins [15]. Active matrix addressing can scale without growing prohibitively expensive, while maintaining the maximum level of flexibility and control so that assays can be programmed with minimal levels of algorithmic complexity. *The online synthesis flow presented in this paper is compatible with direct, cross-referencing, and active-matrix addressing DMFBs.*

2. RELATED WORK

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

2.1 Scheduling

Su and Chakrabarty present modified list scheduling (MLS) and genetic algorithm (GA) heuristics, as well as an optimal integer linear programming (ILP) model for scheduling microfluidic operations onto a DMFB [24]. As expected, the ILP implementation consumes a large amount of time to compute optimal solutions. Although the GA finds optimal or near-optimal results in much less time than ILP, its iterative nature still results in large computation times. MLS produces schedules comparable to GA in much less time. Other scheduling algorithms such as Ricketts' hybrid genetic algorithm [19] and Maftei's tabu search scheduler [14] are iterative improvement algorithms which spend anywhere from 4 seconds to 1 hour computing schedules. We chose list scheduling as the base scheduler for our framework, but other fast schedulers being developed now [8][16] or in the future could be used as well.

2.2 Placement

At the physical level, all electrodes are equally capable of performing the basic microfluidic operations (i.e. merging, mixing, splitting, transport, storage); hence, basic operations can be performed anywhere on a DMFB array. The objective for most placers is to pack as many concurrent operations/modules into as little area as possible. Several direct-addressing placement and unified scheduling-placement algorithms [25][26][30][32] use simulated annealing, which run in minutes or tens of seconds; in contrast, our online flow completes in tens of milliseconds.

Griffith et al. [6] place a virtual topology onto the DMFB, which dictates separate regions for assay operations and droplet routing; however, they only present results for one assay, and their implementation suffers from deadlocks during droplet routing. Our approach is similar, but does not suffer from deadlock; in the absolute worst case, our router will transport one droplet at a time; however, we include a compaction step to transport multiple droplets concurrently.

2.3 Routing

Böhringer [3] modeled droplet routing as an A* search, similar to path planning in robotics, achieving an optimal-length solution, when routable. Su et al. route droplets sequentially and redo placement when routing fails [27]. BioRoute [31] uses a min-cost max-flow algorithm to compute several routes at once, followed by negotiation-based detailed routing. Cho and Pan [4] route droplets one-by-one and sort them based on a *bypassability* metric; if a deadlock occurs, droplets are moved to *concession zones* to break the deadlock. Huang and Ho [9] construct a system of global routing tracks, which are aligned in the same direction as the majority of droplets traveling on that tract. They use an entropy-based equation to determine the order in which droplets are routed, and finally, compact the routes using dynamic programming. Since the aforementioned methods were designed for offline routing, few mention runtimes [3][4][27]. BioRoute [31] and Huang's algorithm [9] both report runtimes below 1s on a desktop PC. The router used in our online flow, in contrast, achieves comparable runtimes on an Intel Atom™ processor.

2.4 Combined Methods

Most work on synthesis has focused on the scheduling, placement or routing problems in isolation. Several papers, however, solve some of these problems together, using iterative improvement heuristics [13][25][26][30][32], whose runtime is prohibitive. These approaches address problems that can arise when one stage of synthesis does not consider the next. For instance, a placer can generate a valid placement that is unroutable. Our virtual topology ensures routability by leaving room for droplets to pass between adjacent "modules" where mixing, storage, and other assay operations are performed.

3. VIRTUAL TOPOLOGY

Our online interpreter utilizes a virtual topology, as seen in **Figure 4**, and takes advantage of its order and structure to yield fast algorithmic runtimes for scheduling, placement and routing. First, we define a *cell* as the 2D area covering an electrode. The virtual topology shows evenly spaced modules (3x3 squares of cells) where basic droplet operations (i.e. merge, mix, split, store) are performed. If at least one of a module's cells is augmented with an external detector or heater, the module can also perform detect or heat operations, respectively. The white cells indicate the area of the DMFB array used explicitly for routing droplets between modules and I/O ports (not pictured); however, any cell can be used for routing if a module is not in use. Dedicated routing cells ensure there is a valid path between any source-destination pair. A perimeter of *interference region (IR) cells* surrounds each module [27], so that interference-free droplet routes can be computed easily; this topology ensures that there is at least one path between all DMFB inputs, modules, and outputs. The inputs and outputs, (not shown in **Figure 4**), are on the perimeter of the chip.

3.1 Module Topology and Synchronization

To help prevent droplet deadlock, droplets have well-defined module entrance and exit locations, as seen in the 3x3 module of **Figure 5(a)**. The two entrances are in the northwest and southwest corners, while the exits are in the northeast and southeast corners. By providing distinct entrances and exits, we prevent droplet deadlock by allowing droplets leaving a module to wait safely in their exit cells as long as necessary to avoid deadlock in the routing cells. **Figure 5(b)** and **Figure 5(c)** show that modules can be elongated along the X- or Y-axis to accommodate larger 2x4 mixers, often used in literature [17][19].

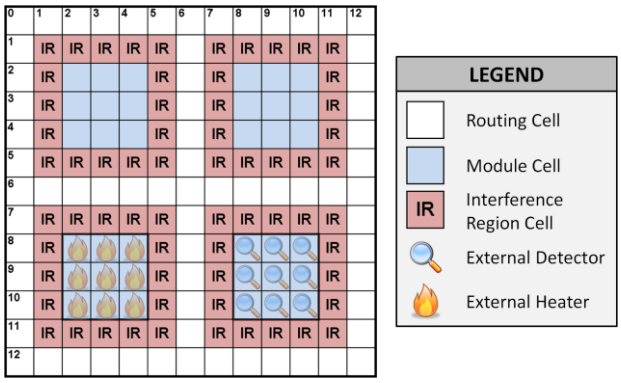


Figure 4. Virtual topology imposed onto a DMFB.

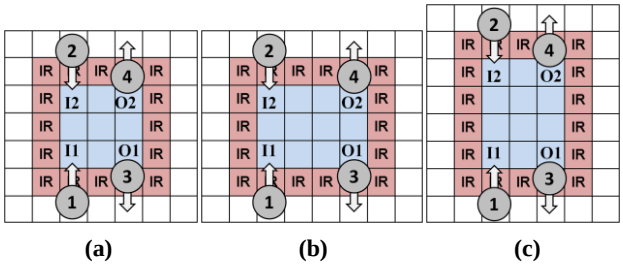


Figure 5. The entrance cells (I1/I2) and exit cells (O1/O2) of (a) a 3x3 module, (b) a 4x3 module and (c) a 3x4 module.

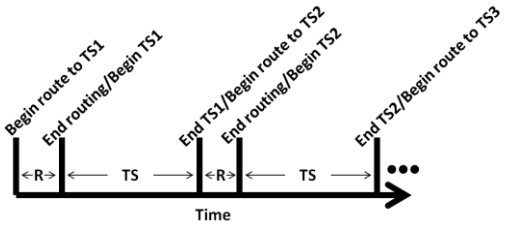


Figure 6. An assay time-line showing that each fixed time-step (TS) is interleaved with a variable-length routing phase (R).

As seen in **Figure 6**, time-step stages of assay operations are interleaved with routing stages until the entire schedule has been processed. A *time-step* is the basic, minimum-resolution unit of time used to schedule microfluidic operations; time-steps usually last one or two seconds and are fixed in length for the duration of the assay. The routing stages are variable in length, depending on the routes that are generated, and can even be instantaneous if no droplets are being routed between time-steps.

Droplets are required to enter/exit a module at one of the two entrances/exits. When a droplet travels to a new module, it must enter the module during the routing phase at one of the module-entrance cells and wait until the time-step officially begins. The droplet is then processed (e.g. split, mixed, stored) during the time-step phase. If a droplet leaves the module after the current time-step, it must position itself at one of the module-exit cells before the end of the time-step. In **Figure 5(a)** droplets 1 and 2 (D1/D2) enter a module to be processed while droplets 3 and 4 (D3/D4) exit to be processed elsewhere. If D1 and D2 arrive before D3 and/or D4 exit, there will be no conflict since the entrance and exit cells are sufficiently spaced to avoid droplet interference. When the time-step begins, D1 and D2 can move freely within the module, as D3 and D4 are at their respective destinations. This synchronization scheme prevents inter-module deadlocks because there is always an open spot at the destination module's entrances for every incoming droplet at every module.

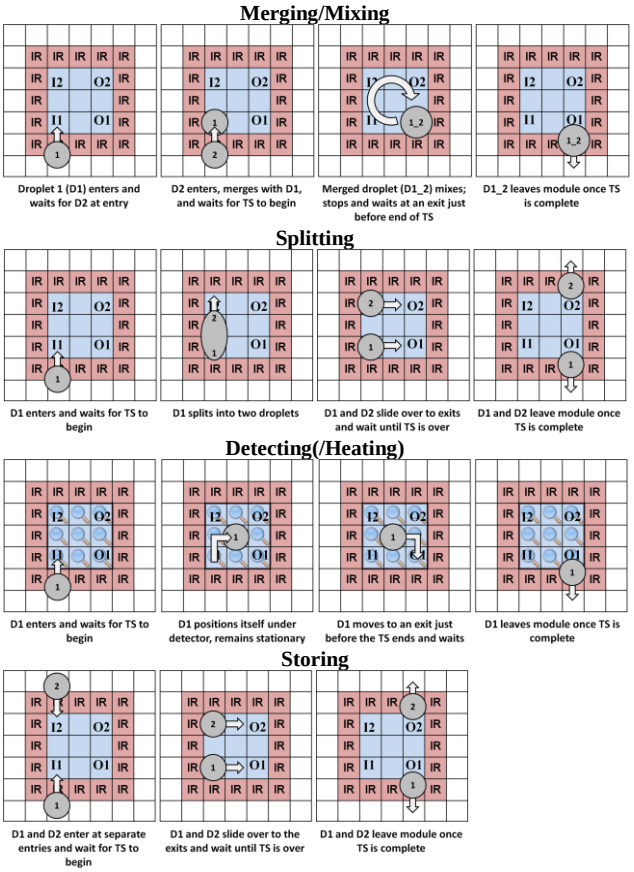


Figure 7. Intra-module droplet processing/routing for microfluidic operations.

Figure 7 shows how a module can perform each assay operation. For each operation, the droplet(s) enters at one of the entrance cells and then waits for the time-step to begin. When the time-step begins, any droplets that were waiting in the exit cells are now gone, and thus, any remaining droplets in the module are free to move about the entire module to perform an operation. If the droplet(s) leaves the module at the end of the time-step, it moves to an exit cell before the time-step ends. Once the time-step is complete, during the subsequent routing stage, the droplet(s) exits the module. If a droplet is scheduled to begin a new operation in the same module at the next time-step, it maneuvers itself to an entrance cell before the time-step ends (not shown in **Figure 7**); this eliminates the need for a droplet to exit and then re-enter the same module.

4. FAST ONLINE SYNTHESIS

In this section, we show how the virtual topology presented in Section 3 can be leveraged to create fast online synthesis methods for scheduling, placement and routing.

4.1 Scheduling

We use list-scheduling (LS) [24] with several extensions to schedule assays. LS is a greedy, constructive algorithm in which each operation (node) in an assay (DAG) is scheduled exactly once. LS is much faster than iterative improvement algorithms, which randomly compute numerous schedules [14][19][24] or optimal algorithms based on integer linear programming [24]; however, these approaches generally do produce higher quality schedules than LS.

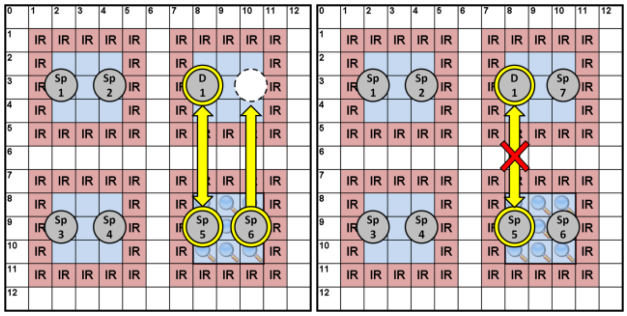


Figure 8. Two DMFB scenarios with droplets that are going to be split (Sp) or detected (D) during the next time-step. In (a), Sp6 can move and occupy the open space in another module, allowing D1 and Sp5 to swap so D1 can be detected in the detect-module. In (b), there is no way to isolate a single droplet and since no droplets will be mixed next time-step, the assay cannot continue.

4.1.1 Definitions and Constraints

As mentioned in Section 1.2, an assay is given to the scheduler in the form of a DAG, $G = (V, E)$, where the vertices (V) and edges (E) represent assay operations and operation dependencies, respectively. If the given DMFB is an $a_x \times a_y$ array of cells and each module is $m_x \times m_y$ cells, then the total number of modules, N_m, can be calculated as seen in **Equation 1**. We add 3 to the module dimensions to encapsulate the IR cells on all sides and the routing cells to the right (for the X dimension) and bottom (for the Y dimension) of each module (see **Figure 4**).

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

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

$$\sum_{i=1}^{numModTypes} availMods[i] = N_m \quad (2)$$

We define s_m to be the number of droplets a module can store and d_{max} to be the maximum number of droplets we allow on the DMFB during any time-step. Since each module has 2 entrance and 2 exit cells, a module can store 2 droplets during a time-step (i.e. $s_m = 2$). Consider **Figure 8(a)** in which all but one of the modules is at maximum capacity. Since the northeast module has room for one droplet, droplets can be shuffled around so that any single droplet on the array can be isolated in any chamber, allowing the assay to continue. However, if all modules are at maximum capacity (**Figure 8(b)**), then deadlock may arise because it is impossible to process more operations unless some of the droplets are scheduled to output or mix with each other next time-step. To reduce the likelihood of scheduling deadlock, we set the maximum number of droplets permitted on the DMFB during any time-step (d_{max}) as follows:

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

Lastly, I is a set of all the input ports and keeps track of when input ports have been scheduled for dispensing.

4.1.2 Scheduling Algorithm

```
1  Given sequencing graph: G = (V, E)
2  Given resource constraints: availMods[numModTypes], I, s_m, d_max
4  Assign priorities for all nodes: v ∈ V  ∀v based on CPP
5  Find candidate operations: C = {v_i ∈ V: Type(v_j) = input, ∀j: (v_j, v_i) ∈ E}
6  Unfinished operations: UF = ∅
7   TimeStep ts = 0;
8
9  while (all candidate operations are scheduled: C = ∅) {
10   if (uf.endTS = ts, ∀uf : uf ∈ UF)
11      Remove uf from unfinished ops UF;
12      availMods[uf.modType]++;
13   end if
14
15   Sort candidate ops C in ascending order of CPP;
16   for (∀c : c ∈ C)
17      if (CanSchedule(c) = true)
18         Add c to unfinished ops UF, remove from candidate ops C;
19         c.SetAsScheduled(ts, ts + c.duration, SelectModType(c));
20         availMods[c.ModType]--;
21
22         for (∀p : p ∈ c.parents)
23            if (Type(p) = input)
24               SetAsScheduled(ts − dispenseTime, ts, SelectInput(p));
25            else if (p.end < ts)
26               Create new storage node s;
27               s.SetAsScheduled(p.end, ts, NULL);
28               Insert s between p and c in G;
29         end ∀p for
30
31         for ( ∀cc: cc ∈ c.children)
32            if (ccp.scheduled = true, ∀ccp: ccp ∈ cc.parents)
33               Add cc to candidate ops C;
34      end canSchedule(c) if
35   end ∀c for
36
37   for (int i = 0; i < [dropsBeingStored(ts) ÷ s_m] )
38      Create new storage-holder node sh;
39      sh.SetAsScheduled(ts, ts + 1, SelectModType(sh));
40      availMods[sh.ModType]--;
41   end for
42   ts++;
43 end while
```

Figure 9. Pseudocode for our list scheduler algorithm.

We present pseudocode for our list-scheduling (LS) algorithm in **Figure 9**. Before scheduling begins, LS assigns a priority to each node (V) in the assay (G) based on that node's critical path length. The *critical path priority* (CPP) for any node n is the largest summation of time-steps from n (including n) to any output in n's fan-out. Finally, before LS begins scheduling, all nodes with only dispense (input) parents are added to the candidate list and the scheduling time-step is set to 0.

Lines 9-43 describe the main scheduling loop which continues until all operations are scheduled; each iteration through this loop, as many operations as possible are scheduled for the current time-step before incrementing to the next scheduling time-step (*Line 42*). As mentioned in the previous section, $availMods[\,]$ holds the available number of free modules for the current time-step. As a new time-step begins, operations that are finishing relinquish the modules they were using to the scheduler (*Lines 10-13*).

Next, the candidate nodes are sorted (*Line 15*) so that they are examined in ascending order (*Lines 16-35*) based on their CPP. Each candidate is checked to see if it can be scheduled by the $CanSchedule(\,)$ function in *Line 17*. $CanSchedule(c)$ returns true if (1) all $c's$ dispense parents have free input ports of the proper fluid-type in the previous time-steps, (2) all $c's$ non-dispense parents have been scheduled to complete before the current time-step, (3) there is a free module to process c, and (4) processing c does not violate **Equation 3**. If c can be scheduled, it is $SetAsScheduled(\,)$ with a starting and ending time-step and

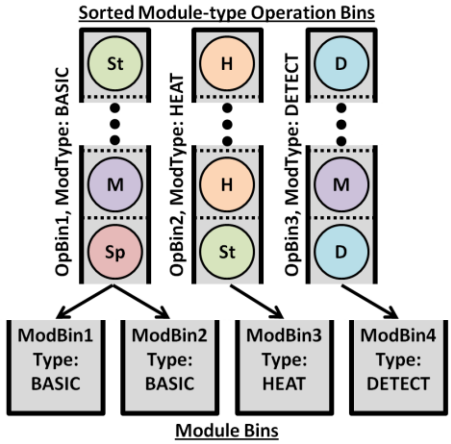


Figure 10. Left-edge algorithm binding operations to modules.

module-type. $SelectModType(c)$ selects a module type based on $c's$ operation-type and always selects a basic module, if possible, to conserve the special modules.

Once c is scheduled, *Lines 22-29* examine $c's$ parents. If a parent is an input, it is scheduled (inputs are not scheduled until a non-input child is scheduled). For non-dispense parents, a storage node is created and inserted between the parent (p) and child (c) if the parent was scheduled to complete more than one time-step before c was scheduled to begin. In *Lines 31-33*, any of $c's$ children whose parents are all scheduled are added to the candidate list.

Lastly, since s_m storage nodes (each node represents 1 droplet) can be stored in a single module, a storage-holder node is added to G for each set of s_m droplets being stored at the current time-step. Each storage-holder node is scheduled for 1 time-step, even if the droplets they are storing are stored for a longer period of time. This is done to prevent specialized modules from storing droplets for long periods of time; this way, a droplet stored in a specialized module can be moved to a basic module if a specialized operation (e.g., heat or detect) requires the specialized module.

4.2 Placement

Microfluidic placement is NP-complete [25]; the virtual topology limits the reconfigurable capabilities of the DMFB by pre-placing the location of modules. In our framework, operations are bound to pre-placed modules in accordance with the schedule that has been computed a-priori. The scheduler assigns operations to module types (e.g., basic or specialized), but does not select a specific module for each operation; this is the job of the binder.

4.2.1 Binding Algorithm

Our binder is based on the left-edge algorithm, which has been used in the past for register allocation and track assignment in channel routing [11]. The left-edge algorithm has an $O(|V|^2)$ time complexity, where $|V|$ is the number of assay operations.

Figure 10 shows how the left-edge algorithm binds operations to modules for the DMFB shown in **Figure 4**. **Figure 11** provides pseudocode for our binding algorithm and can be followed (up to *Line 20*) in the example. A *fixed-module bin* is created (*Line 2*) for each module in the virtual topology; $N_m = 4$ for this example. Then, each operation is placed into an *operation-bin* based on the module-type it was assigned during scheduling (*Lines 9-11*). Next, the operations in each bin are sorted in ascending order based on their start time (*Lines 13*).

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

Figure 11. Pseudocode for our left-edge-based binding algorithm. NOTE: Only module binding is shown; input and output binding is left out for brevity.

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

Lines 23-40 describe how storage operations are bound. Note that storage operations are not placed into the operation bins in *Lines 9-11*, and thus, were not bound to specific modules; however, the storage-holder nodes were bound in *Lines 17-20*. As mentioned in Section 4.1.2, a storage-holder node was created in the scheduling phase for each set of s_m droplets being stored each time-step (i.e. $\lceil st_t/s_m \rceil$ 1-time-step storage-holder nodes are created at time-step t, where st_t is the number of droplets being stored at t). Storage-holder nodes are created so that storage nodes can be broken into a number of smaller, contiguous storage nodes to prevent rare modules (e.g. detect modules) from being tied up as storage.

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

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

4.3 Routing

To complete the synthesis flow, we use a simplified version of an existing droplet router by Roy [20]. We created a number of routing methods that restricted routes to the cells in between modules, but found that Roy's maze-routing approach produced shorter routes in only a few more milliseconds of computation time compared to the alternatives. As in Roy's router, we use Soukup's fast maze router [21] to produce sequential routes for droplets and then compact the routes together, adding stalls in the middle of the routes to avoid droplet interference.

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

4.3.1 Routing Algorithm Overview

We provide pseudocode for the simplified Roy routing algorithm in **Figure 12** and **Figure 13**. The router receives a scheduled and placed DAG, G, from the placer. Throughout the routing process, all droplets in motion must maintain static and dynamic spacing constraints to prevent interference, as shown in **Figure 14**.

Instead of having a module-type or fluid-type (for dispense nodes), each node now has a reference to a specific module or I/O port. An empty set of routes is created which holds a separate route for each droplet (*Line 3*). A *route* is composed of a set of *routing points* that specify the droplet's location each cycle. The operations of G are copied into a new list, ops, and sorted first by ascending start time; operations with the same start time are then sorted by ascending end time (*Line 5*).

Starting at time-step 0 (*Line 6*), droplet routes are computed one routing sub-problem at a time. As seen in **Figure 6**, a *routing sub-problem* (or phase) is the problem of routing a number of droplets from their source (input or module) to their destination (module or output); routing sub-problems occur between the end of one time-step and the beginning of the subsequent time-step. Each iteration of the loop in *Lines 8-15* computes one routing sub-problem.

For a specific sub-problem, individual routes are first computed for each droplet using Soukup's fast maze routing algorithm (*Line*

```
1  Given a scheduled and placed sequencing graph G = (V, E)
2  Assay operations: ops = V
3  List of routes for each droplet for entire assay: routes = ∅
4
5  Sort ops by start time first, by stop time second, in ascending order;
6  TimeStep ts = 0;
7
8  while (! ops.empty)
9     subProbRoutes = GetIndivSoukupRoutes(ops, ts);
10    CompactRoutes(subProbRoutes);
11    Append routes in subProbRoutes to routes;
12    ts++;
13    if (o.end ≤ ts, ∀o : o ∈ ops)
14       Remove o from ops;
15 end while
```

Figure 12. Pseudocode for the simplified Roy routing algorithm.

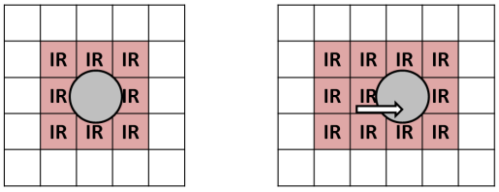
```
1  Given list of individual routes for each sub-problem droplet: subProbRoutes
2  List of compacted routes: compactedRoutes = ∅
3  Sort subProbRoutes in descending order by length;
4
5  for (∀r : r ∈ subProbRoutes)
6     bool routeMustWait = false;
7     RouteCycle i = 0;
8     while (i < r.size AND r.size > 0)
9        for (∀cr : cr ∈ compactedRoutes)
10          if(IRV(r[i], cr[i − 1]) OR IRV(r[i], cr[i + 1]) OR
11             IRV(cr[i], r[i − 1]) OR IRV(cr[i], r[i + 1]) OR
12             IRV(r[i], cr[i]) OR IRV(r[i − 1], cr[i − 1]))
13             routeMustWait = true, jump to line 17;
14          end if
15       end ∀cr for
16
17       if (routeMustWait = true)
18          Add a single-cycle stall to r at r[i − 2];
19          routeMustWait = false;
20       else
21          i++;
22       end if
23    end while
24    Add r to compactedRoutes;
25 end ∀r for
```

Figure 13. Pseudocode for the CompactRoutes( ) function.

9) [21]. Soukup's maze router works by routing around blockages; it routes straight to its destination until it hits a blockage, at which point it attempts to route around it.

A *persisting module* is defined as a module that starts before and ends after time-step ts. For the entire sub-problem at time-step ts, $ts's$ persisting modules (including interference regions) are marked as blockages for Soukup's algorithm. Let us define D_{ts} as the set of droplets to be routed during the sub-problem at time-step ts. Then, for each sub-problem droplet $d_i \in D_{ts}$, the algorithm sets the source and destination cells, along with an 8-cell interference region surrounding them, as blockages for each sub-problem droplet $d_j \in D_{ts}, d_j \neq d_i$. The blockages from the persisting modules remain persistent throughout the sub-problem, whereas, the blockages from sources and destinations vary depending on which droplet is currently being routed.

The result of Soukup's routing (*Line 9*) is a list containing a route for each droplet. The routes have no stalls and are not concurrent (i.e. the droplets may interfere with each other if routed at the same time). Thus, once the individual, sequential routes are computed, they are *compacted* so droplets can be routed concurrently (*Line 10*, explained in the next subsection). Once the sub-problem routes are compacted, they are appended to the matching droplet routes for the entire assay (*Line 11*). Finally, the time-step (ts) is incremented (*Line 12*) and any operations finished before ts are removed (*Lines 13-14*). When there are no more operations to route to, routing is complete.



(a) **(b)**

Figure 14. The interference region (IR) for a droplet at (a) the beginning and (b) end of a droplet-actuation cycle.

4.3.2 Route Compaction

Route compaction is the process of taking a number of sequential routes and causing the droplets to move in parallel at the same time; however, the original routes are not created with concern to other droplet routes and caution must be taken when compacting to prevent routes from intersecting in time and space. When compacting routes, droplets may not enter any cell that is adjacent to any other droplet or the droplets will interfere (merge) with one another. To prevent this, a static interference region (IR) is created around each droplet at the beginning of each droplet-actuation cycle, as seen in **Figure 14(a)**. As a droplet moves from one cell to the next, the IR is stretched dynamically to include the union of the static IRs of the beginning and end cells (see **Figure 14(b)**). In general, a droplet may not enter any other droplet's IR while routing. Static and dynamic droplet interference rules are formally defined in ref. [27].

Figure 13 shows pseudocode for the $CompactRoutes( )$ function (**Figure 12**, *Line 10*). It receives the list of individual routes ($subProbRoutes$) as described in the last section. The routes are sorted in descending order so that the longest route is processed first. *Lines 5-25* proceed to compact these routes one-at-a-time. The index of a route now represents time (i.e. $r[0]$ describes the xy-coordinate of a droplet during the first cycle of this routing phase). Thus, when determining if route r can be compacted, we check each cycle i of $r's$ route against any previously-compacted route, cr, at cycle i. *Lines 10-12* compute the dynamic and static droplet restraints described in **Figure 14** and ref. [27]. If an interference region violation (IRV) is found between the two routes, then r cannot be compacted as is. To remedy this, a single stall is added (*Line 18*) 2 routing cycles before the IRV is discovered until the algorithm can successfully compact r.

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

Roy's router attempts to recover from deadlock by moving one of the droplets backward [20]. We simplify the process by taking advantage of our virtual topology. With our module synchronization, described in Section 3.1, droplets have designated sources (module exits) and destinations (module entries) that do not interfere with any other sources and destinations in a given time-step (i.e. a droplet source will never interfere with another droplet's destination). Thus, a droplet can stay at its source as long as necessary, until all other droplets are safely off its path, and then commence its route. By employing this method, we are guaranteed to avoid deadlock.

With this in mind, the router keeps track of the number of stalls added to any route r. If the number of stalls added to route r reaches some threshold, $stallThresh$, all of the stalls added to

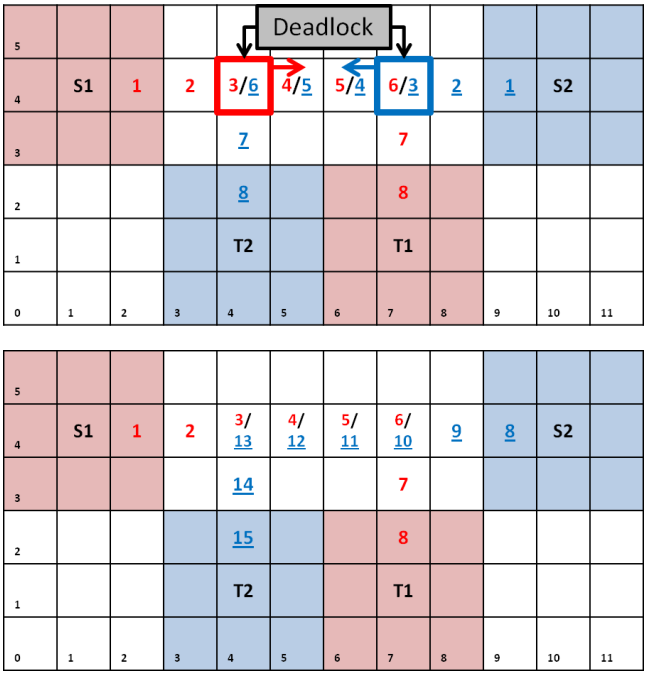


Figure 15. Droplets 1 and 2 are traveling from source 1 and 2 (S1/S2) to target 1 and 2 (T1/T2), respectively. The red and blue (blue also underlined for clarity) numbers are time-stamps for droplets 1 and 2, respectively). The top scenario shows that deadlock can occur when routes 1 and 2 are compacted and stalls are added mid-route. The bottom scenario shows that both routes are safely completed if droplet 2 stalls at its source location until droplet 1 is safely out of the way.

any route thus far in the sub-problem are removed. Then, the entire sub-problem is compacted again, except this time, stalls are added to the beginning of the routes instead of the middle (i.e. the compaction algorithm in **Figure 13** is applied, except *Line 18* adds a stall at $r[0]$ instead of $r[i-2]$). In this case, droplets do not leave the safety of their source cell until they are guaranteed an unobstructed path in space and time to their destination.

Consider **Figure 15** in which droplets 1 and 2 are being routed from their sources (S1 and S2) to their target cells (T1 and T2). As seen in the top scenario, if the routes start at the same time, deadlock will occur at cycle 3 as droplet 1, at cell (4, 4), and droplet 2, at cell (7, 4), cannot move forward without merging. No amount of mid-route stalls will resolve this deadlock since they are heading straight toward each other; it is not a matter of allowing one droplet to pass. The bottom scenario shows that if droplet 2 is allowed to stay in its source until droplet 1 is safely off its route, droplet 1 can reach its target. Since the cells around S2 are considered as blockages to droplet 1, droplet 2 is safe to wait at S2 as long as necessary because droplet 1 will never attempt to pass through that area, even if its destination is to the east of S2.

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

5. EXPERIMENTS

Our experiments compare the online DMFB synthesis flow described in this paper, with a prototypical offline synthesis flow based on algorithms described in prior literature. The goal of this experiment is to demonstrate that our online flow can achieve competitive solutions in terms of total assay execution time, while maintaining a low algorithmic runtime overhead.

5.1 Benchmarks

We used three standard benchmarks: PCR, in-vitro diagnostics, and a protein assay, whose DAGs have been made publicly available by researchers at Duke University [23]; we also used the provided module libraries to obtain operation timings. We used a 2x4 mixer (3s) for all PCR mixing operations. For the protein assay, we used a 2x4 diluter (5s) and 2x4 mixer (3s) for all dilute and mixing operations, respectively; all 2-input, 2-output dilute operations in the protein assay were implemented using a mix operation, followed by a split operation, which took 5s in total.

In-vitro diagnostics is actually a family of assays that mixes and detects up to 4 samples with 4 reagents (e.g., up to 16 mix-and-detect operations). We use the 5 in-vitro assays, along with mixing and detection times, as listed in Table 1 of ref. [22].

We assume a droplet actuation frequency of 100 Hz [31].

5.2 Offline Synthesis Flow (Baseline)

Our offline synthesis flow is based on algorithms that have been published previously for DMFB synthesis; these algorithms have generally been optimized for solution quality, not for runtime. We used a genetic algorithm to perform scheduling [24], simulated annealing to perform module placement [25] and our implementation of Roy's algorithm for droplet routing [20]. We chose Roy's algorithm for droplet routing because it achieved short routes and never failed for any of the routing test cases. Roy's router did not fail in any of our experiments. The offline algorithms mentioned in this section are freely available as open source [1].

5.3 Online Synthesis Flow

The online synthesis flow is described in Section 4 of this paper. The use of the virtual topology eliminates the need for a placer. The flow includes three main steps: scheduling, binding, and routing; the same router was used in both the online and offline flow, as it obtained good quality results and fast runtimes. The online algorithms described in this paper are freely available as open source [1].

5.4 Implementation Details

The offline and online synthesis flows were implemented in C++ using the University of California, Riverside's (UCR's) DMFB Synthesis Framework [7]. We evaluated their performance on two computers available in our laboratory. The first is an Inforce SYS9402-01 development board, with a 1GHz Intel Atom™ E638 processor and 512MB RAM, running TimeSys 11 Linux. This is a relatively low-end embedded system that could be integrated to control a DMFB, e.g., to provide a platform for portable healthcare diagnostics.

The second is a 64-bit Windows 7 desktop PC, with 4GB of RAM and an Intel Core i7™ CPU operating at 2.8GHz. This platform represents a typical use case for a controlled laboratory setting.

5.5 Results and Discussion

Table 1 compares the genetic scheduler and list scheduler; consistent with prior work [24], the list scheduler produces schedules that are several time steps longer than the genetic algorithm in most cases; however, this comes at a significant overhead in terms of runtime. On both platforms, the list scheduler was able to schedule all assays in less than 30ms.

Table 1. Results showing the schedule lengths and computation times (on Intel i7 and Atom processors) for genetic and list scheduling.

Genetic Scheduling vs. List Scheduling						
Benchmark	Genetic Scheduling			List Scheduling		
	i7 (ms)	Atom (ms)	Sched (s)	i7 (ms)	Atom (ms)	Sched (s)
PCR	395	2,621	12	1	1	12
InVitro_1	665	4,475	15	0	2	15
InVitro_2	1,293	8,122	17	0	5	19
InVitro_3	1,990	13,156	19	1	13	23
InVitro_4	3,541	22,376	23	1	17	26
InVitro_5	5,744	39,410	31	2	27	35
Protein	3,297	22,334	110	3	14	116

Table 2. Results showing computation times (on Intel i7 and Atom processors) for simulated annealing and module binding.

Placement vs. Binding						
Benchmark	Simulated Annealing Placer			Module Binder		
	i7 (ms)	Atom (ms)	%Cells Used	i7 (ms)	Atom (ms)	%Cells Used
PCR	16	200	16	0	0	20
InVitro_1	621	12,843	16	0	0	24
InVitro_2	105,138	141,177	21	0	0	28
InVitro_3	72,311	506,767	29	0	0	36
InVitro_4	19,789	3,317,571	32	0	0	45
InVitro_5	74,899	1,399,936	36	0	0	48
Protein	4,867,220	79,531,695	29	0	4	46

Table 3. Results showing route lengths and computation times (on Intel i7 and Atom processors) for the classic offline flow (GA/SA) and our proposed online flow (List/Binding).

Simplified Roy Routing						
Benchmark	GA Scheduling - SA Placement Flow			List Scheduling - Module Binding Flow		
	i7 (ms)	Atom (ms)	# Routing Cycles / # Sub-problems	i7 (ms)	Atom (ms)	# Routing Cycles / # Sub-problems
PCR	0	2	78 / 4	0	6	56 / 4
InVitro_1	0	2	135 / 9	0	3	111 / 9
InVitro_2	0	4	180 / 11	0	7	167 / 12
InVitro_3	0	10	207 / 15	0	12	209 / 16
InVitro_4	0	7	234 / 15	1	19	299 / 19
InVitro_5	1	9	342 / 22	1	26	351 / 24
Protein	5	32	1212 / 71	3	76	638 / 45

Table 2 compares the simulated annealing-based placer to the online binder proposed in this paper. The binder runs several orders of magnitude faster than the placer on both platforms; in most cases, the runtime of the binder was faster than the resolution of the function that we use to measure execution time (e.g., faster than 0.5 ms). As we will see shortly, the impact on total assay execution time is negligible.

Table 3 reports the results of the router; due to inferior schedule qualities, the online synthesis flow has more routing sub-problems to solve than the offline flow. **Table 3** sums the total number of cycles (1 cycle = .01s with 100Hz droplet-actuation frequency) required for droplet routing across all sub-problems. It is important to note that even when the schedules are identical, the placer will yield a different sub-problem than the virtual topology. We ignore sub-problems where all droplet routes start and stop at the same module; this explains, for example, why the online flow has 45 sub-problems for Protein, while the offline router has 71.

Table 3 shows that the runtimes of the router are a little bit slower for our virtual topology, compared to the placer; however, the algorithmic runtime advantages of the list scheduler and binder compared to the genetic scheduler and placer far outweigh the disparity between the runtimes of the routers.

Table 4. Results showing the assay time (AT, i.e. the schedule and route lengths) and computation times (CT) for the entire offline flow on the Intel Atom processor.

GA Scheduling - SA Placement - Simp. Roy Routing Offline Flow						
Benchmark	Schedule		Placement	Routing		Total
	AT (s)	CT (s)	CT (s)	AT (s)	CT (s)	(AT + CT) (s)
PCR	12	3	0	1	0	16
InVitro_1	15	4	13	1	0	34
InVitro_2	17	8	141	2	0	168
InVitro_3	19	13	507	2	0	541
InVitro_4	23	22	3,318	2	0	3,365
InVitro_5	31	39	1,400	3	0	1,474
Protein	110	22	79,532	12	0	79,676

Table 5. Results showing the assay time (AT, i.e. the schedule and route lengths) and computation times (CT) for the entire online flow on the Intel Atom processor.

List Scheduling - Module Binding - Simp. Roy Routing Online Flow						
Benchmark	Schedule		Placement	Routing		Total
	AT (s)	CT (s)	CT (s)	AT (s)	CT (s)	(AT + CT) (s)
PCR	12	0	0	1	0	13
InVitro_1	15	0	0	1	0	16
InVitro_2	19	0	0	2	0	21
InVitro_3	23	0	0	2	0	25
InVitro_4	26	0	0	3	0	29
InVitro_5	35	0	0	4	0	39
Protein	116	0	0	6	0	122

Tables 4 and **Table 5** compare the total computation time (CT) and the assay execution time (AT) for both flows on the Intel Atom™ processor; please note that the times reported here are in seconds (s), whereas the times reported in **Tables 1-3** are in milliseconds (ms). We can see that the total runtime, AT+CT, is significantly lower for the online flow than for the offline flow, except for PCR, which is a very small benchmark. The most dramatic results that favor the online synthesis flow are for the largest benchmarks, InVitro_4 and 5 and Protein, where the runtimes are reduced by two orders of magnitude.

6. CONCLUSION

The online synthesis flow introduced in this paper can run in real-time on a low-cost embedded processor, as typified by the Intel Atom™ used in our experiments. Empirically, this work has shown that a virtual topology coupled with a simple binding algorithm can be nearly as effective as longer-running placement algorithms based on simulated annealing. Using this approach, the synthesis flow spends most of its time on scheduling, which is the step that has the most significant impact on assay execution time.

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

7. ACKNOWLEDGMENTS

This work was supported in part by NSF Grant CNS-1035603. Daniel Grissom was supported by an NSF Graduate Research Fellowship.

8. REFERENCES

[1] www.microfluidics.cs.ucr.edu

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

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

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

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

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

[7] D. Grissom, et al., "A digital microfluidic biochip synthesis framework," in *Proc. VLSI-SOC*, Santa Cruz, CA, 2012.

[8] D. Grissom and P. Brisk, "Path scheduling on digital microfluidic biochips," in *Proc. DAC*, San Francisco, CA, 2012, pp. 26-35.

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

[10] T. Ho, K. Chakrabarty, and P. Pop, "Digital microfluidic biochips: recent research and emerging challenges," in *Proc. CODES+ISSS*, Taipei, Taiwan, 2011, pp. 335-343.

[11] F. J. Kurdahi and A. C. Parker, "REAL: a program for REgister ALlocation," in *Proc. ACM/IEEE DAC*, Miami, FL, 1987, pp. 210- 215.

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

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

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

[15] J.H.Noh, J. Noh, E. Kreit, J. Heikenfeld, and P. Rack, "Toward active-matrix lab-on-a-chip: programmable electrofluidic control enabled by arrayed oxide thin film transistors," *Lab Chip*, vol. 12, no. 2, pp. 353-360, Dec. 2011.

[16] K. O' Neal, D. Grissom, and P. Brisk, "Force-directed list scheduling for digital microfluidic biochips," in *Proc. VLSI-SOC*, Santa Cruz, CA, 2012.

[17] P. Paik, V. Pamula, and R. Fair, "Rapid droplet mixers for digital microfluidic systems," *Lab Chip*, vol. 3, no. 4, pp. 253-259, Sep. 2003.

[18] M. G. Pollack, A.D. Shenderov, and R. B. Fair, "Electrowetting-based actuation of droplets for integrated microfluidics," *Lab Chip*, vol. 2, no. 2, pp. 96-101, 2002.

[19] A. J. Ricketts, K. Irick, N. Vijaykrishnan, and M. J. Irwin, "Priority scheduling in digital microfluidics-based biochips," in *Proc. DATE*, Munich, Germany, 2006, pp. 329-334.

[20] P. Roy, H. Rahaman, and P. Dasgupta, "A novel droplet routing algorithm for digital microfluidic biochips," in *Proc. GLSVLSI*, Providence, RI, 2010, pp. 441-446.

[21] J. Soukup, "Fast maze router," in *Proc. DAC*, Las Vegas, NV, 1978, pp. 100-102.

[22] F Su and K. Chakrabarty, "Architectural-level synthesis of digital microfluidics-based biochips," in *Proc. ICCAD*, San Jose, CA, 2004, pp. 223-228.

[23] F. Su and K. Chakrabarty, "Benchmarks for digital microfluidic biochip design and synthesis," *Duke Univ. Dept. ECE*, 2006: http://www.ee.duke.edu/~fs/Benchmark.pdf

[24] F. Su and K. Chakrabarty, "High-level synthesis of digital microfluidic biochips," *ACM J. Emerging Tech. Comput. Syst.*, vol. 3, no. 4, pp. 16.1-16.32, Jan. 2008.

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

[26] F. Su and K. Chakrabarty, "Unified high-level synthesis and module placement for defect-tolerant microfluidic biochips," in *Proc. DAC*, Anaheim, CA, 2005, pp. 825-830.

[27] F. Su, W. Hwang, and K. Chakrabarty, "Droplet routing in the synthesis of digital microfluidic biochips," in *Proc. DATE*, Munich, Germany, 2006, pp. 323-328.

[28] Z. Xiao, and E. F. Y. Young, "Placement and routing for cross-referencing digital microfluidic biochips," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 30, no. 7, pp. 1000-1010, Jul. 2011.

[29] T. Xu and K. Chakrabarty, "Broadcast electrode-addressing for pin-constrained multi-functional digital microfluidic biochips," in *Proc. DAC*, Anaheim, CA, 2008, pp. 173-178.

[30] T. Xu and K. Chakrabarty, "Integrated droplet routing in the synthesis of microfluidic biochips," in *Proc. DAC*, San Diego, CA, 2007, pp. 948-953.

[31] P-H. Yuh, C-L. Yang, and Y-W. Chang, "BioRoute: a network-flow-based routing algorithm for the synthesis of digital microfluidic biochips," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 27, no. 11, pp. 1928-1941, Nov. 2008.

[32] P-H. Yuh, C-L. Yang, and Y-W. Chang, "Placement of defect-tolerant digital microfluidic biochips using the T-tree formulation," *ACM J. Emerging Tech. Comput. Syst.*, vol. 3, no. 3, pp. 13.1-13.32, Nov. 2007.