# Force-directed List Scheduling for Digital Microfluidic Biochips

Kenneth O'Neal, Daniel Grissom, Philip Brisk

Department of Computer Science and Engineering
University of California, Riverside
Riverside, CA 92521

***Abstract*—We introduce a *Force-directed List Scheduling (FDLS)* algorithm for resource-constrained assay compilation targeting *Digital Microfluidic Biochips (DMFBs)*. This algorithm has been used in the past for high-level synthesis of digital signal processing systems, and is now applied to DMFB synthesis. The results show improvements compared to *List Scheduling (LS)* and *Path Scheduling (PS)*, the most efficient heuristics that have been proposed, to date, for DMFBs. FDLS was also competitive with longer-running iterative improvement DMFB scheduling algorithms based on genetic algorithms.**

***Keywords-Digital Microfluidic Biochip (DMFB), Force-directed List Scheduling***

## I. INTRODUCTION

We introduce a *Force-Directed List Scheduling (FDLS)* algorithm for *Digital Microfluidic Biochips (DMFBs)*. FDLS produces shorter schedules than *List Scheduling (LS)* [11] and *Path Scheduling (PS)* [3] in some cases. LS and PS are two efficient polynomial-time scheduling heuristics proposed for DMFBs.

Compared to benchtop chemistry, DMFBs offer the benefits of miniaturization and automation: they use smaller chemical quantities and automate manual processes that have been subject to human error in the past. This reduces laboratory space, material cost, and the time to complete assays (biochemical protocols) by a significant margin.

A DMFB manipulates discrete droplets of liquid on a 2D grid using an actuation process called electrowetting on dielectric (EWoD) [8]. As shown in Fig. 1, there is a control electrode underneath each *cell* in the grid, and a single ground electrode on top of the grid. Activating the control electrode beneath a cell that stores a droplet holds the droplet in place; activating a control electrode in an adjacent cell induces droplet motion. Fig. 2 depicts the droplet operations that a DMFB can perform: transport, splitting, merging, and mixing, along with in-place droplet storage (not shown). Thus, a DMFB is a reconfigurable computing device, as different cells can perform different operations at different times during assay execution.

Fig. 3 depicts a compilation flow targeting a DMFB. Assays are specified as *directed acyclic graphs (DAGs)* [10]. Assay compilation involves three steps: (1) scheduling, (2) placement of operations on the DMFB (in time and in space), and (3) routing droplets between inputs, outputs, and operation locations [4]. This paper focuses on the scheduling step; in principle, any placer and/or router could be used in conjunction with our scheduler to complete the compilation flow.

The time of assay operations (e.g., mixing) is several orders of magnitude greater than the time to transport droplets. Thus, scheduling algorithms have a much greater impact on assay execution time than routing algorithms, as long as a legal route is found. This paper focuses exclusively on scheduling.

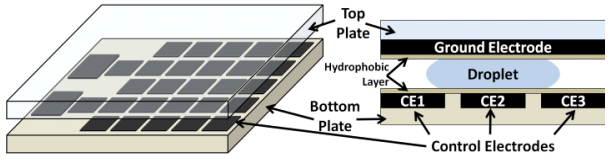


Figure 1. A DMFB with a 2D array of electrodes (left) and cross-sectional view of a DMFB (right).

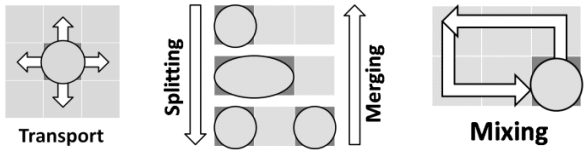


Figure 2. Basic microfluidic operations performed on 2D array of electrodes.

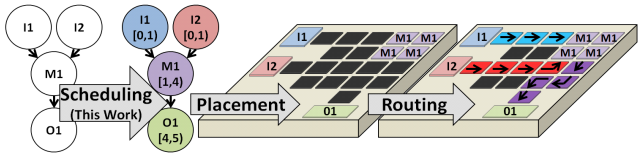


Figure 3. DMFB synthesis is composed of three, sequential, steps: operation scheduling, module placement and droplet routing.

## II. RELATED WORK

We are aware of just a handful of peer-reviewed papers that have studied the scheduling problem, to date. We are aware of two heuristics, *Modified List Scheduling (MLS)* [11] and *Path Scheduling (PS)* [3], two *genetic algorithms (GA-1, GA-2)* [9][11], and two optimal *integer linear programming (ILP)* formulations [2][11].

MLS [11] is an efficient polynomial-time greedy heuristic that runs in *O(nlogn)* time, where *n* is the number of vertices in the graph. MLS augments the standard *List Scheduling (LS)* algorithm [1] with a rescheduling step that is invoked to increase the likelihood of achieving a legal schedule while meeting resource constraints. *Force-Directed List Scheduling (FDLS)* is based on LS, not MLS, and borrows ideas from force-directed scheduling in high-level synthesis for digital systems [7][13] to prioritize vertices for greedy scheduling.

PS [3] is based on the observation that storing droplets on-chip reduces the availability of resources for other operations. This is important for assays with a large fanout in their DAG representation. PS prioritizes the different paths in a DAG, and, when given a choice, schedules operations from paths that have already started to execute, rather than starting to execute a new path. PS was shown to find legal schedules for graphs that LS was unable to schedule, due to large assay DAGs and DMFBs with limited resources.

The two GAs randomly generate schedules and converge over time to locally optimal solutions; GA-2 [9] offers improvements over GA-1 [11] for assays whose DAGs have multiple connected components. GA-1 and GA-2 generally produce better quality solutions than (M)LS and PS, but run significantly slower; FDLS produces schedules almost as short as those produced by GA-1 and GA-2 for most benchmarks that we considered. The ILP formulations [2][11] achieve optimal schedules; however, the algorithms that solve the ILP have exponential time complexities, unless it is eventually proven that P=NP.

Luo and Akella [5] analyzed the PCR assay [10] and developed a polynomial-time algorithm that schedules it optimally; in contrast, the other algorithms mentioned here are sufficiently general to handle any assay specified as a DAG.

Several papers perform scheduling in conjunction with other synthesis steps. Yu et al. [14] combine scheduling and placement into a 3D placement problem in space and time; they solve the problem using simulated annealing and a data structure (the T-tree) to represent the placement. Maftei et al. [6] solve scheduling, module selection, placement, and routing using Tabu Search. These iterative improvement algorithms are much like GA-1 and GA-2 in terms of quality and runtime.

## III. Scheduling Algorithms

### A. Preliminaries and Assumptions

Assay operations may occur anywhere on a DMFB. The execution time of mixing depends on the size of the mixer (e.g., 2x2, 2x3, 3x3, etc.); larger mixers are generally faster [10]. This complicates scheduling because: (1) scheduling cannot be decoupled from module selection; and (2) module selection cannot be decoupled from placement; if we assume that we start with a legal placement, increasing the size of a module may cause the placement to become illegal. We assume uniform module size as a simplifying assumption. It is not immediately clear how many concurrent assay operations can be supported in this context. In resource-constrained scheduling in high-level synthesis for digital systems [1], the number and type of each resource is provided as an input to the problem: e.g., 10 adders, 5 multipliers, etc.

A secondary complication is that DMFB resources perform both assay operations and storage; in contrast, high-level synthesis for digital systems decouples scheduling and storage allocation [1], since functional units and registers are two completely different resources.

In DMFB synthesis, all that we know about the DMFB is its length and width. Our scheduler divides the regions of the DMFB into *work modules*, which perform all assay and storage operations; we assume that work modules are packed as tightly as possible onto the device, while providing sufficient pathways for droplet routes. The number of work modules, $N$, is derived deterministically from the dimensions of the DMFB. Each work module can perform one assay operation (e.g., mixing, splitting, etc.) or store up to $k$ droplets at a time while ensuring legal spacing constraints [12] within the module; we use $k$ values of *2* and *4* in our experiments.

#### 1) Components: Work Modules and I/O

Many assays require specialized operations, e.g., heating and detection, which the DMFB alone cannot perform. External devices, e.g., heaters and detectors, are affixed to the DMFB to perform these operations. To simplify scheduling, we assume that these devices are affixed to *specialized work modules*, which can perform specialized assay operations that *general* work modules cannot; we assume that each work module supports at most one specialized operation.

I/O reservoirs perform specialized input and output operations, but no other operations (e.g., an input reservoir cannot perform mixing or splitting. Input reservoirs dispense fluids; we assume that the fluid connected to each input reservoir remains static during assay execution. Any fluid can be routed to a waste module for disposal. Our benchmarks [10] do not produce non-waste output for collection.

We use the generic term *component* to refer to a work module or I/O reservoir.

#### 2) Operation and Component Types

We associate an integer *type* with each assay operation and component; we use a *compatibility* function to determine whether a given component can execute an assay operation.

Let $D$ denote the number of specialized operations (e.g., heating, detection, etc.) supported by the DMFB; we assume that any assay will require a subset of these specialized operations; an assay that requires a specialized operation that is not supported by the DMFB cannot be scheduled.

Let $F$ denote the number of distinct fluids used by the assay; multiple droplets of the same fluid are *not* distinct. The DMFB provides at least one input reservoir per distinct fluid.

$T = \{0, 1, \ldots, D+F+1\}$ is the set of types, described next:

*General Types*: Type *0* refers to general operations (e.g., mixing, splitting, merging) that any work module can perform.

*Specialized Types*: Types *1…D* refer to specialized operations that require external devices (e.g., heating, detection); only specialized work modules can perform them.

*Fluid Input Types*: Types $D+1 \ldots D+F$ refer to dispense operations (fluid input). Only an input reservoir that contains a fluid of type $f$ can input that fluid type.

*Disposal Types*: Type $D+F+1$ is a generic output type, which any waste output can perform.

The assay is specified as a DAG $G = (V, E)$, and the set of components is $C = \{M, I, O\}$, where $M$ is the set of work modules (both general and specialized) and $I$ and $O$ are the respective sets of input and output reservoirs.

#### 3) Compatible Components and Operations

General work modules *only* perform Type *0* operations. A specialized work module of type $j$, $1 \leq j \leq D$, can perform an operation of Type *0* or $j$. An input reservoir can only dispense a fluid of Type $j$, $D+1 \leq j \leq D+F$; and an output reservoir can only perform disposal operations of Type $D+F+1$.

Let $f: V \cup C \rightarrow T$ be a function that associates a type with each assay operation and component. A DAG vertex $v \in V$ and a component $c \in C$ are *compatible*, denoted $v \preccurlyeq c$, if either of the following two conditions holds:

- $f(v) = 0$ and $0 \leq f(c) \leq D$, i.e., $v$ performs a general assay operation, and $c$ is a general or specialized work module; or
- $1 \leq f(v) \leq D+F+1$ and $f(c) = f(v)$, i.e., $v$ is a specialized assay operation or I/O operation, and $c$ is a specialized work module or I/O reservoir that can perform the operation.

Let $g: T \rightarrow \{T\}$ be a one-to-many function that returns the set of all component types that are compatible with a vertex of a given type. In other words, $g(f(v))$ is the set of all component types that are compatible with vertex $v \in V$; for a given DMFB, $g$ can be computed offline once and stored in a vector.

Let $h: V \rightarrow T$ be a function that associates a component type with each vertex; $h(v)$ denotes the type of the component onto which vertex $v \in V$ is scheduled.

#### 4) Scheduling Operations and Droplet Storage

The latency $L(u)$ is provided for each assay operation $u \in V$; our benchmarks specify assay operations in terms of seconds [10]. The schedule computes a start time $S(u)$ for $u$; the duration of the operation is the interval $[S(u), S(u) + L(u)]$.

Consider edge $(u, v) \in E$; if droplet routing time is negligible [11], then $S(v) \geq S(u) + L(u)$; otherwise, the schedule would not satisfy precedence constraints. If $S(v) > S(u) + L(u)$, then the droplet produced by operation $u$ must be stored for time interval $H(u) = [S(u) + L(u), S(v)]$, and possibly longer if $u$ has other successors whose start times are later than $S(v)$. Thus, some work module must be available to store $u$ during $H(u)$; recall that work modules can store up to $k$ droplets. For vertex $u \in V$, the storage time is:

$$H(u) = max\{S(v) | (u, v) \in E\} - (S(u) + L(u)). \quad (1)$$

$H(u) = 0$, for all *sinks*, i.e., DAG vertices with no successors.

### B. Problem Statement

Resource-constrained scheduling for DMFBs, as described here, is a constrained optimization problem; the corresponding decision problem is NP-complete. The problem is characterized as follows:

#### 1) Inputs

The inputs to the resource constrained scheduling problem for DMFBs are as follows:

- An assay specified as a DAG $G = (V, E)$;
- The latency of each operation: a function $L: V \rightarrow \{1, 2, \ldots\}$;
- The number of distinct fluids $F$;
- The set of component and operation types $T$; and
- A DMFB, characterized by the number of components of each type: $N = N_0 + N_1 + \ldots + N_{D+F+1}$.

#### 2) Objective

The objective is to compute a legal schedule that minimizes the total schedule length, i.e.,

$$Obj = min\{max_{u \in V}\{S(u) + L(u)\}\}. \quad (2)$$

The final vertex to finish in the schedule is guaranteed to be a sink, which will require no storage time.

#### 3) Legality

A legal schedule must satisfy the precedence constraint:

$$\forall (u, v) \in E, S(v) > S(u) + L(u). \quad (3)$$

Resource constraints are more complicated, and must be satisfied for each time-step. The schedule does not need to bind operations to specific resources; it simply needs to ensure that sufficient resources are available to perform the operations that have been scheduled.

Let $t$ be a time step in the schedule. Let $r_j(t)$ be the set of operations of type $j$ scheduled at time step $t$, and $h(t)$ be the set of operations stored at time step $t$, i.e.:

$$r_j(t) = \{u | u \in V, f(u) = j, S(u) < t < S(u) + L(u)\}; \text{ and} \quad (4)$$

$$h(t) = \{u | u \in V, S(u) + L(u) < t < S(u) + L(u) + H(u)\}. \quad (5)$$

The resource constraints for specialized assay operations and I/O operations are straightforward:

$$|r_j(t)| < N_j, 1 < j < D + F + 1, \quad (6)$$

i.e.,, the number of non-general operations of each type scheduled at each time step cannot exceed the number of available modules or I/O reservoirs of that type.

The resource constraints for general assay operations and storage are more complicated, as any work chamber can perform them. If we assume that criteria (6) is satisfied, then let $N_m(t)$ denote the number of available work modules for these operations:

$$N_m(t) = N_0 + \sum_{j=1}^{D} (N_j - |r_j(t)|). \quad (7)$$

Recall that each work module can store up to $k$ droplets. Then the final resource constraint for general operations and storage is

$$|r_0(t)| + \lceil |h(t)|/k \rceil \leq N_m. \quad (8)$$

### C. List Scheduling (LS) Implementation

LS [11] computes a priority function for each vertex in the DAG. Vertices that can be scheduled immediately (sources) are sorted in priority order using a priority queue. The main loop of the algorithm steps through the schedule, one time-step at a time, until all vertices are scheduled, or it becomes apparent that no feasible schedule is possible, given the resource constraints. At a given time step, a vertex is available if all of its predecessors have completed their operations.

LS maintains a vector $A[0 \ldots D+F+1]$, where $A[j]$ is the number of available resources of type $j$. Initially, $A[j] = N_j$, i.e., the total number of resources of type $j$ in the DMFB.

At each time step, the available vertices are processed in priority order. Let $u \in V$ be an available vertex. For each compatible component type $j \in g(f(u))$, at least one component of type $j$ is available for $v$ if $A[j] > 0$. If so, then $A[j]$ is decremented, and the scheduler sets $h(u) = j$ to remember the type of the resource onto which $v$ is scheduled; this way, the scheduler can free up a resource of the appropriate type when $v$ finishes its operation.

General assay operations (e.g., mixing, dilution, etc.) can execute on either general or specialized work modules. When both types of work module are available, we give preference to general work modules, because this increases the likelihood that a specialized assay operation can be scheduled in an upcoming time-step. If vertex $u \in V$ finishes its operation at the end of the current time step, the scheduler decrements $A[h(u)]$ to free up a resource of the appropriate type.

#### 1) Tracking Droplet Storage

A significant amount of bookkeeping is required to track which modules of each type store droplets. In particular, droplets may move between work chambers if doing so is advantageous. Examples of useful droplet movements include:

- Over time, droplets enter and leave chambers for storage. A situation may exist where modules $m_1$ and $m_2$ are initially required to store $n$ droplets in total, where $k < n < 2k$. At some future time step, the number of droplets stored in $m_1$ and $m_2$, denoted by $n_1$ and $n_2$ respectively, may be reduced and satisfy $n_1 + n_2 < k$. When this occurs, without loss of generality, all of the droplets stored in $m_1$ can be transported to $m_2$, freeing up $m_1$ for other assay operations.
- Suppose that droplets are stored in a specialized work module of type $j_1$, that $A[j_1] = 0$, i.e., no other modules of type $j_1$ are available, and that $A[j_2] > 0$ for some specialized work module type $j_2 \neq j_1$, i.e., at least one module of type $j_2$ is available. If a vertex $u \in V$ of type $f(u) = j_1$ is available for scheduling, and no vertex $v \in V$ of type $f(v) = j_2$ is available, then it is beneficial to move the droplets to an available module of type $j_2$; then $u$ can be scheduled immediately.

- If a general work chamber is freed because an operation finishes, then moving droplets from a specialized work chamber to the general chamber reduces competition for the specialized chamber.

Therefore, the scheduler must track the work module types that store each droplet in the system at each time-step. It is important to note that the scheduler does not bind droplets to specific work modules of a given type; it simply adjusts the available modules of each type accordingly as the assay progresses, and tracks the storage information at the end to derive a legal schedule.

A DAG edge $e \in E$ represents a droplet. The scheduler maintains an ordered list $S(e)$ for each edge; each entry of $S(e)$ is a triple $(t_1, t_2, j)$, which indicates that $e$ is stored in a module of type $j$ from time-step $t_1$ to time-step $t_2$. When $e$ is initially stored in a module of type $j$ at time $t_1$, the initial entry is $(t_1, -, j)$, since the scheduler does not yet know when $e$ will leave the module. When $e$ leaves the module at time $t_2$, it fills in the entry $(t_1, t_2, j)$. If $e$ is transported to an assay operation, the scheduler is done. If $e$ is transported to a different module of type $j'$ for storage, the scheduler then allocates a new triple $(t_2, -, j')$ and inserts it at the end of $S(e)$ to maintain the order.

The scheduler must track the number of droplets *stored* by each module type, in addition to the total number of modules of each type that have been allocated. It tracks this information in an array $B[0 \ldots D+F+1]$. Recall that $k$ is the maximum number of droplets that a module can store. Suppose that the scheduler stores a droplet in a work module of type $j$. If $B[j] \% k = 0$, then incrementing $B[j]$ to accommodate the new droplet will required a new module of type $j$, so the scheduler must increment $A[j]$. Similarly, if the scheduler removes a droplet from a module of type $j$, and $B[j] \% k = 1$, then a module of type $j$ can be reclaimed, so the scheduler decrements $A[j]$.

#### 2) Scheduling Fails

If the scheduler is unable to find a work module to store a droplet, then scheduling fails, as no resources are available. This *does not* mean that no legal schedule exists; this simply means that list scheduling, a polynomial-time heuristic, has failed to find an affirmative answer to an NP-complete decision problem.

The second cause of failure is subtler. If a large number of droplets are stored on-chip, it may be impossible to allocate *any* of the available assay operations in the priority queue. As the algorithm progresses through future time-steps, all ongoing assay operations that are currently scheduled will complete; these assay operations may produce droplets that require additional storage. If all ongoing operations complete, and none of the available operations can be scheduled due to a lack of available resources, then the list scheduler has failed to find a legal schedule.

### D. Force-Directed List Scheduling (FDLS)

FDLS is a well-established resource-constrained scheduling algorithm in high-level synthesis of digital circuits [7][13]; here, we describe an adaptation of FDLS that targets DMFBs.

#### 1) Implementation Differences and Discussion

Our implementation of FDLS is simpler than the description provided by its inventors, Paulin and Knight [7], in two respects. Firstly, Paulin and Knight dynamically update the priorities for each vertex at each scheduling step, in response to the constraints imposed by the vertices that have been scheduled in previous time steps; our implementation computes priorities statically. Secondly, our force calculation is simpler than Paulin and Knight's; we found that the greedy nature of the underlying list scheduling mechanism performs best with a truncated version of the force calculation that considers only the first few time steps at which an operation can be scheduled.

Paulin and Knight introduced *Force-Directed Scheduling* (not to be confused with FDLS) as a latency-constrained scheduling heuristic, and suggested FDLS as a resource-constrained variant. Under the latency-constrained model, the objective is to minimize the cost of the resources that are allocated to perform a computation. FDS makes an explicit decision to schedule vertex $v$ at time-step $t$, and uses forces to model the constraints that this decision imposes on other vertices. Our implementation of FDLS for DMFBs uses the force computation—performed once, up-front, for each vertex—to guide the priority in which vertices are considered for scheduling during LS. The force calculation is effective, but based on a latency-constrained model.

#### 2) Force Calculation

The first step is to determine the set of all time steps at which each operation can be scheduled. In the resource-constrained context, it is NP-complete to determine if a legal schedule of length $K$ can be found, given a pre-allocated set of resources [1]. Latency-constrained scheduling is therefore used as a proxy.

The *As Soon As Possible (ASAP)* and *As Late As Possible (ALAP)* scheduling algorithms determine the earliest and latest possible time steps in which each given vertex can be scheduled, assuming an infinite supply of resources [1]; under the force-directed paradigm, each operation is assumed to be equally likely to be scheduled at any of its possible time-steps. The *slack* of a vertex is the difference between its ALAP and ASAP scheduling times, i.e.:

$$Slack(v) = ALAP(v) - ASAP(v). \quad (9)$$

The probability of a vertex $v$ being scheduled at time-step $t$, such that $ASAP(v) \leq t \leq ALAP(v)$, is:

$$P(v,t) = \frac{1}{(Slack(v)+1)}. \quad (10)$$

If $v$ cannot be scheduled at time t, then $P(v, t) = 0$. Next, we compute a probability distribution, $Q(t)$, for the time steps in the schedule:

$$Q(t) = \sum_{v \in V} P(v,t). \quad (11)$$

In other words, $Q(v)$ is the sum of the probabilities of all vertices that can be scheduled at time-step $t$. The $Q(t)$ values for all time steps $t$ forms a histogram called a *distribution graph*.

Fig. 4 shows an example assay; we assume that all operations have a latency of 2, and the output operation has a latency of 0; the longest path in the graph has length 8. We compute ASAP and ALAP schedules using a latency constraint of 10. Table 1 shows the ASAP and ALAP values for each vertex, along with the probabilities; it is important to note that Table I shows the time step at which each vertex starts, so the final output vertex finishes 2 time steps *after* the time step at which it is scheduled. Fig. 5 shows the distribution graph.

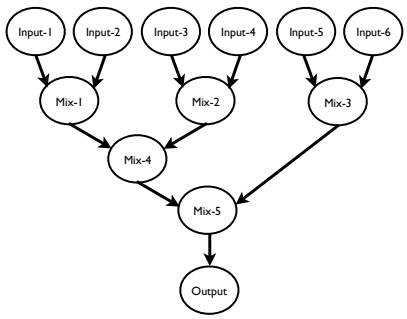


Figure 4. Example assay, specified as a DAG.

TABLE I. ASAP, ALAP AND PROBABILITY DISTRIBUTION VALUES FOR EACH VERTEX IN FIG.4.

| Node | ASAP | ALAP | Time Steps | Probability |
|---|---|---|---|---|
| Input-1 | 0 | 0+2 | 0,1,2 | 1/(2+1)=0.333 |
| Input-2 | 0 | 0+2 | 0,1,2 | 1/(2+1)=0.333 |
| Input-3 | 0 | 0+2 | 0,1,2 | 1/(2+1)=0.333 |
| Input-4 | 0 | 0+2 | 0,1,2 | 1/(2+1)=0.333 |
| Input-5 | 0 | 2+2 | 0,1,2,3,4 | 1/(4+1)=0.2 |
| Input-6 | 0 | 2+2 | 0,1,2,3,4 | 1/(4+1)=0.2 |
| Mix-1 | 2 | 2+2 | 2,3,4 | 1/(2+1)=0.333 |
| Mix-2 | 2 | 2+2 | 2,3,4 | 1/(2+1)=0.333 |
| Mix-3 | 2 | 4+2 | 2,3,4,5,6 | 1/(2+4)=0.2 |
| Mix-4 | 4 | 4+2 | 4,5,6 | 1/(2+1)=0.333 |
| Mix-5 | 6 | 6+2 | 6,7,8 | 1/(2+1)=0.333 |
| Output | 8 | 8+2 | 8,9,10 | 1/(2+1)=0.333 |

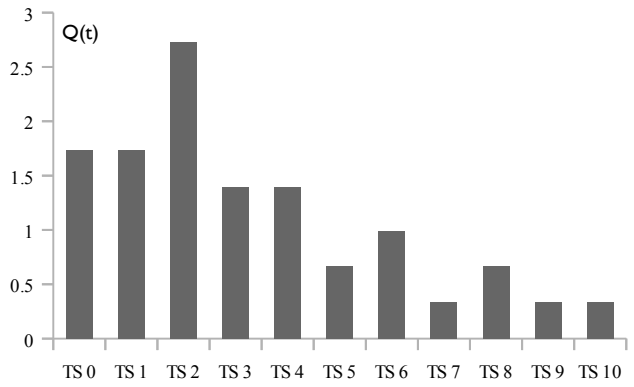


Figure 5. Distribution graph for each vertex in Fig. 4, derived from Table I.

Paulin and Knight's FDS implementation uses the distribution graph to compute probabilistic forces for each vertex, at each time step at which it can be legally scheduled [7]. The time complexity of computing all the forces for vertex $v$ is cubic in *Slack(v)*, which contributed significant overhead to the runtime of the algorithm. We also observed that the schedule quality produced using their cost function was inferior to list scheduling. We believe that their cost is more appropriate for dynamic implementations of list scheduling that update the costs of all available vertices each time that a vertex is scheduled; in contrast, our implementation assigns a static priority to all of the vertices, which never changes.

We experimented with approximately 50 cost functions for assigning static priorities to vertices, and we found two that performed approximately well across all benchmarks. We refer to them as $FauxForce_1()$ and $FauxForce_2()$, because they are reduced versions of Paulin and Knight's force equation [7]:

$$FauxForce_1(v) = \frac{1}{max_{t=ASAP(v)}^{ALAP(v)} P(v,t)Q(t)} \quad (12)$$

$$FauxForce_2(v) = \frac{1}{max_{t=ASAP(v)}^{ASAP(v)+1} P(v,t)Q(t)} \quad (13)$$

The vertices are sorted in increasing order of *FauxForce* in order to prioritize them. Intuitively, $FauxForce_1()$ assigns the highest priority to vertices that have high slack values and at least one time step in a latency-constrained schedule with very little competition from other vertices. FDS would thus assign such a vertex to this time-step.

$FauxForce_2()$ is based on the observation that the list scheduling framework tends to assign vertices to earlier time steps within their slack window because of its greedy nature; consequently therefore, $FauxForce_1()$ could assign a high priority to a vertex whose most favorable time step in the latency-constrained context is fairly late in its slack window. To prevent this from occurring, $FauxForce_2()$ only considers the first two time-steps in the slack window of each vertex.

## IV. EXPERIMENTAL RESULTS

We compare the runtime and solution quality of FDLS with three other DMFB scheduling algorithms: List Scheduling (LS), as described in Section III.C, Path Scheduling (PS) [3], and two genetic scheduling algorithms (GA-1 [11] and GA-2 [9]). $FDLS_1$ and $FDLS_2$ respectively refer to FDLS using cost functions $FauxForce_1()$ and $FauxForce_2()$, as shown in Eqs. (11) and (12).

LS uses a priority function similar to modified list scheduling (MLS) [11]; however, it does not include MLS' rescheduling step, as our goal is to compare the quality of cost functions within the standard LS framework. The PS implementation is identical to ref. [3]. As both LS and PS are polynomial-time heuristics, we are most interested to see whether FDLS can improve their results within a reasonable runtime overhead.

GA-1 and 2 are iterative improvement algorithms that find locally optimal solutions, but have significantly longer runtimes than LS, PS, or FDLS. Both GA implementations start with an initial population size of 20 schedules, and run for 100 generations, evaluating approximately 2000 randomly generated schedules (some of which may be redundant with respect to one another).

We considered three publicly available assays: PCR, In-vitro, and Protein [10]. PCR is very small and all algorithms easily scheduled it optimally; we omit PCR from our results. We targeted a DMFB with four work modules; all modules were specialized with appropriate detectors for the two assays [10]. Input generation and dispensing times were 2s for the In-vitro assays [11] and 7s for the Protein assay [10]. We ran two separate sets of experiments, where each chamber can store up to $k = 2$ and $k = 4$ droplets.

Tables II and III report the schedule length (TS, 1s time-steps) and runtime (ms) of each scheduling algorithm on each benchmark. All computations were performed on an Intel Core2 Duo T5550 machine running at 1.83GHz with 3GB of RAM.

### A. Schedule Length

**In-vitro:** All scheduling algorithms achieved the same schedule lengths for $k = 2$ and $k = 4$. $FDLS_1$ and $FDLS_2$ produced schedules no longer than LS and PS for all In-vitro assays, and shorter than one or the other for the four largest. PS generated longer schedules than LS in two cases, a shorter schedule in one case, and equal-length schedules in two cases.

TABLE II. SCHEDULE LENGTHS (TS) OBTAINED BY DIFFERENT SCHEDULING ALGORITHMS FOR THE DMFB.

| | In-Vitro (Identical results for k=2 and k=4,) | | | | | Protein | |
|---|---|---|---|---|---|---|---|
| | (4s_4r ) | (3s_4r) | (3s_3r) | (2s_3r) | (2s_2r) | k=4 | k=2 |
| LS | 45 | 31 | 25 | 21 | 15 | 198 | 226 |
| PS | 45 | 33 | 27 | 19 | 15 | 187 | 187 |
| $FDLS_1$ | 41 | 31 | 25 | 18 | 15 | 182 | 209 |
| $FDLS_2$ | 41 | 31 | 25 | 18 | 15 | 182 | 209 |
| GA-1 | 39 | 29 | 23 | 18 | 15 | 179 | 200 |
| GA-2 | 39 | 29 | 23 | 19 | 15 | 194 | 199 |

TABLE III. RUNTIMES (MS) OF THE DMFB SCHEDULING ALGORITHMS FOR THE CASE WHERE EACH WORK MODULE CAN STORE UP TO K=4 DROPLETS.

| | In-Vitro | | | | | Protein |
|---|---|---|---|---|---|---|
| | (4s_4r ) | (3s_4r) | (3s_3r) | (2s_3r) | (2s_2r) | |
| LS | 10 | 4 | 2 | 1 | <1 | 8 |
| PS | 3 | 2 | 1 | 1 | <1 | 3 |
| $FDLS_1$ | 37 | 15 | 9 | 4 | 2 | 198 |
| $FDLS_2$ | 29 | 14 | 8 | 4 | 1 | 154 |
| GA-1 | 15,297 | 9,846 | 5,450 | 3,226 | 1,534 | 12,573 |
| GA-2 | 15,964 | 10,121 | 5,655 | 2,974 | 1,687 | 10,124 |

In general, GA-1 and GA-2 produced the shortest schedules, with the exception of (2s_3r), where GA-2 produced schedule of length 19 TS, while $FDLS_1$, $FDLS_2$, and GA-1 produced schedules of length 18 TS. The difference in schedule length between GA-1, GA-2, $FDLS_1$, and $FDLS_2$ was at most 2 TS for all In-vitro assays. Given that GA-1 and GA-2 are long-running iterative improvement algorithms, it is expected that they would produce the shortest schedules in this study.

**Protein:** Varying the number of droplets, *k*, that each work chamber can store affected the length of the schedules significantly, except for PS which achieved schedules of 187 time-steps for $k = 2$ and $k = 4$. LS, $FDLS_1$ and $FDLS_2$ produced shorter schedules than PS for $k = 4$, and longer schedules when $k = 2$. These results echo ref. [3], which showed that PS is more robust to variations in storage availability than LS and FDLS.

$FDLS_1$ and $FDLS_2$ produced significantly shorter schedules than LS for both $k = 2$ and $k = 4$; taken in conjunction with the results for In-vitro, FDLS appears to be an unequivocal improvement over LS.

GA-1 produced the shortest overall schedule for the case $k = 4$, while PS produced the shortest overall schedule for the case $k = 2$; GA-2 produced a notably poor quality schedule for $k = 4$. It is important to note that GA-1 and GA-2 repeatedly run LS, varying the vertex priorities randomly. Thus, for the resource-constrained case ($k = 2$), the results effectively show that one run of PS is more effective than multiple runs of LS (which generalizes to FDLS, GA-1, and GA-2). When more resources are available ($k = 4$), approaches based on LS can produce more effective schedules than PS.

### B. Schedule Length

Table III reports the runtime of the heuristics for the case $k = 4$; similar results were obtained for $k = 2$, but are omitted to conserve space. Among the heuristics, PS had the fastest runtime, followed by LS, then $FDLS_2$ and then $FDLS_1$; the speed of PS aligns with the results reported in ref. [3]. The genetic algorithms ran considerably slower than the heuristics, which is to be expected. $FDLS_1$ and $FDLS_2$ run slower than LS, because they both involve a pre-processing stage to compute the forces used for priority. $FDLS_2$ runs faster than $FDLS_1$ because its $FauxForce_2$ calculation in Eq. (13) is a truncated version of $FDLS_1$'s $FauxForce_1$ calculation in Eq. (12).

### C. Summary

$FDLS_1$ and $FDLS_2$ consistently produced schedules of better or comparable quality to LS and PS, while often approaching the quality of GA-1 and GA-2. Between $FDLS_1$ and $FDLS_2$, we consider $FDLS_2$ to be superior, since it produced equal-length schedules as $FDLS_1$ in all cases, while running faster because it performs a truncated version of $FDLS_1$'s *faux-force* calculation.

## V. CONCLUSION

We have improved the quality of list scheduling for DMFBs by developing cost functions inspired by FDS in high-level synthesis of digital systems. The most effective cost function that we found performed quite well for the In-vitro assays that we considered, but left room for improvement with Protein, especially when storage space was restricted (i.e., the case $k = 2$ in Table III). We suspect that a dynamic scheduling approach that re-computes priorities after each operation is scheduled could potentially yield better results, albeit with a larger runtime overhead. Such an approach would be closer to the original FDLS algorithm proposed by Paulin and Knight [7].

## ACKNOWLEDGMENT

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

## REFERENCES

[1] G. De Micheli. Synthesis and Optimization of Digital Circuits. McGraw-Hill Higher Education, 1994.

[2] J. Ding, K. Chakrabarty, and R. B. Fair, "Scheduling of microfluidic operations for reconfigurable two-dimensional electrowetting arrays," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 20, no. 12, pp. 1463-1468, Dec. 2001.

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

[4] T. Ho, K. Chakrabarty and P. Pop, "Digital microfluidic biochips: recent research and emerging challenges," in *Proc. Int. Conf. HW/SW Codesign and Sys. Synth. (CODES+ISSS)*, Taipei, Taiwan, 2011, pp. 335-343.

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

[6] E. Maftei, P. Pop, and J. Madsen, "Tabu search-based synthesis of dynamically reconfigurable digital microfluidic biochips," in *Proc. Int. Conf. Compilers, Architecture, and Synthesis for Embedded Sys. (CASES)*, Grenoble, France, 2009, pp. 195-204.

[7] P. G. Paulin and J. P. Knight, "Force-directed scheduling for the behavioral synthesis of ASICs," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 8, no. 6, pp. 661-679, June, 1989.

[8] 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, Mar. 2002.

[9] A. J. Ricketts, K. Irick, N. Vijaykrishnan and M. J. Irwin, "Priority scheduling in digital microfluidics-based biochips," in *Proc. Conf. on Design Automation and Test in Europe (DATE)*, Munich, Germany, 2006, pp. 329-334.

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

[11] 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.

[12] F. Su, W. Hwang and K. Chakrabarty, "Droplet routing in the synthesis of digital microfluidic biochips," in *Proc. Conf. on Design Automation and Test in Europe (DATE)*, Munich, Germany, 2006, pp. 323-328.

[13] W. F. J. Verhaegh, P. E. R. Lippens, E. H. L. Aarts, J. H.M. Korst, J. L. van Meerbergen, and A. van der Werf, "Improved force-directed scheduling in high-throughput digital signal processing," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 14, no. 8, pp. 945-960, Aug. 1995.

[14] 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, article #13, 2007.