Title

Resource-Constrained Scheduling for Digital Microfluidic Biochips

Journal

ACM Journal on Emerging Technologies in Computing Systems, 14(1)

ISSN

1550-4832

Authors

O'neal, Kenneth
Grissom, Daniel
Brisk, Philip

Publication Date

2018-01-31

DOI

10.1145/3093930

Peer reviewed

Resource-Constrained Scheduling for Digital Microfluidic Biochips

KENNETH O'NEAL, University of California, Riverside
DANIEL GRISSOM, Azusa Pacific University
PHILIP BRISK, University of California, Riverside

Digital microfluidics based on electrowetting-on-dielectric technology is poised to revolutionize many aspects of chemistry and biochemistry through miniaturization, automation, and software programmability. Digital microfluidic biochips (DMFBs) offer ample spatial parallelism, which is then exposed to the compiler. The first problem that a DMFB compiler must solve is resource-constrained scheduling, which is NP-complete. If the compiler is applied off-line, then long-running algorithms that produce solutions of high quality, such as iterative improvement or branch-and-bound search, can be applied; in an online context, where a biochemical reaction is to be executed as soon as it is specified by the programmer, heuristics that sacrifice solution quality to attain a fast runtime are used. This article describes in detail the algorithms and heuristics that have been proposed for resource-constrained scheduling, focusing on several recent contributions: path scheduling and force-directed list scheduling. It also discusses shortcomings and limitations of existing optimal scheduling problem formulations based on Integer Linear Programming and presents an updated formulation that addresses these issues. The algorithms are compared and evaluated on an extensive benchmark suite of biochemical assays used for applications, such as *in vitro* diagnostics, protein crystallization, and automated sample preparation.

CCS Concepts: • **Applied computing** → **Health informatics**; • **Computer systems organization** → **Embedded and cyber-physical systems**; • **Hardware** → **Operations scheduling**; **Biology-related information processing**; **Microelectromechanical systems**;

Additional Key Words and Phrases: Laboratory-on-a-Chip (LoC), digital microfluidic biochip (DMFB), electrowetting-on-dielectric (EWoD)

ACM Reference format:
Kenneth O'Neal, Daniel Grissom, and Philip Brisk. 2017. Resource-Constrained Scheduling for Digital Microfluidic Biochips. *J. Emerg. Technol. Comput. Syst.* 14, 1, Article 7 (October 2017), 26 pages.
https://doi.org/10.1145/3093930

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

Preliminary and abridged versions of different aspects of this article were presented in *Proceedings of the 49th ACM/IEEE Design Automation Conference*. 2012, 26–35, and *Proceedings of the 20th IEEE/IFIP International Conference on VLSI and System-on-Chip*. 2012. 7–12.

Authors' addresses: K. O'Neal, Department of Computer Science and Engineering, Winston Chung Hall, Room 351, University of California, Riverside, Riverside, CA 92521; email: konea001@ucr.edu; D. Grissom, Building One, room 204, 901 E. Alosta Ave., Azusa, CA 91702; email: dgrissom@apu.edu; P. Brisk, Winston Chung Hall, Room 339, University of California, Riverside, Riverside, CA 92521; email: philip@cs.ucr.edu.

https://doi.org/10.1145/3093930

1 INTRODUCTION

Emerging laboratory-on-a-chip (LoC) technology is poised to revolutionize a wide variety of biochemical analyses through miniaturization and automation. One of the most promising LoC technologies is electrowetting-on-dielectric (EWoD), which manipulates discrete liquid droplets on a two-dimensional grid through electrostatic actuation [Pollack et al. 2002; Dhindsa et al. 2011; Noh et al. 2012; Hadwen et al. 2012].

We refer to a two-dimensional electrowetting array as a Digital Microfluidic Biochip (DMFB). Figure 1 illustrates the working principles of a DMFB. Figure 1(a) depicts a two-dimensional electrode array. Figure 1(b) shows a one-dimensional cross-section: a droplet is sandwiched between two layers. The top layer includes a top plate, ground electrode, and a hydrophobic layer that directly contacts the droplet. In the bottom layer, the control electrodes are embedded in a bottom plate, which is covered by a hydrophobic layer. The hydrophobic layer above and below insulates the electrodes from the droplet, which prevents electrolysis [Huang et al. 2012]. Figure 1(c) illustrates the working principle of droplet transport via electrostatic actuation. Actuating control electrode CE2 beneath the droplet holds it in-place. The droplet itself is longer than CE2's length, and hangs over adjacent control electrodes CE1 and CE3. Activating CE3 pulls the droplet to the right, holding it in place between CE2 and CE3. Deactivating CE2 allows the droplet to be pulled further to right, eventually resting atop CE3.

DMFB technology is inherently programmable and offers abundant spatial parallelism. At present, DMFBs are programmed at a very low level: the user (or "programmer") manually creates a sequence of electrodes to activate to achieve the desired biochemical functionality. Recent work has shown that DMFBs can be programmed at a much higher level of abstraction using domain-specific languages [Grissom et al. 2014]. A key challenge is to design and implement compiler algorithms to best take advantage of the spatial parallelism inherent to DMFB technology.

Figure 2 illustrates the five basic operations that form the DFMB's instruction set. DMFBs can perform droplet transport, splitting, merging, mixing, and storage. The optional inclusion of integrated sensing and actuation technologies can provide certain regions of a DMFB with additional capabilities (e.g., capacitive touch sensing for droplet positioning, heating, etc.).

Figure 3 illustrates a three-stage DMFB compiler. The biochemical reaction (assay) is specified as a directed acyclic graph (DAG): vertices represent biochemical operations, and edges represent droplet dependencies between operations. The scheduler determines the timestep at which each assay operation executes while adhering to resource and dependence constraints; the placer determines the location on the DMFB to perform each scheduled operation; and the router transports droplets between placed operations, while ensuring that droplets do not inadvertently mix (Su et al. 2006) and/or contaminate one another (Zhao and Chakrabarty 2012); these problems are NP-complete (Böhringer 2006; Su and Chakrabarty 2006; Su and Chakrabarty 2008).

This article presents a comparative analysis of DMFB scheduling algorithms published by the authors and others. This article compares *List Scheduling (LS)* (Su and Chakrabarty 2008; Grissom and Brisk 2012a), *Force-directed List Scheduling (FDLS)* (O'Neal et al. 2012), and *Path Scheduling (PS)* (Grissom and Brisk 2012b), the lattermost of which has been optimized for DAGs that are either trees or forests of trees. This article includes two genetic extensions to List Scheduling (GA-LS1 and GA-LS2) (Ricketts et al. 2006; Su and Chakrabarty 2008) and a novel genetic extension to Path Scheduling (GA-PS). This article also includes an optimal *Integer Linear Programming (ILP)* formulation of the DMFB scheduling problem, which corrects several shortcomings of prior formulations (Ding et al. 2001; Su and Chakrabarty 2008). To the best of our knowledge, this article represents the most comprehensive analysis of algorithmic solutions to the DMFB scheduling problem published, to date.

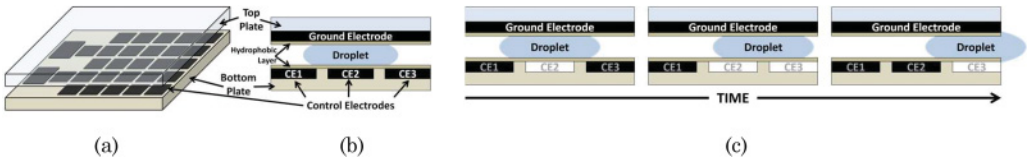

Fig. 1. (a) A DMFB is a planar array of electrodes; (b) cross-sectional view of electrode array; (c) a droplet is transported from control electrode 2 (CE2) to CE3 by activating (white) CE3, while deactivating (black) CE2, allowing for droplets to be transported around the DMFB.

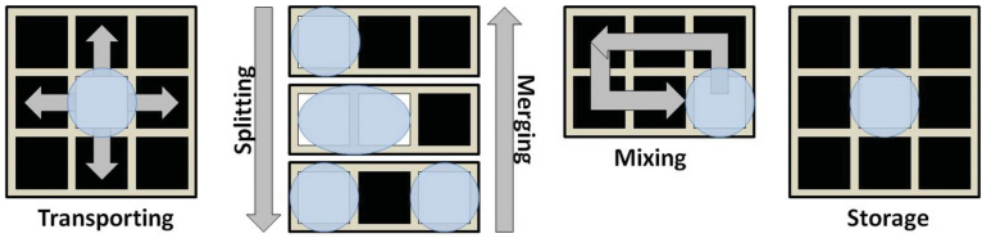

Fig. 2. Basic microfluidic operations form the building blocks for assays to be executed.

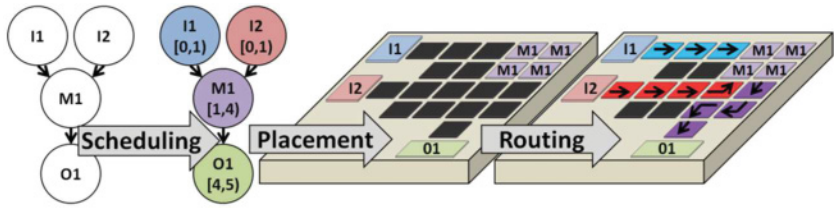

Fig. 3. A DMFB compiler consists of three stages: scheduling, placement, and droplet routing.

2 RELATED WORK

2.1 Resource-Constrained Scheduling

DMFBs are inherently reconfigurable, unlike digital logic, as one resource (electrodes or contiguous groups of electrodes) performs droplet operations (mixing, etc.), storage, and transport. In resource-constrained scheduling, the information provided is the DMFB dimensions, the number of droplet I/O ports on the perimeter of the chip, and any external devices (heaters, detectors, etc.) that may be available. During scheduling, each scheduled mix operation consumes reconfigurable resources that may be otherwise used for storage; as a consequence, it is not always possible to find a legal schedule, if the resource requirements of the DAG far exceed the DMFB capacity.

Moreover, the legality of a given DMFB schedule may not be known until after placement completes, as illustrated in Figure 4(a). Here, seven operations are scheduled to execute concurrently, but only six can be placed on-chip due to fragmentation. Assuming a better placement cannot be found, the only way to rectify the situation is to recompute the schedule with more stringent resource constraints. To avoid this situation, all present and prior work on resource-constrained DMFB scheduling (including this article) conservatively under-approximates the number of operations that can execute concurrently (Ding et al. 2001; Ricketts et al. 2006; Su and Chakrabarty 2008; Grissom and Brisk 2012a; O'Neal et al. 2012); Section 3 and 4 discuss these models in detail.

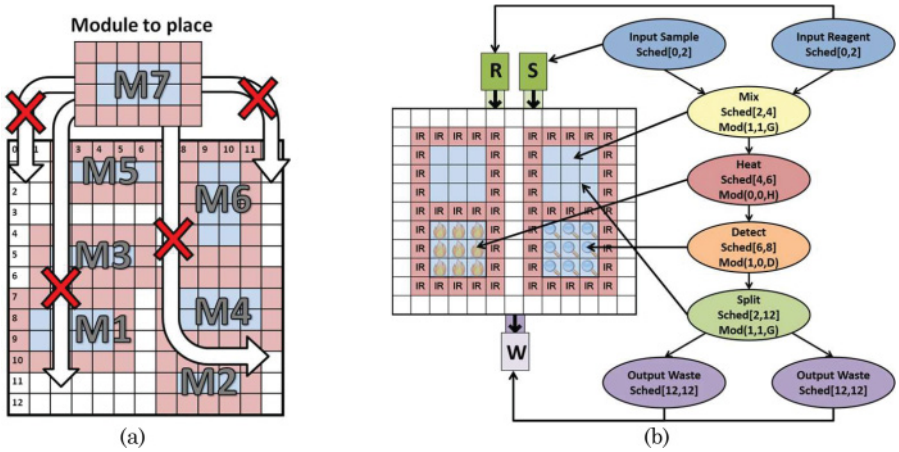

Fig. 4. (a) A placement failure may occur due to fragmentation: even though there are enough empty cells in the DMFB to accommodate operation M7, a contiguous 4×6 or 6×4 sub-array is unavailable. (b) A virtual topology articulates the available resources to the scheduler, ensuring that fragmentation does not occur.

The aforementioned models generalize the notion of a virtual topology (Grissom and Brisk 2012b; Grissom and Brisk 2014; Grissom et al. 2014), as shown in Figure 4(b). A virtual topology partitions the DMFB into a 2D array of work modules connected by a 2D mesh of streets for droplet transport. Work modules are reconfigurable and may perform basic assay operations (mix, merge, split, store), and may be specialized by integrating external devices (e.g., heaters, detection, etc.). I/O reservoirs on the perimeter of the chip are non-reconfigurable and can only perform dispense and output operations (Ding and Chakrabarty 2001). Figure 4(b) depicts a 2×2 array of four work modules, two of which are specialized. The scheduler can issue four mix operations concurrently, three mix and one heat or detect operation, two mix operations plus one heat and one detect operation, and so on. By binding operations to work modules, the placer downstream avoids the fragmentation problem shown in Figure 4(a).

The objective of the schedulers presented in this article, and in the related work described above, is to minimize schedule length. For online scheduling, compute time is more important than the solution quality (Luo et al. 2013, Grissom and Brisk 2014; Grissom et al. 2014; Jaress et al. 2015). There has also been work on dynamic scheduling where operation latencies are determined dynamically (e.g., via sensor) (Luo et al. 2014; Alistar and Pop 2015). These latter systems necessitate an online approach to compilation, including scheduling, which is beyond the scope of this article.

Luo and Akella (2011) schedule multiple instances of the same assay (DAG) to execute in a pipelined fashion. They introduce an optimal preemptive scheduling algorithms for binary mixing trees on DMFBs with a fixed number of mixers. The scheduling model discussed here differs in three respects: (1) each DAG is scheduled once, and pipelining is not considered; (2) preemption is not supported; and (3) most algorithms presented here can schedule any DAG, although some are limited to trees as forests that support any assay operation and do not require the tree to be binary. Future work may consider pipelined scheduling models with preemption.

2.2 Combined Optimization of Scheduling and Placement

Scheduling and placement can be co-optimized together, offering the compiler an extra degree of freedom: module selection. Varying the mixer dimensions changes the operation latencies (Paik

et al. 2003): larger mixers yield faster operation times; however, using large mixers reduces the available spatial parallelism.

In contrast, resource-constrained scheduling assumes that all mixing operations use the same module size and have equal latencies. Xu and Chakrabarty (2008), Xu et al. (2008), Maftei et al. (2010), and Maftei et al. (2013) relaxed this assumption using iterative improvement metaheuristics: the module associated with each mixing operation is perturbed randomly; fast scheduling and placement heuristics are applied; illegal scheduling and/or placement solutions are discarded; legal solutions are compared to the best solution found thus far, and may be used to generate further perturbations. These metaheuristics yield locally optimal solutions, when they can be found, but do not guarantee discovery of a legal solution, if one exists.

Yuh et al. (2007) convert scheduling and placement into a 3D placement problem, where the third dimension is time. Their approach uses a data structure called the T-Tree, which requires that each module placed in the 2D space is abutted to another module that is scheduled to execute at the same time. The T-Tree placer can generate highly compact application-specific DMFB architectures; however, it may run into problems when targeting pre-fabricated DMFBs. For example, it cannot simultaneously schedule two concurrent detection operations on modules that are placed on opposite sides of the chip, since there is no way to abut them.

A more comprehensive approach is to formulate combined scheduling, placement, and routing as an instance of the Boolean satisfiability (SAT) problem, which can then be solved optimally using a SAT solver (Keszocze et al. 2014). This approach can only compile small assays onto small DMFB arrays due to the large search space size; it cannot handle the larger problem instances evaluated in this article.

Latency-Optimization Synthesis with MOdule Selection (LOSMOS) (Liu et al. 2013) starts by binding each mixing operation to the largest and fastest module available. LOSMOS then executes a variant of modified list scheduling (Su and Chakrabarty 2008) whose objective function tries to minimize storage requirements. This is followed by Iterative operation rebinding, which can further reduce the schedule length. Rebinding can be beneficial, for example, in situations where rebinding one or more operations to slower, yet smaller, modules frees up enough space on-chip to increase the number of concurrently scheduled operations. By iteratively altering the binding and repeatedly rescheduling, LOSMOS can often reduce the latency compared to the initial list scheduling result. LOSMOS considers resource constraints in one dimension but does not guarantee that the resulting 2D placement for each timestep is legal.

3 ASSUMPTIONS in RESOURCE-CONSTRAINEd dmfb scheduling

3.1 General Assumptions

The objective of the scheduling problem is to minimize the latency of the schedule, that is, to minimum the maximum completion time of all assay operations. The latency of operations is assumed to be known at compile time, and we assume that all mixers have the same dimensions, which is determined *a priori*. Biochemical operations and applications do not have strict deadline requirements.

3.2 Assumptions in Prior Work

As per Section 2.1, DMFB schedulers conservatively approximate DMFB resource availability. Su and Chakrabarty (2008) apply a one-dimensional constraint: N_a is the number of available resource-on chip; at each timestep of the schedule, N_{mix} is the number of mixing operations scheduled, and N_{mem} is the number of droplet storage operations scheduled. The resource demand of stored droplets at the given timestep is βN_{mem}, where β represents the ratio of storage module to

mixer module size. Ignoring external devices, the resource constraint for each timestep is

$$N_{mix} + \beta N_{mem} \leq N_a. \tag{1}$$

It is left to the user to select a value of N_a that conservatively under-approximates the spatial DMFB resources without grossly underutilizing the chip.

Constraints involving external devices may limit spatial parallelism. For example, if there are N_d detectors, then at most N_d detection operations can execute concurrently; however, the space on top of an unused detector could still perform mixing or storage; this detail is often omitted from the problem formulation (Su and Chakrabarty 2008).

3.3 Assumptions in Our Work

As mentioned in Section 2.1 and Figure 4(b), our formulation of resource-constrained DMFB scheduling provides I/O reservoirs and work modules as the set of resources.

I/O reservoirs are non-reconfigurable (Ding et al. 2001): input reservoirs dispense fluids; each reservoir dispenses exactly one fluid type during assay execution. Output modules can be used to collect fluid for removal from the chip for subsequent analysis or for waste; any fluid can be routed to a waste module for disposal. Our benchmarks do not produce non-waste output for collection.

Work modules are reconfigurable, but can perform non-reconfigurable operations if augmented with dedicated external devices. In Figure 4(b), the two modules on top have no external devices, and are called *general* work modules; the two on the bottom are *specialized* (one has a heater; the other has an integrated detector). Each work module can perform one mix or split operation, or store up to k droplets. For 3×3 work modules, $k = 4$ is the maximum number of droplets that can be stored, but setting $k \leq 2$ ensures provably deadlock-free droplet routing (Grissom and Brisk 2014).

4 PRELIMINARIES

4.1 Operation and Component Types

We associate an integer *type* with each assay operation and component; we use a *compatibility* function to determine whether a component can execute an 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 $1 \ldots D$ 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 (e.g., for waste).

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

4.2 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 $0 \leq f(c) \leq D$, that is, 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 $f(c) = f(v)$, that is, v is a specialized or I/O assay 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 $v \in V$.

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.

4.3 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. 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 (Su and Chakrabarty 2008), 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 $H(u) = [S(u) + L(u), S(v)]$; split may have multiple successors, some of whose start times might be later than $S(v)$. Thus, some work module must be available to store u during $H(u)$. For vertex $u \in V$, the storage time is:

$$H(u) = max\{S(v) \mid (u, v) \in E\} - (S(u) + L(u)). \tag{2}$$

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

5 PROBLEM FORMULATION

5.1 Inputs and Objective

The inputs to the resource constrained scheduling problem for DMFBs are as follows: (i) an assay specified as a DAG $G = (V, E)$; (ii) the latency of each operation: a function $L : V \rightarrow \{1, 2, \ldots\}$; (iii) the number of distinct fluids F; (iv) the type of each operation: a function T; and (v) a DMFB, including work modules and I/O reservoirs, characterized by the number of components of each type: $N = N_0 = N_1 + \cdots + N_{D+F+1}$.

The objective is to compute a legal schedule that minimizes execution time, that is,

$$Obj = min\{max_{u \in V}\{S(u) + L(u)\}\}. \tag{3}$$

The last vertex to finish is guaranteed to be a sink, which requires no storage time.

5.2 Legality

A legal schedule must satisfy the precedence constraint:

$$\forall (u,\ v) \in E,\ S(v)\ \geq\ S(u)\ +\ L(u). \tag{4}$$

This constraint satisfies fluidic dependencies: each fluidic operation can only begin after all of its predecessors complete their respective operations.

Resource constraints must be satisfied for each timestep. The schedule does not bind operations to specific resources, but it does need to ensure that sufficient resources are available to perform the operations that have been scheduled.

Let t be a timestep in the schedule. Let $r_j(t)$ be the set of operations of type j scheduled at timestep t, and $Z(t)$ be the set of operations stored at timestep t, that is,

$$r_j(t)\ =\ \{u | u \in V,\ h(u) =\ j,\ S(u) < t < S(u) + L(u)\}, \tag{5}$$

$$Z(t) =\ \{u | u \in V,\ S(u) + L(u) < t < S(u) + L(u) + H(u)\}. \tag{6}$$

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

$$|r_j(t)|\ <\ N_j,\ 1 < j < D + F + 1, \tag{7}$$

that is, the number of non-general operations of each type scheduled at each timestep cannot exceed the number of available work modules or I/O reservoirs of that type.

Any work module can perform a reconfigurable operation. If constraint Equation (5) is satisfied, then let $N_m(t)$ denote the number of available work modules for these operations:

$$N_m(t) =\ N_0\ +\ \sum_{j=1}^{D}(N_j\ -\ |r_j(t)|). \tag{8}$$

In other words, $N_m(t)$ is the number of available work modules that are not performing non-reconfigurable operations (e.g., heating, detection, etc.).

The last resource constraint for reconfigurable operations (including storage) is

$$|r_0(t)| + \lceil Z(t)/k \rceil \leq N_m(t). \tag{9}$$

Constraint Equation (9) ensures that the number of reconfigurable operations $|r_0(t)|$ (mix, split) and the number of storage operations $Z(t)$ scheduled at timestep t do not exceed the number of work modules remaining $N_m(t)$. Since each work module can store up to k droplets, the number of work modules required for droplet storage is $\lceil Z(t)/k \rceil$.

6 IMPLEMENTATION DETAILS

6.1 Managing Available Resources

Our schedulers maintain a vector $A[0 \ldots D + F + 1]$, where $A[j]$ is the number of available resources of type j. Initially, $A[j] = N_j$, that is, the total number of resources of type j in the DMFB. Let $u \in V$ be a vertex presently being scheduled. 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 u is scheduled; this way, the scheduler can free up a resource of the appropriate type when u finishes its operation.

Reconfigurable operations can execute on general or specialized work modules. When both module types are available, we give preference to general work modules, which increases the likelihood that a non-reconfigurable operation can be scheduled in a future timestep. If $u \in V$ finishes its operation at the end of the current timestep, then the scheduler decrements $A[h(u)]$ to free up a resource of the appropriate type.

6.2 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 module if doing so is advantageous. Examples of useful droplet movements include:

- Suppose that work modules m_1 and m_2 are initially required to store n droplets, where $k < n < 2k$. In the future, the number of droplets stored in m_1 and m_2, denoted by n_1 and n_2, respectively, may be reduced such that $n_1 + n_2 < k$. Without loss of generality, the droplets stored in m_1 can be transported to m_2, freeing up m_1.
- Suppose that a specialized work module of type j_1 stores a droplet, $A[j_1] = 0$, (no other module of type j_1 is available), and $A[j_2] > 0$ for specialized work module type $j_2 \neq j_1$ (at least one module of type j_2 is available). If vertex $u \in V$ of type $f(u) = j_1$ is ready to be scheduled, and no vertex $v \in V$ of type $f(v) = j_2$ is ready, then it is beneficial to move the droplet to an available module of type j_2; then u can be scheduled immediately.
- If a general work module m is available, then moving droplets from a specialized work module to m reduces competition for the specialized resources (e.g., detectors).

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), indicating that e is stored in a work module of type j from timestep t_1 to t_2. When e is initially stored at timestep t_1, the entry is $(t_1, -, j)$; when e leaves the work module at time t_2, the scheduler fills in the entry (t_1, t_2, j). If e is transported to an operation, then the scheduler is done. If e is transported to a different module of type j' for storage, the scheduler then allocates a new triple $(t_2, \text{-}, j')$ and inserts it at the end of $S(e)$ to maintain the order.

Let $\chi_j(t)$ denote the number of droplets stored in work modules of type j at time t. When the scheduler stores (removes) a droplet in (from) a work module of type j, it increments (decrements) $\chi_j(t)$. After storing a new droplet, condition $\chi_j(t) \ \% \ k = 1$ requires the allocation of a new work module, so the scheduler increments $A[j]$; after removing a droplet, condition $\chi_j(t) \ \% \ k = 0$ frees a work module. Tracking work module usage from one timestep to the next helps the scheduler satisfy resource constraints.

6.3 Scheduling Failures

If a 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 the heuristic, has failed. A second cause of failure is subtler. If many droplets are stored on-chip, then it may be impossible to allocate *any* additional modules to schedule operations that are ready. As the heuristic progresses through future timesteps, all ongoing assay operations that are currently scheduled will complete; these operations may produce droplets that require additional storage. If all ongoing operations complete, and none of the ready operations can be scheduled due to a lack of available resources, then the heuristic has failed to find a legal schedule.

To reduce the likelihood of scheduling deadlock, we can limit the number of droplets permitted on the DMFB during any timestep (d_{max}) as follows:

$$d_{max} = kN_m - 1. \tag{10}$$

7 RESOURCE-CONSTRAINED DMFB SCHEDULING ALGORITHMS

7.1 List Scheduling and Its Variants

List scheduling is a greedy, constructive heuristic; pseudocode is shown in Figure 5. The first step is to assign a priority to each operation (vertex) in the DAG; many different priority functions are possible. List scheduling maintains a candidate list of operations that are schedulable. Initially

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

Fig. 5. Pseudocode for List Scheduling.

(at timestep $t = 0$), only primary input (droplet dispense) operations are schedulable; as the algorithm progresses (t increases), operations become schedulable when all of their parents complete.

Lines 7–40 describe the main scheduling loop, which continues until all operations are scheduled; during each iteration, as many operations as possible are scheduled for the current timestep before incrementing to the next available timestep (*Line 39*). Array *availMods*[] holds the available number of available work modules. At each timestep, operations that have finished relinquish the work modules they were using (*Lines 8–11*). Next, the candidate nodes are sorted (*Line 12*) and examined in ascending order (*Lines 13–33*) based on a user-specified priority function. Each candidate is checked to see if it can be scheduled (*Line 14*): *CanSchedule*(*c*) returns true if (1) all of *c*'s dispense parents have free input ports of the proper fluid-type, (2) all of *c*'s non-dispense parents are scheduled to complete before the current timestep, (3) there is a free work module to process

c, and (4) processing does not violate Constraint Equation (10). If c can be scheduled, then it is *SetAsScheduled*() with starting and ending timestep s and work module type. *SelectModType*(c) selects a work module type based on c's operation type and always selects a general work module, if possible, to increase the availability of specialized work modules for operations that require them.

Once c is scheduled, *Lines 18–26* examine c's parents. If a parent is an input, then it is scheduled (dispense operations are not scheduled their children are scheduled). Non-dispense parents may require storage if the parent operation finishes before its child operation is scheduled to start. Storage operations are represented by inserting vertices representing one timestep of storage into the DAG G; if a droplet is scheduled for t timesteps of storage, then the scheduler will insert a chain of t storage vertices, one-by-one. This simplifies the process of transporting stored droplets from one work module to another, as discussed earlier. Last, in *Lines 27–31*, any of c's children whose parents are all scheduled are added to the candidate list.

The time complexity of List Scheduling is $O(|V|\log|V| + |E|)$, assuming that a heap-based priority queue is used to sort vertices that are ready to be scheduled.

Priority Functions: Different implementations of List Scheduling use different priority functions. For example, the *Critical Path Priority (CPP)* (Grissom and Brisk 2012b) is the maximum length in the DAG from vertex v to a primary output reachable from v:

$$CPP\,(v) = max\,\{dist\,(v, v_{sink})\}. \tag{11}$$

LOSMOS' List Scheduler (Liu et al. 2013) extends CPP with a second-term,

$$Pri_{LOSMOS}\,(v) = CPP\,(v) + \varphi \cdot st\_saving\,(v), \tag{12}$$

to consider the impact of scheduling v on storage requirements downstream. The article does not describe what value of φ was used in the experiments or the impact of varying the value of φ on schedule length. For these reasons, we do not include a comparison with LOSMOS' List Scheduling in our experiments.

Force-Directed LS (FDLS): *Force-Directed List Scheduling (FDLS)* replaces the CPP priority function with a physics-inspired force computation based on a probability distribution (Paulin and Knight 1989; Verhaegh et al. 1995). We compute forces statically for each vertex, and use them as part of an alternative priority function**.**

The first step is to estimate the timesteps s at which each operation can be scheduled using the *As Soon As Possible (ASAP)* and *As Late As Possible (ALAP)* heuristics, assuming an infinite supply of resources (Paulin and Knight 1989; De Micheli 1994). The *slack* of a vertex is the difference between its ALAP and ASAP times, that is,

$$Slack\,(v) = ALAP\,(v) - ASAP\,(v). \tag{13}$$

The probability to schedule vertex v at timestep t is

$$P\,(v, t) = \frac{1}{Slack\,(v) + 1}, \tag{14}$$

under the assumption that v is equally likely to be scheduled any timestep in the range $[ASAP(v), ALAP(v)]$. For each timestep t outside that range, $P(v, t) = 0$. Next, we compute a probability distribution, $Q(t)$, for the timesteps in the schedule,

$$Q\,(t) = \sum_{v \in V} P\,(v, t), \tag{15}$$

which is the sum of the probabilities of all vertices that can be scheduled at timestep t. The $Q(t)$ values for all timesteps t forms a histogram called a *distribution graph*.

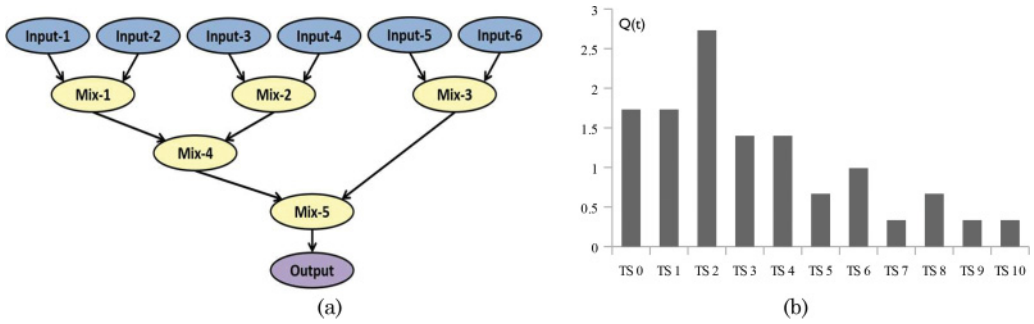

Fig. 6. (a) An example assay and (b) its distribution graph (derived from the probabilities listed in Table 1).

Table 1. ASAP, ALAP, and Probability Distribution Values for Each Vertex in Figure 6(a)

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

Figure 6 shows an example assay and its distribution graph; Table 1 shows the ASAP and ALAP values and probabilities for each vertex. All operations have a latency of 2 timesteps; the longest path in the graph has a length of 10 timesteps. It is important to note that Table 1 shows the timestep at which each vertex starts, so the final output vertex finishes 2 timesteps *after* the timestep at which it is scheduled.

We experimented with around 50 cost functions that assign static vertex priorities, and identified two that performed well across all of our benchmarks; both are reduced versions of the original FDS force equation (O'Neal et al. 2012):

$$FauxForce_1\,(v) = min\left\{\frac{1}{P\,(v,t)\,Q\,(t)}\right\} ASAP\,(v) \le t \le AL\,(v), \tag{16}$$

$$FauxForce_2\,(v) = min\left\{\frac{1}{P\,(v,t)\,Q\,(t)}\right\} ASAP\,(v) \le t \le ASAP\,(v) + 1. \tag{17}$$

Vertices are prioritized in increasing order of *FauxForce*. Intuitively, $FauxForce_1()$ assigns the highest priority to vertices that have large slack values and at least one timestep with limited competition from other vertices. $FauxForce_2()$ is based on the observation that List Scheduling tends to assign vertices to early timesteps within their slack window and therefore only considers the first-two timestep in the slack window of each vertex. We report FDLS results using $FauxForce_2()$ in our experiments.

Modified LS (MLS): *Modified List Scheduling* (Su and Chakrabarty, 2008) includes a rescheduling step that is invoked when the basic List Scheduling algorithm fails (Section 6.3). Su and Chakrabarty present an example that illustrates how rescheduling can be beneficial, while limiting their rescheduling scheme to backtrack by at most one timestep. Our experimental evaluation includes LS but not MLS.

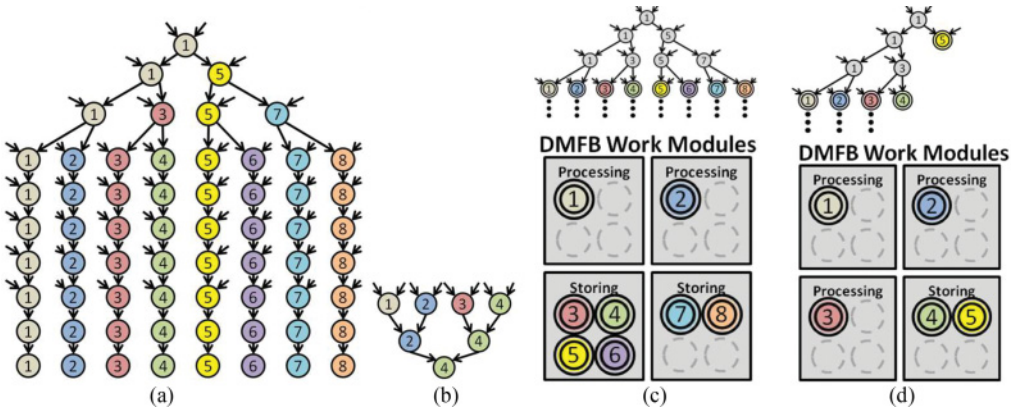

Fig. 7. (a) A protein assay with eight paths; (b) a PCR mixing tree with four paths. Unattached input arrows to a node on both sides represent dispense operations (nodes omitted for clarity). (c) List Scheduling may attempt to process all paths simultaneously, forcing two modules to be used as storage; (d) Path Scheduling first schedules paths 1, 2, and 3, and uses only one module to store droplets.

7.2 Path Scheduling for Trees

Many DAGs representing DMFB assays are either trees or forests in which each connected component is a tree. Path Scheduling (Grissom et al. 2012a) is an effective approach for scheduling trees and performs exceptionally well on resource-constrained DMFBs whose spatial capacity is relatively small compared to the operation-level parallelism in the DAG being scheduled.

As an example, consider the protein dilution assay shown in Figure 7(a): eight paths are identified. Path 1 starts with two dispense operations, while paths 2–7 originate from a dispense operation combined with a split from another path; all paths terminate with an output operation, although this may not occur in the general case (e.g., see Figure 7(b)). In Figure 7(c), List Scheduling tries to schedule parallel operations along all eight paths, injecting eight droplets into the system early on. If the DMFB has four work modules, each of which can store up to four droplets, then two work modules are allocated to storage, leaving just two modules available to perform further assay operations.

Instead, if operations are scheduled exclusively along paths 1, 2, and 3, then just one module is required for storage, as shown in Figure 7(d), providing more effective use of spatial resources. Split operations are deferred until at least one child can be scheduled. This reduces storage contention, which increases work module utilization.

Decomposition into paths is not unique; for example, Path 1 could be any of the eight paths in Figure 7(a). Once paths are identified, the order in which they are scheduled becomes important. Further, path decomposition only makes sense for trees, as scheduling decisions are made on the granularity of paths, not vertices. A DAG exhibits reconvergent paths if there exist two vertices u and v and two paths p_1 and p_2 from u to v that share no other common vertices. Without loss of generality, a decision to schedule path p_1 imposes constraints on p_2: all vertices between u and v must start after the finishing time of u and complete before the starting time of v; it may not be possible to satisfy this constraint after p_1 has been scheduled, or a poor-quality schedule may result. Thus, Path Scheduler is limited exclusively to trees and forests.

Priority Function: Path Scheduling uses a combination of two priorities to produce schedules that minimize the amount of time a droplet spends in the system.

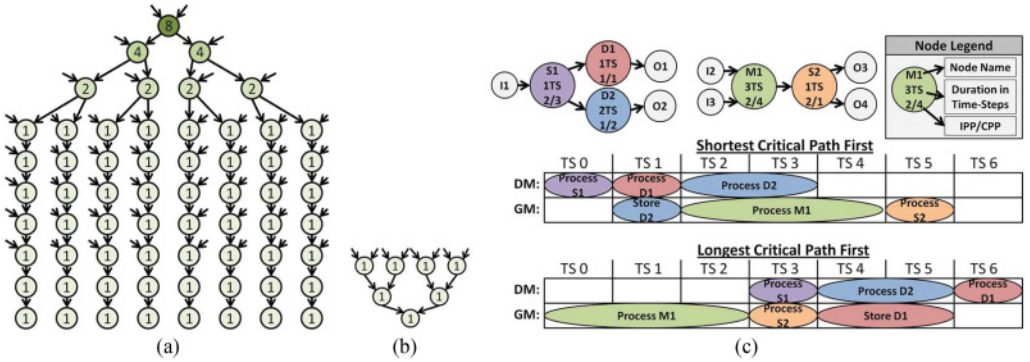

Fig. 8. (a) The high fanout Protein Split 3 assay; nodes are labeled with IPP values (i.e., the number of terminal vertices reachable from the node). (b) The PCR assay with high fan-in, similarly labeled. (c) Two DAGs and their schedules using longest and shortest critical path priorities (CPPs). The DMFB has one general work module (GM) and one detect work module (DM). Since path leaders S1 and M1 have the same IPP values, CPP is chosen as a tie-breaker. Using the shortest CPP as a tiebreaker yields the shorter schedule.

To prevent droplets from entering the system (via a dispense or split) until as late as possible, the *independent path priority (IPP)* of each node is the number of droplets being output in a node's fan-out, as shown in Figures 8(a) and 8(b). The first operation on each path is called the *Path Leader*. If Path 1 is scheduled first, then leaders from Paths 2, 3, and 5 become candidates to be scheduled next, with IPPs of 1, 2, and 4, respectively. Path 2, with the lowest IPP, is processed next, preventing additional splits from being scheduled until later; this reduces demand for droplet storage, which frees up spatial parallelism.

The *Critical Path Priority (CPP)* from Equation (11) is used as a tiebreaker when two paths have equal independent path priorities (IPPs). In Figure 8(c), the IPP of nodes (S1 and M1) are equal, so we use shortest CPP as a tiebreaker. Figure 8(c) shows two schedules where the paths having the shortest and longest CPPs are, respectively, scheduled first. Here, the DMFB has one detection module, so one of the droplets produced by split-node S1 must be stored while the other undergoes detection. Selecting the shortest CPP reduces the number of timesteps dedicated to storage, yielding a shorter schedule. This justifies our choice of shortest CPP as a tiebreaker.

Pseudocode: Figure 9 presents pseudocode for the Path Scheduling heuristic. Prior to scheduling, the IPP and CPP are computed for all operations. Initial candidate operations to be scheduled are those whose parents are dispense operations.

Lines 5–31 describe the main scheduling process that repeats until all candidate nodes have been scheduled. *Lines 6–8* first select the candidate node, or *path leader*, with the lowest priority value (the lowest IPP first, and in the event of an IPP tie, the lowest CPP), reset the scheduling timestep and initialize an empty path.

Lines 9–21 try to allocate resources for an entire path of operations starting with the path leader chosen in *Line 6*, and ending with an output or merge operation (see Figure 7(a)). *Lines 10–12* search for the earliest time gap where the current node, S, can fit, given the available resources. If a gap is found, then there is a resource of type k available from timestep t_i to $t_i + S.duration$, any required input reservoirs are available and there are sufficient resources to store any incoming droplets from S's parent nodes, if needed. If no gap is found in *Lines 10–12*, then the current path cannot be scheduled until later, due to a resource conflict (e.g., insufficient storage for droplet from one of S's parents (in path P) to S); the attempt to schedule the path is discarded and resources that were previously reserved are relinquished. On the next iteration of the main loop (*Lines 5–31*), the same

```
1   Given sequencing graph:            G=(V, E)
2   Given resource constraints:        N_gm N_dm and s_c
3   Assign priorities for all nodes v ∈ V, ∀v based on IPP and CPP
4   Find candidate operations R = {v_i ∈ V: Type(v_j) == input, ∀j: (v_j, v_i) ∈ E}
5   Repeat
6       Select S ⊆ R : Priority(S) ≤ Priority(r), ∀r : r ∈ R
7       Time-step t = S.start + 1    // Try to start scheduling path after last attempted starting timestep
8       Path P = ∅
9       Repeat
10          Attempt to find earliest time-step t_i and module-type k for S:
11              k ∈ (gc, sp): Avail(ts, k) == true, ∀ts : t ≤ t_i ≤ ts < t_i + S.duration
12              while satisfying resource constraints
13          if (Attempt found a gap for S)
14              Set S.start = t_i,  Set S.resType = k
15              Add S to P
16          else // Gap could not be found, S not schedulable now
17              Set P.schedulable = false
18          end if
19          Select new S ⊆ S.children: Priority(S) ≤ Priority(S_ch), ∀S_ch: S_ch ∈ S.children
20      until ( ((Type(S) == mix) AND (S.scheduled == false))
21              OR (Type(S) == output) OR (P.schedulable == false) )
22      if (P.schedulable == true)
23          for ( ∀p ∈ P ){
24              Set p.scheduled = true
25              Reserve resources for path P
26              for (∀c : c ∈ p.children ∧ c ∉ P )
27                  Add c to candidate operations R
28              end for
29          end for
30      end if
31  until (all candidate operations are scheduled: R = ∅)
```

Fig. 9. Pseudocode for Path Scheduling.

node will be selected again (*Line 6*), because the priorities are not changed; however, the algorithm will attempt to begin placing the path leader, S, at a later timestep than the previous attempt, as seen in *Line 7*. If necessary, then the algorithm will continue pushing back the beginning of the path leader by a single timestep each iteration until there is a gap for the entire path; in the worst case, the path leader will begin after all other currently scheduled operations.

If a gap is found for S, then its starting timestep and resource type are temporarily saved and S is added to the current path P (*Lines 13–15*); however, S is not marked as scheduled. Next, *Line 19* select the next node to add to the path from S's children. If the chosen node is an output or unscheduled mixing operation, then P is complete, can be scheduled and can break from the path-constructing loop of *Lines 9–21*; otherwise the loop continues and path scheduler attempts to find a gap for the new S.

Lines 22–30, mark each operation in the schedulable path P as being scheduled and officially reserve the resources allocated to it in *Line 25*. Also, any unscheduled children of the nodes in path P are added to the candidate operations as path leaders.

When *Line 27* adds an operation to the candidate list, the corresponding edge is added to a list of droplets stored indefinitely for later use, starting from their parent's scheduled ending timestep. When path scheduler identifies a gap for operations in *Lines 10–12*, it considers all droplets being stored indefinitely at that point. When an operation is scheduled in *Lines 24–25*, any stored edges/droplets connected to that node are removed from the indefinite storage list and the finite period that the droplet must be stored for, if any, is accounted for in the system's available resources.

7.3 Evolutionary Schedulers

Iterative improvement algorithms are a class of probabilistic metaheuristics for global optimization problems over discrete search spaces, which are tuned to find locally optimal solutions but cannot guarantee global optimality. Depending on the user's choice of parameter values, they often run several orders of magnitude longer than polynomial-time heuristics while avoiding the exponential time complexities of optimal algorithms (assuming P != NP). Here, we describe three evolutionary DMFB schedulers: two, based on List Scheduling, have been published previously (Ricketts et al. 2006; Su and Chakrabarty 2008); the third, based on Path Scheduling (Grissom and Brisk 2012a), is presented for the first time.

Evolutionary algorithms maintain a pool of solutions (legal schedules), which are randomly perturbed by *mutation* and *crossover* operations. Let $G = (V, E)$ be the DAG to schedule. A legal schedule is represented by *chromosome* $C_j = \{g_j(v_1), g_j(v_2), \ldots, g_j(v_n)\}$, $n = |V|$; the value $g_j(v_i)$, a *gene*, is randomly generated in the range $[0, 1]$. The pool of solutions is a set $P = \{C_1, C_2, \ldots, C_m\}$ of m chromosomes; Su and Chakrabarty (2008) set $m = 2n$. The chromosomes adhere to the *random keys* representation (Bean 1994), which ensures that a legal schedule can be derived from *any* chromosome, and that the crossover operation yields two new chromosomes that represent *legal* schedules.

During each evolutionary iteration, new chromosomes are generated randomly through mutation and crossover; a new schedule is derived from each chromosome. The m chromosomes that have the highest fitness values (shortest schedule lengths) survive to the next iteration; the remaining chromosomes are discarded.

The *mutation* operation randomly generates new chromosomes to guarantee population diversity. The *crossover* operation randomly selects two distinct chromosomes C_i *and* C_j, and generates a random integer k in the range $[1, n]$. The crossover operation produces two new chromosomes, C_{ij} and C_{ji} as follows:

$$C_{ij} = \left\{g_i\left(v_1\right), g_i\left(v_2\right), \ldots, g_i\left(v_k\right), g_j\left(v_{k+1}\right), g_j\left(v_{k+2}\right), \ldots, g_j\left(v_n\right)\right\}, \tag{18}$$

$$C_{ji} = \left\{g_j\left(v_1\right), g_j\left(v_2\right), \ldots, g_j\left(v_k\right), g_i\left(v_{k+1}\right), g_i\left(v_{k+2}\right), \ldots, g_i\left(v_n\right)\right\}. \tag{19}$$

If C_i and C_j both have properties that lead to good quality schedules in different parts of the DAG, then C_{ij} and/or C_{ji} could inherit both, improving the overall schedule.

The evolutionary scheduler iterates a fixed number of times. Relevant parameters that affect runtime and quality of solution include: population size |P|, the number of iterations, the number of mutations and crossovers performed during iteration.

We present three evolutionary schedulers, *GA-LS1, GA-LS2,* and *GA-PS*. They use different methods to derive schedules from chromosomes; but are otherwise identical.

GA-LS1: *GA-LS1* (Su and Chakrabarty 2008) sorts the vertices based on their genes. Given chromosome C_i, let $r(g_i(v_j))$ denote the rank of vertex v_j in the sorted list. List Scheduling is then called with $r(g_i(v_j))$ as the priority value associated with vertex v_j; this ensures that all vertices in the DAG have unique priority values.

GA-LS2: *GA-LS2* (Ricketts et al. 2006) encodes a chromosome in which all vertices belonging to each connected component in a DAG receive the same priority. This encoding scheme was specific to the multiplexed *in vitro* diagnostics assay, which has been widely used as a benchmark for scheduling. Our implementation generalizes this encoding to ensure that all vertices in each connected component receive contiguous priority values: if the first component has k_1 vertices, then its vertex priorities (after sorting) are $\{1, 2, \ldots, k_1\}$; if the second component has k_2 vertices, then its vertex priorities (after sorting) are $\{k_1 + 1, k_1 + 2, \ldots k_1 + k_2\}$, and so on. The basic premise,

in terms of the internal List Scheduler, is that once an operation from one connected component is scheduled, it is better to make progress on that component, than to start scheduling operations from another component; this tends to reduce demand for on-chip storage.

GA-PS: *GA-PS* is similar to *GA-LS1*, but uses Path Scheduling, rather than List Scheduling to compute the schedule. In principle, this approach combines the primary benefit of Path Scheduling (superior spatial resource management) with the ability of genetic algorithms to efficiently explore a large search space. Like Path Scheduling, GA-PS can only schedule trees and forests, not general DAGs.

7.4 Optimal Scheduling Based on Integer Linear Programming

Linear Programming (LP) is an optimization technique that can solve resource allocation problems, including scheduling. An LP model comprises a set of variables, an objective function to optimize, and a set of linear constraints to satisfy. An LP solver finds legal values for the variables that satisfy all constraints while optimizing the objective function. *Integer Linear Programming (ILP)* is a subclass of LP in which some or all values are integral (Bradley et al. 1977); ILP solving is known to be NP-Hard in the general case. The ILP-based scheduler presented here uses the LPSolve [LPSolve Reference Guide] C++ API.

The ILP formulation presented here for DMFB scheduling is more general than prior ones. The first ILP formulation (Ding and Chakrabarty 2001) pre-allocates mix and storage resources on the DMFB, failing to leverage its reconfigurable capabilities. It also fails to account for external devices (heaters, detectors, etc.) that actuate specific regions of the chip; when the external device is not in use, these regions can be used for mixing or storage. A subsequent ILP formulation by Su and Chakrabarty (2008) correctly models mixing and storage as being reconfigurable; however, it sub-optimally treats external devices as being non-reconfigurable and cannot schedule DAGs that feature vertices with fanout greater than 1. The ILP formulation presented here addresses the aforementioned shortcomings of both prior ILPs.

First, one or more heuristics are called to schedule the DAG, providing an upper bound B on scheduling latency. This enables us to introduce a set of binary variables to determine the timestep at which each operation is scheduled:

$$x_{i,t} = \begin{cases} 1, & \text{if operation } i \text{ is scheduled to start at time-step } t \\ 0, & \text{otherwise} \end{cases}. \tag{20}$$

Recall that V is set of vertices in the DAG; it follows that $1 \leq i \leq |V|$ and $1 \leq t \leq B$; thus, the formulation encompasses $|V|B$ binary variables in total.

A legal schedule must satisfy the constraint that each assay operation is scheduled to *start* at exactly one timestep:

$$\sum_{t=1}^{B} x_{i,t} = 1, \quad 1 \leq i \leq |V|. \tag{21}$$

Using notation introduced earlier, $S(u_i)$ is the start time of operation u_i. Based on Equations (20) and (21), it follows that

$$S(u_i) = t \,|x_{i,t} = 1. \tag{22}$$

Recalling that $L(u_i)$ is the latency of u_i, the finishing time of u_i is $S(u_i) + L(u_i)$. We then introduce dependency constraints to the ILP model derived from the DAG edges:

$$S(u_j) \geq S(u_i) + L(u_i) \; \forall (u_i, u_j) \in B. \tag{23}$$

The objective is to minimize the maximum finishing time among all assay operations:

$$\text{Minimize}: \left\{ \max_{1 \le i \le |V|} S(u_i) + L(u_i) \right\}, \ 1 \le i \le |V|. \tag{24}$$

Our foremost technical challenge is to properly model resource constraints in a manner that does not restrict reconfigurability. We overcome this challenge by modeling a virtual topology, as discussed in Section 2.1 (Grissom and Brisk, 2014; Grissom et al. 2014), allowing us to express resource constraints in terms of work modules and their capabilities. I/O models on the perimeter of the chip are not reconfigurable (Ding and Chakrabarty 2001).

Recall from Section 4.1 that operations have different types: F is the number of distinct fluids in the system; Type-0 operations are reconfigurable (e.g., mixing, splitting, storage) and can be scheduled onto any type of work module; Type-1 . . . D operations use specialized modules that require external devices (without loss of generality, we consider two specialized module and operation times: heating and optical detection); and Type-$D+1 \ldots D+F$ operations dispense fluids of varying types. The scheduler does not account for Type-$D+F+1$ operations (disposal).

We partition the set of assay operations V as follows: let I be the set of input operation (I_f is the set of input operations that dispense fluid of type f), O be the set of output operations, H be the set of heat operations, Y be the set of detection operations, and $V' = V \setminus \{I, O, H, Y\}$ be the remaining set of reconfigurable assay operations (mix, store, etc.). We assume that output operations take no time and are handled as part of the routing process; therefore, we do not need to schedule output operations in O.

The following constraint ensures that at most I_f droplets of type f are injected into the system at timestep t:

$$\sum_{u_i \in I_f} x_{i,t} \le |I_f|, \ 1 \le t \le B, \ D+1 \le f \le F. \tag{25}$$

Next, we compute $H(t)$ and $Y(t)$, the number of heat and detect modules *in use* at timestep t.

$$H(t) = \sum_{u_i \ \epsilon \ H} \sum_{j=\max\{1, \ t-L(u_i)\}}^{t} x_{i,j}, \ 1 \le t \le B, \tag{26}$$

$$Y(t) = \sum_{u_i \ \epsilon \ Y} \sum_{j=\max\{1, \ t-L(u_i)\}}^{t} x_{i,j}, \ 1 \le t \le B. \tag{27}$$

Suppose that heat or detect operation u_i starts at timestep t (i.e., $x_{i,t} = 1$); then, it is bound to a heat or detect module for the duration of its latency $L(u_i)$; equivalently, u_i occupies a heat or detect module at *any* timestep t if $x_{i,t'} = 1$, $t - L(u_i) \le t' \le t$.

Let N_H, and N_Y be the number of heat and detect modules on-chip. The following constraints ensure that at most N_H heat operations and at most N_Y detect operations are scheduled at each timestep:

$$H(t) \le N_H, \ 1 \le t \le B, \tag{28}$$

$$Y(t) \le N_Y, \ 1 \le t \le B. \tag{29}$$

Next, we turn to reconfigurable operations: *mix, store, split,* and *merge*. The droplet router performs split and merge operations, so we schedule mix operations exclusively and insert storage operations when needed. Hence, V' contains mix operations only.

Table 2. Summary of Scheduling Algorithms and Qualitative Comparison

Algorithm	Description	Solution Quality	Runtime	Limitations
LS	Heuristic	Sub-optimal	Fast	None
FDLS	Heuristic	Sub-optimal	Fast	None
PS	Heuristic	Sub-optimal (Effective on small DMFBs)	Very Fast	Trees/Forests Only
GA-LS1	Iterative Improvement	Locally optimal	Slow (Parameterizable)	None
GA-LS2	Iterative Improvement	Locally optimal	Slow (Parameterizable)	None
GA-PS	Iterative Improvement	Locally optimal (Effective on small DFMBs)	Slow (Parameterizable)	Trees/Forests Only
ILP	Solver	Optimal	Exponential (Worst-Case)	None

First, we compute $M(t)$, the number of mix operations that execute at timestep t:

$$M(t) = \sum_{u_i \in V'} \sum_{j=\max\{1,\ t-L(u_i)\}}^{t} x_{i,j},\ 1 \le t \le B. \tag{30}$$

This computation is similar in principle to Equations (26) and (27).

Allocating storage is somewhat more complicated. Each droplet corresponds to a DAG edge $e = (u_i, u_j)$. Storage is required for droplet e at time t if

$$S(u_i) + L(u_i) < t < S(u_j). \tag{31}$$

This condition can be expressed by a decision variable,

$$z_{i,j,t} = \sum_{k=1}^{t-L(u_i)} x_{i,k} - \sum_{k=0}^{t} x_{j,k}, \tag{32}$$

which takes a value of 1 if t satisfies Equation (31) and a value of 0, otherwise. From there, we can compute $Z(t)$ the number of droplets that require storage at timestep t as

$$Z(t) = \sum_{(u_i,u_j)\in E} z_{i,j,t}, 1 \le t \le B. \tag{33}$$

Let N be the number of work modules. Constraint Equation (34), below, ensures that resource constraints are satisfied at timestep t:

$$H(t) + Y(t) + M(t) + \left\lceil \frac{Z(t)}{k} \right\rceil \le N, 1 \le t \le B. \tag{34}$$

Including Constraint Equation (34) in the ILP significantly increases the runtime of the solver. We can run the ILP without enforcing Constraint Equation (34): if the ILP converges, then the schedule obtained may satisfy the constraint; otherwise, we can rerun the ILP with the constraint enforced. We present experimental results for both ILP variants.

7.5 Algorithm Summary

Table 2 summarizes the scheduling algorithms introduced in this section. The algorithms summarized in the table each have their strengths and weaknesses: the heuristics run faster than the iterative improvement algorithms and ILP but will produce lower quality solutions in most cases. PS and GA-PS are specialized for trees and forests, while the others can schedule any DAG,

regardless of topology. These descriptions are not absolute. For example, a heuristic may be able to find the optimal schedule for many problem instances. Likewise, given a fixed amount of time, the ILP may not converge to an optimal schedule, or even find a legal schedule at all, while a heuristic obtains a good quality schedule in far less time. The experimental analysis, presented in the following section, provides a quantitative comparison of the scheduling algorithms listed in Table 2 using 34 different benchmark DAGs.

8 SIMULATION RESULTS

8.1 Simulation Platform

Our evaluation platform is a desktop PC with an Intel I7-4720HQ processor running at 2.6GHz, 16GB of DDR3 memory, with an 8GB SSD and 1TB hard disk. Our experiments were performed using an open source DMFB synthesis tool and simulator released by our group (Grissom et al. 2012, 2015). All scheduling algorithms were implemented by us in C++ and integrated into the simulator. Genetic algorithms run for 1000 generations. ILPs were generated and solved using (LPSolve]; the ILP-based schedulers were integrated into the framework and produce workable solutions that can be processed downstream, for example, to obtain placement and routing results. The source code for all of the scheduling algorithms described here is publicly available [UCRa], except for GA-PS and the ILPs, we plan to release at the time of publication.

Our simulations report the schedule lengths in terms of 1s timesteps, as well as the runtime of the scheduling algorithms in seconds. For the ILP, we enforce a limit of 4h (14,400s); if the ILP does not converge after four hours, then we report the shortest legal schedule found so far by the ILP (with no guarantee of optimality) or we report that the ILP failed to find a legal solution. We report results for the ILP with and without Constraint Equation (34) enforced.

All benchmark DAGs used for scheduling are available and/or can be generated by tools that we have publicly released. *PCR*, *In Vitro Diagnostics*, *Protein*, and *Protein Split* have been used by ourselves and others to evaluate DMFB synthesis algorithms (Ding et al. 2001; Su and Chakrabarty 2008; Grissom and Brisk 2012b). *Dilution and Mixing with Reduced Wastage (DMRW)* (Roy et al. 2010), *Linear Dilution Gradient (LDG)* (Bhattacharjee et al. 2013), *Dilution Tree Pruning with a Cost Matrix (DTPCM)* (Bhattacharjee et al. 2012), and *WAste Recycling Algorithm (WARA)* (Huang et al. 2013) refer to algorithmically generated DAGs that perform sample preparation tasks; the numbers next to each of these benchmarks refer to the parameter values that were passed to the algorithms to generate the DAGs. The source code for these four (and other) sample preparation algorithms is publicly available [UCRb].

All of our experiments target a 15×19 DMFB with different configurations in terms of I/O ports and specialized modules:

For PCR, the DMFB featured 6 detectors, 8 input ports (top: tris-hcl, kcl, bovine, gelatin; left: primer, beosynucleotide, amplitag, lambda) and one output port (right).

For the Protein and Protein Split assays, the DMFB features 8 detection units and 5 inputs (top: DsS, DsB, DsB, DsR; left: DsR) and 1 output port (right).

For the remaining assays, the DMFB featured 8 detectors, 12 inputs (top: 6 inputs; left: 6 inputs) and 1 output port (right). For the *in vitro* diagnostics assays, 4 inputs on top were used for Plasma, Serum, Saliva, and Urine, and 4 inputs on the left were used for Glucose, Lactate, Pyruvate, and Glutamate, as needed. The largest of the mixing/dilution tree assays used all 12 inputs.

8.2 Scheduling Results

Table 3 reports the results of the different scheduling algorithms. Benchmarks for which a given scheduler could not find a legal solution are denoted as "Fail"; the situation where the ILP converged after 4h while finding a legal, but not necessarily optimal, solution are denoted as

Table 3. Scheduling Results in Terms of 1s Timesteps for Each of the Scheduling Algorithms Reported Here

Benchmark DAG	LS	FDLS	PS	GA-LS1	GA-LS2	GA-PS	ILP	ILP + Constr. (34)
PCR	**12**	**12**	**12**	**12**	**12**	**12**	**12**	**12**
InVitro 2s,2r	**15**	**15**	**15**	**15**	**15**	**15**	**15**	**15**
InVitro 2s,3r	19	**17**	**17**	**17**	**17**	**17**	**17**	**17**
InVitro 3s,3r	23	19	21	**18**	**18**	19	(Feas.) 19	(Feas.) 19
InVitro 3s,4r	25	24	27	**21**	**21**	**21**	(Feas.) 25	(Feas.) 25
InVitro 4s,4r	33	33	35	**29**	**29**	**29**	(Feas.) 34	(Feas.) 34
Protein	116	99	119	101	**98**	117	Fail	Fail
Protein Split 1	**53**	**53**	55	**53**	**53**	55	**53**	**53**
Protein Split 2	63	63	70	63	63	70	**58**	Fail
Protein Split 3	116	**99**	119	101	111	117	Fail	Fail
Protein Split 4	264	530	218	242	235	**212**	Fail	Fail
Protein Split 5	696	Fail	**418**	696	635	**418**	Fail	Fail
Protein Split 6	Fail	Fail	**864**	Fail	Fail	**864**	Fail	Fail
Protein Split 7	Fail	Fail	**1,796**	Fail	Fail	**1,796**	Fail	Fail
DMRW 5	**8**	**8**	**8**	**8**	**8**	**8**	**8**	**8**
DMRW 25	**13**	**13**	**13**	**13**	**13**	**13**	**13**	**13**
DMRW 125	**18**	**18**	**18**	**18**	**18**	**18**	**18**	**18**
DMRW 0625	**23**	**23**	**23**	**23**	**23**	**23**	**23**	**23**
DMRW 03125	**28**	**28**	**28**	**28**	**28**	**28**	**28**	**28**
DMRW 0434782609	55	**53**	Fail	**53**	**53**	Fail	**53**	**53**
DMRW 124023437	59	54	Fail	54	54	Fail	**53**	**53**
DMRW 1015625	40	**38**	Fail	**38**	**38**	Fail	**38**	**38**
DMRW 3056640625	58	54	Fail	54	54	Fail	**53**	**53**
DMRW 5009765625	61	54	Fail	54	58	Fail	**53**	**53**
DMRW 0223	55	**53**	Fail	**53**	**53**	Fail	**53**	**53**
LDG 11,10,6,5	63	**52**	Fail	**52**	54	Fail	Fail	Fail
LDG 13,12,8,5	85	69	Fail	**65**	69	Fail	(Feas.) 67	(Feas.) 67
LDG 10,8,6,5	53	47	Fail	47	47	Fail	**43**	**43**
DTPCM 2,7,9	**23**	**23**	25	**23**	**23**	25	**23**	**23**
DTPCM 1,3,8	**28**	**28**	Fail	**28**	**28**	Fail	**28**	**28**
DTPCM 3,4,7	**18**	**18**	20	**18**	**18**	20	**18**	**18**
WARA 43,67,123,256	67	53	Fail	**49**	55	Fail	(Feas.) 51	(Feas.) 51
WARA 27,59,223,256	61	53	Fail	49	50	Fail	**(Feas.) 47**	**(Feas.) 47**
WARA 57,128,251,256	56	48	Fail	**46**	48	Fail	(Feas.) 48	(Feas.) 48

Note: The best schedule found for each benchmark is bolded and underlined. For the ILP and ILP + Constraint Equation (34) category, "Feas." denotes a legal, but not necessarily optimal schedule found after 4h of running.

"(Feas.)"ible. The best result for each benchmark (which may be found by multiple algorithms) is underlined and bolded. We can only confirm that the best result is optimal in cases where the ILP converged.

LS: LS found legal schedules for 32 of the 34 benchmark DAGs. Collectively, FDLS and PS dominate LS: in all cases, at least one of the two generates a schedule at least as short as LS. LS produces a shorter schedule than FDLS for Protein Split 4; for Protein Split 5, FDLS fails while LS produces a legal schedule. For the trees and forests that PS can schedule, LS produced shorter schedules for eight benchmarks.

FDLS vs. LS: As noted above, FDLS produces schedules no longer than LS for all but one benchmark. The only difference between the two algorithms is the objective function, thus it is

worthwhile to consider the difference. The CPP priority function, Equation (11), employed by LS considers the number of vertices on the longest path from vertex v to a primary output, not the respective latencies of the vertices on the path. Intuitively, LS schedules vertices in ascending order of CPP. Scheduling vertices with small CPP values as a general strategy tends to schedule vertices in their fanout cones early as well. This tends to eliminate contention for storage and other resources, freeing up spatial area to execute other assay operations.

In contrast, the FauxForce objective functions employed by FDLS, Equations (16) and (17), indirectly account for latencies as part of the Slack computation in Equation (13), as well as resource contention at each timestep t via the $P(v, t)$ and $Q(t)$ functions, Equations (14) and (15), respectively. This allows FDLS to make better scheduling decisions than LS on a vertex-by-vertex basis, with the exception of Protein Split 4 and 5. These DAGs have 4- and 5-level fanout trees, each of which culminates in a long path of dilution operations followed by an optical detection (Su and Chakrabarty 2008; Figure 25]. The sequence of operations on each path are identical, thus the graph is highly symmetric, as evidence by Figure 7(a). The kth vertex on each path will have identical slack and $P(v, t)$ values; as their slacks are identical, they will share the same $Q(t)$ and have identical FauxForce values at each timestep. Due to these symmetries, FDLS cannot effectively distinguish between vertices and runs into trouble as the problem instances get more constrained (adding one more level of splits doubles the number of paths, while DMFB size remains constant). This provides LS with an advantage in these particular cases.

PS vs. LS: LS retains an advantage for DAGs other than trees or forests, which PS cannot schedule. For trees and forests, PS retains an advantage in situations where the spatial parallelism provided by the DMFB is less than the spatial parallelism in the DAG itself (e.g., the larger Protein Splits). By scheduling paths contiguously without storage operations, PS performs better in these cases; LS, in contrast, may inadvertently issue too many vertices from many paths at once, leading to reduced spatial parallelism and/or failure due to poor storage choices, as illustrated by Figures 7(c) and 7(d). On the other hand, when spatial parallelism is abundant, PS' decision to schedule paths contiguously may be sub-optimal, as this can delay the starting times of operations early on the paths. In these cases, LS's more aggressive vertex-by-vertex approach to scheduling leads to shorter schedules.

PS vs. FDLS: PS produced a shorter schedule than FDLS for Protein Split 4, and succeeded where FDLS failed for Protein Split 5–8; in all other cases, FDLS produced schedules no longer than PS. The cases where PS and FDLS produced schedules of equal length tended to be relatively easy benchmarks (PCR, the smaller *in vitros*, the DMRWs) where LS was competitive as well. Overall, FDLS tends to be more effective than PS, with the exception of the medium-to-large Protein Splits; the analysis is similar to PS vs. LS, noting that (1) FDLS tended to produce shorter schedules than LS; and (2) the two exceptions were Protein Split 4 and 5, where PS is superior to LS.

GA-LS1, GA-LS2, and GA-PS: The iterative improvement heuristics retain the advantage over LS, FDLS, and PS of being able to randomly generate a large number of schedules, as opposed to converging to just one. In most cases, they were able to improve upon the respective initial schedules provided by LS and PS, except for relatively easy benchmarks where LS and PS produced optimal solutions upfront.

One notable case is Protein Split 3, where FDLS produced a shorter schedule than any of the GAs. In this case, all GAs were able to improve on the initial schedules provided by LS and PS. This benchmark was clearly challenging, because neither ILP variant was able to find a feasible schedule within the 4h time limit. Given enough time, GA-LS1 and/or 2 could have randomly generated a vertex priority order for vertices that would be the same as the priority order computed by FDLS.

GA-LS1 and 2 failed for benchmarks where LS failed; the same is true for GA-PS and PS; this is because initial schedules are needed to start the iterative improvement process. It is also worth noting that all three GA algorithms are limited in the sense that they call greedy heuristics; if there exists a high-quality schedule for some benchmark that could not be found using the LS or PS frameworks (with any priority function), then the GAs would not be able to find it.

GA-LS1 vs. GA-LS2: The primary difference between these two algorithms is an encoding for GA-LS2 introduced by Ricketts et al. (2006), to favor DAGs with multiple connected components, for example, the *in vitro* benchmarks; in our experiments, GA-1 and GA-2 found optimal solutions for all five *in vitro* benchmarks, calling into question the value of this encoding (perhaps GA-2 would retain an advantage when the number of iterations is constrained. For the remaining benchmarks, which have one connected component, scheduling differences between GA-LS1 and –LS2 is essentially noise.

ILP: The two ILP variants, with and without Constraint Equation (34) enforced, produced identical results in all but one case: Protein Split 2, where the ILP without the constraint converged to a legal and optimal solution, while the ILP with the constraint enforced did not converge after 4h. In all but one case where the ILPs produced feasible but sub-optimal schedules, at least one of the six deterministic heuristics or iterative improvement metaheuristics was able to produce a shorter schedule. Both ILPs were able to find optimal solutions for most of the benchmark DAGs.

8.3 Scheduler Execution Time

Table 4 reports the execution times of the scheduling algorithms. In most cases, the three heuristics ran the fastest. For trees and forests, PS ran the fastest, because it computed its priority function on the granularity of paths, whereas LS and FDLS compute a priority function for each vertex (the same observation holds when comparing the runtime of GA-PS to GA-LS1 and GA-LS2, with a few exceptions). FDLS ran slower than LS, because its priority function is substantially more complicated.

As expected, the iterative improvement heuristics ran much slower than LS, FDLS, and PS, because they compute many more schedules. Disparities in runtimes between GA-LS1 and GA-LS2 are primarily due to random effects, for example, getting stuck at local minima and terminating early. In the case of DTPCM 3,4,7, GA-LS1 and GA-LS2 found the optimal solution (18 timesteps), while GT-LS1 converged 8.5× faster than GA-LS2. For this particular benchmark, the ILPs converged to optimal solutions several orders of magnitude faster than any of the evolutionary metaheuristics.

The inherent strength of the ILP is that it yields optimal schedules; however, this cannot be done efficiently unless it is proven that P = NP. The convergence time of the ILP is certainly too slow for use in an online context and likely too slow for use in an offline context for all but the smallest problem instances. Removing constraint Equation (34) can reduce the runtime of the ILP for many problem instances (the one exception being LDG 10,8,6,5); however, this reduction cannot overcome the inherent exponential worst-case runtime of the ILP solver. In one case (Protein Split 2), removing Constraint Equation (34) was the difference between converging and failure to converge within the 4h time limit. In practice, it may be allowable to run the ILP for a user-selected time period, reverting to heuristic methods in cases where the ILP does not converge.

9 CONCLUSION

We have evaluated the performance of eight DMFB scheduling algorithms on 34 benchmark DAGs. Our results empirically demonstrate that different algorithms have different strengths in terms of finding optimal and near-optimal solutions; ignoring the ILPs, which clearly suffer from

Table 4. Execution Time (ms) of the Different Scheduling Algorithms Reported in Table 2 Values of 0 Indicate Execution Times of Less than 1ms

Benchmark DAG	LS	FDLS	PS	GA-LS1	GA-LS2	GA-PS	ILP	ILP + Constr. (34)
PCR	0	1	0	62,088	57,286	7,009	48	77
InVitro 2s,2r	2	7	0	34,127	33,114	10,764	104	142
InVitro 2s,3r	3	9	1	60,661	60,036	22,184	767	1,020
InVitro 3s,3r	6	20	2	102,073	100,509	34,681	14,400,000	144,00,000
InVitro 3s,4r	9	35	3	134,273	160,823	51,013	14,400,000	14,400,000
InVitro 4s,4r	28	75	4	270,870	266,619	69,387	14,400,000	14,400,000
Protein	5	202	2	80,934	82,477	56,920	14,400,000	14,400,000
Protein Split 1	0	7	0	12,528	12,268	7,993	1,074	3,589
Protein Split 2	1	37	1	28,805	27,880	19,827	5,140	Fail
Protein Split 3	5	194	2	82,642	81,450	48,247	Fail	Fail
Protein Split 4	14	1,073	5	1,801,105	262,834	Fail	Fail	Fail
Protein Split 5	41	5,677	11	Fail	Fail	Fail	Fail	Fail
Protein Split 6	Fail	Fail	23	Fail	Fail	Fail	Fail	Fail
Protein Split 7	Fail	Fail	50	Fail	Fail	Fail	Fail	Fail
DMRW 5	1	1	0	3,547	3,559	2,015	8	23
DMRW 25	0	0	0	4,479	4,527	2,640	17	61
DMRW 125	0	1	0	6,249	6,477	3,248	31	116
DMRW 0625	0	2	0	6,362	6,410	3,882	57	241
DMRW 03125	0	18	0	7,289	7,551	4,542	95	429
DMRW 0434782609	3	28	Fail	16,057	28,802	Fail	876	3,447
DMRW 124023437	3	55	Fail	23,991	27,412	Fail	1,207	4,344
DMRW 1015625	1	14	Fail	10,842	12,751	Fail	226	1,047
DMRW 3056640625	6	71	Fail	13,619	41,756	Fail	1,749	6,237
DMRW 5009765625	4	73	Fail	46,158	50,543	Fail	1,162	4,756
DMRW 0223	3	34	Fail	16,263	21,735	Fail	991	7,781
LDG 11,10,6,5	31	513	Fail	1,800,015	782,048	Fail	14,400,000	14,400,000
LDG 13,12,8,5	51	1,259	Fail	1,722,782	1,594,602	Fail	14,400,000	14,400,000
LDG 10,8,6,5	18	324	Fail	945,774	1,201,310	Fail	402,299	252,910
DTPCM 2,7,9	5	26	0	261,951	262,525	7,501	406	589
DTPCM 1,3,8	5	20	Fail	185,888	175,891	Fail	425	525
DTPCM 3,4,7	3	7	1	22,741	193,394	132,922	245	249
WARA 43,67,123,256	7	434	Fail	835,011	859,294	Fail	14,400,000	14,400,000
WARA 27,59,223,256	38	706	Fail	1,062,527	715,725	Fail	14,400,000	14,400,000
WARA 57,128,251,256	6	459	Fail	596,180	610,351	Fail	14,400,000	14,400,000

tractability issues, all of the heuristics other than LS find shorter schedules than the others for at least one benchmark. Deterministic polynomial-time heuristics (LS, FDLS, PS) are unlikely to match the evolutionary metaheuristics in terms of solution quality for challenging problem instances; however, in laboratory situations where compute time is more important than schedule quality, it should be possible to run all three to select the best result. Although we cannot claim that any one or two schedulers are clearly superior to the others, our results demonstrate the benefit of having a library of high quality algorithms and heuristics at our disposal. Future work will investigate better quality heuristics.

REFERENCES

Mirela Alistar and Paul Pop. 2015. Synthesis of biochemical applications on digital microfluidic biochips with operation execution time variability. *Integr.: VLSI J.* 51, C (Sept. 2015), 158–168.

James C. Bean. 1994. Genetic algorithms and random keys for sequencing and optimization. *ORSA J. Comput.* 6, 2 (May 1994) 154–160.

Karl F. Böhringer. 2006. Modeling and controlling parallel tasks in droplet-based microfluidic systems. *IEEE TCAD* 25, 2 (Feb. 2006), 334–344.

Stephen P. Bradley, Arnoldo C. Hax, and Thomas L. Magnanti. 1977. *Applied Mathematical Programming. Addison-Wesley*, 272–319.

Sukanta Bhattacharjee, Ansuman Banerjee, and Bhargab B. Bhattacharya. 2012. Multiple dilution sample preparation using digital microfluidic biochips. In *Proceedings of ISED*. 188–192.

Sukanta Bhattacharjee, Ansuman Banerjee, Tsung-Yi Ho, Krishendu Chakrabarty, B. B. Bhattacharya. 2013. On producing linear dilution gradient of a sample with a digital microfluidic biochip. In *Proceedings of ISED*. 77–81.

Giovanni De Micheli. 1994. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, Inc.

Manjeet Dhindsa, Stein Kuiper, and Jason Heikenfeld. 2011. Reliable and low-voltage electrowetting on thin parylene films. *Thin Solid Films* 519, 10 (Mar. 2011) 3346–3351.

Jie Ding, Krishnendu Chakrabarty, and Richard B. Fair. 2001. Scheduling of microfluidic operations for reconfigurable two-dimensional electrowetting arrays. *IEEE TCAD* 20, 12, (Dec. 2001), 1463–1468.

Daniel Grissom and Philip Brisk. 2012a. Fast online synthesis of generally programmable digital microfluidic biochips. In *Proceedings of CODES+ ISSS*, Tampere, Finland, 413–422.

Daniel Grissom and Philip Brisk. 2012b. Path scheduling on digital microfluidic biochips. In *Proceedings of DAC*. 26–35.

Daniel Grissom and Philip Brisk. 2014. Fast online synthesis of digital microfluidic biochips. *IEEE TCAD* 33, 3 (Mar. 2014) 356–369.

Daniel Grissom, Christopher Curtis, and Philip Brisk. 2014. Interpreting assays with control flow on digital microfluidic biochips. *ACM JETC* 10, 3, Article 24 (Apr. 2014).

Daniel Grissom, Kenneth oneal, Benjamin Preciado, Hiral Patel, Robert Doherty, Nick Liao, and Philip Brisk. 2012. A digital microfluidic biochip synthesis framework. In *Proceedings of VLSI-SoC*. 7–12.

Daniel Grissom. 2015. An open-source compiler and PCB synthesis tool for digital microfluidic biochips. *Integr.: VLSI J.* 34, 12 (Sep. 2015), 169–193.

B. Hadwen, G. R. Broder, D. Morganti, A. Jacobs, C. Brown, J. R. Hector, Y. Kubota, and H. Morgan. 2012. Programmable large area digital microfluidic array with integrated droplet sensing for bioassays. *Lab-on-a-Chip* 12, 18 (Sep. 2012) 3305–3313.

Juinn-Dar Huang, Chia-Hung Liu, and Huei-Shan Lin. 2013. Reactant and waste minimization in multi- target sample preparation on digital microfluidic biochips. *IEEE TCAD* 32, 10, (Oct. 2013), 1484–1494.

Lian-Xin Huang, Bonhye Koo, and C.-J. Kim. 2012. Evaluation of anodic Ta_2O_5 as the dielectric layer for EWOD devices. In *Proceedings of IEEE MEMS Conference*. 428–431.

Christopher Jaress, Philip Brisk, and Daniel Grissom. 2015. Rapid online fault recovery for cyber-physical digital microfluidic biochips. In *Proceedings of VTS*, Napa, CA, 1–6.

Oliver Keszocze, Robert Wille, Tsung-Yi Ho, and Rolf Drechsler. 2014. Exact one-pass synthesis of digital microfluidic biochips. In *Proceedings of DAC*, San Francisco, CA, USA, 1–6.

Chia-Hung Liu, Kuang-Cheng Liu, and Juinn-Dar Huang. 2013. Latency-optimization synthesis with module selection for digital microfluidic biochips. In *Proceedings of SOCC*, Erlangen, Germany 159–164.

LPSolve Sourceforge: LPSolve Version 5.5.2.0 Reference Guide, Retrieved Mar. 08, 2016, from LPSolve Retrieved from http://lpsolve.sourceforge.net/5.5/.

Lingzhi Luo and Srinivas Akella. 2011. Optimal scheduling of biochemical analyses on digital microfluidic systems. *IEEE T-ASE* 8, 1, (Jan. 2011), 216–227.

Yan Luo, Krishnendu Chakrabarty, and Tsung-Yi Ho. 2013. Error recovery in cyberphysical digital microfluidic biochips. *IEEE TCAD* 32, 1 (Jan. 2013), 59–72.

Yan Luo, Krishnendu Chakrabarty, and Tsung-Yi Ho. 2014. Biochemistry synthesis on a cyberphysical digital microfluidics platform under completion-time uncertainties in fluidic operations. *IEEE TCAD* 33, 6 (June 2014), 903–916.

Elena Maftei, Paul Pop, and Jan Madsen. 2010. Tabu search-based synthesis of digital microfluidic biochips with dynamically reconfigurable non-rectangular devices. *Des. Autom. Embed. Sys.* 14, 3 (Sept. 2010) 287–307.

Elena Maftei, Paul Pop, and Jan Madsen. 2013. Module-based synthesis of digital microfluidic biochips with droplet-aware operation execution. *ACM JETC* 9, 1, Article 2 (Feb. 2013).

Joo Hyon Noh, Jiyong Noh, Eric Kreit, Jason Heikenfeld, and P. D. Rack. 2012. Toward active-matrix lab-on-a-chip: programmable electrofluidic control enabled by arrayed oxide thin film transistors. *Lab-on-a-Chip* 12, 2 (Jan. 2012), 353–360.

Kenneth O'Neal, Daniel Grissom, and Philip Brisk. 2012. Force-directed list scheduling for digital microfluidic biochips. In *Proceedings of VLSI-SoC,* Santa Cruz, CA, 7–12.

Phil Paik, Vamsee K. Pamula, and Richard B. Fair. 2003. Rapid droplet mixers for digital microfluidic systems. *Lab-on-a-Chip* 3, 4 (Nov. 2003), 253–259.

Pierre G. Paulin and John P. Knight. 1989. Force-directed scheduling for the behavioral synthesis of ASICs. *IEEE TCAD* 8, 6 (Aug. 1989), 661–679.

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

Andrew J. Ricketts et al. 2006. Priority scheduling in digital microfluidics-based biochips. In *Proceedings of DATE*, Munich, Germany, 239–334.

Sudip Roy, Barghab B. Bhattacharya, and Krishnendu Chakrabarty. 2010. Optimization of dilution and mixing of biochemical samples using digital microfluidic biochips. *IEEE TCAD* 29, 11 (Nov. 2010) 1696–1708.

Fei Su and Krishnendu Chakrabarty. 2006. Module placement for fault-tolerant microfluidics-based biochips. *ACM TODAES* 11, 3, (July 2006) 682–710.

Fei Su and Krishnendu Chakrabarty. 2008. High-level synthesis of digital microfluidic biochips. *ACM JETC* 3, 4, Article 1 (Jan. 2008).

Fei Su, William Hwang, and Krishnendu Chakrabarty. 2006. Droplet routing in the synthesis of digital microfluidic biochips. In *Proceedings of DATE*, Munich, Germany, 323–328.

UCRa. Digital Microfluidic Biochip Static Synthesis Simulator. Retrieved from http://microfluidics.cs.ucr.edu/staticsim.html.

UCRb. Dilution Framwork. Retrieved from http://microfluidics.cs.ucr.edu/DilutionFramework.html.

Wim F. J. Verhaegh et al. 1995. Improved force-directed scheduling in high-throughput digital signal processing. *IEEE TCAD* 14, 8 (Aug. 1995) 945–960.

Tao Xu and Krishnendu Chakrabarty. 2008. Integrated droplet routing and defect tolerance in the synthesis of digital microfluidic biochips. *ACM JETC* 4, 3, Article 11 (Aug. 2008).

Tao Xu, Krishnendu Chakrabarty, and Fei Su. 2008. Defect-aware high-level synthesis and module placement for microfluidic biochips. *IEEE TBCAS* 2, 1 (Mar. 2008), 50–62.

Ping-Hung Yuh, Chia-Lin Yang, and Yao-Wen Chang. 2007. Placement of defect-tolerant digital microfluidic biochips using the T-tree formulation. *ACM JETC* 3, 3, Article 13 (Nov. 2007).

Yang Zhao and Krishnendu Chakrabarty. 2012. Cross-contamination avoidance for droplet routing in digital microfluidic biochips. *IEEE TCAD* 31, 6 (June 2012) 817–83.

Received July 2016; revised May 2017; accepted May 2017