Automatic Volume Management for Programmable Microfluidics

Ahmed M. Amin, Mithuna Thottethodi, T. N. Vijaykumar
Steven Wereley[*], and Stephen C. Jacobson[Υ]

School of Electrical & Computer Engineering, Purdue University
[*]School of Mechanical Engineering, Purdue University
[Υ]Department of Chemistry, Indiana University
{amamin, mithuna, vijay, wereley}@purdue.edu, jacobson@indiana.edu

Abstract

Microfluidics has enabled lab-on-a-chip technology to miniaturize and integrate biological and chemical analyses to a single chip comprising channels, valves, mixers, heaters, separators, and sensors. Recent papers have proposed programmable labs-on-a-chip as an alternative to traditional application-specific chips to reduce design effort, time, and cost. While these previous papers provide the basic support for programmability, this paper identifies and addresses a practical issue, namely, fluid volume management. Volume management addresses the problem that the use of a fluid depletes it and unless the given volume of a fluid is distributed carefully among all its uses, execution may run out of the fluid before all its uses are complete. Additionally, fluid volumes should not overflow (i.e., exceed hardware capacity) or underflow (i.e., fall below hardware resolution). We show that the problem can be formulated as a linear programming problem (LP). Because LP's complexity and slow execution times in practice may be a concern, we propose another approach, called DAGSolve, which over-constrains the problem to achieve linear complexity while maintaining good solution quality. We also propose two optimizations, called cascading and static replication, to handle cases involving extreme mix ratios and numerous fluid uses which may defeat both LP and DAGSolve. Using some real-world assays, we show that our techniques produce good solutions while being faster than LP.

Categories and Subject Descriptors
D.3 [**Programming Languages**]: Compilers
General Terms: Algorithms, Design
Keywords: Microfluidics, Programmable lab-on-a-chip, Fluid volume management

1 Introduction

Microfluidics is the field of handling fluids in small quantities, typically at the scale of nano or pico liters. Microfluidics has miniaturized and integrated channels, valves, mixers, heaters, and separators to enable chemical and biological analyses in a single chip called a lab-on-a-chip (LoC). LoCs enable faster, cheaper and higher-precision analyses over the traditional bench-scale methods. To date, LoCs have been applied in diverse industrial and academic domains, such as drug discovery, virology, clinical applications, genomics, biochemistry and chemical synthesis.

Thus far LoCs have been designed as application-specific chips which closely map an *assay* (fluidic algorithm) to the LoC hardware. Assays are analogous to conventional computer programs, where fluids correspond to variables, and operations such as mix, incubate, separate and sense manipulate input fluids to generate new fluids. The application-specific approach leads to considerable design effort, turn-around time, and cost. To address these limitations, Urbanski et al. [13] pioneer the idea of *programmable* labs-on-a-chip (PLoC)[1], and introduce a new programming language for microfluidics called Biostream [10]. In prior work, we proposed a comprehensive instruction set called AquaCore Instruction Set (AIS), and a microarchitecture called AquaCore [2].

While the previous papers provide the basic support for programmability, this paper identifies and addresses a practical issue, namely, *fluid volume management*. The issue of fluid volume management arises because fluids have a fixed total volume, and the use of a fluid (variable) depletes it. If there are many uses of a fluid, the given volume of the fluid must be distributed carefully among the uses to prevent execution from running out of the fluid before all of the uses occur. This distribution poses a challenge when the uses require different proportions of volumes as is the case when a fluid is mixed with different other fluids in different ratios (*e.g.*, one use for a fluid is in a mix ratio of 1:2 while another use for the same fluid is in a mix ratio of 1:10). The destructive nature of fluid uses is fundamentally different than that of use of variable values in conventional computers where uses (*i.e.*, reads) are non-destructive. This difference indicates the novelty of the problem, motivating the need for volume management. Dealing with destructive uses is further complicated by low-level, implementation-dependent details of the fluidic hardware, such as maximum capacity (of reservoirs and functional units) and minimum fluid transport resolution (imposed by the fluid transport/ handling hardware). Forcing the programmer to handle these constraints would diminish the practicality of PLoCs. Consequently, we handle this issue automatically using a combination of the compiler and run-time system.

Biostream has proposed a *reactive* approach for volume management, called regeneration [10]. Regeneration allows the fluid to run out and re-generates the fluid just before the next use by re-executing the code fragments that produce the fluid (*i.e.*, the backward slice [11]). While elegant in theory, regeneration may place a high or unbounded demand on LoC resources. Mimicking unbounded resources through virtualization is feasible in conventional computers, but microfluidic technology is not yet at that

---

1. We clarify that PLoC's programmability is software programmability (like microprocessors), and not burn-in programmability (like FPGAs).

level of maturity. Even when regeneration is feasible, regeneration re-executes fluidic instructions (in the fluidic datapath) which are slow and are likely to incur overhead (PLoCs use a heterogeneous organization where the datapath is fluidic and control is electronic and orders-of-magnitude faster [10][2]). In contrast, we take the *pro-active* approach of reducing the chances of running out of a fluid (but not eliminating it, as we explain later). Thus, our approach mostly avoids regeneration's overhead while regeneration can be used as our back-up when our approach cannot avoid fluid running out.

The key challenge is ensuring that not only all the direct uses of a given fluid but also all the uses of fluids indirectly derived from the given fluid satisfy their respective usage proportions. For instance, an assay mixes fluids a and b in some ratio to produce fluid c and then mixes fluids a and c in some other ratio to produce fluid d, which is later used to produce fluid e. Thus, we need to ensure that our distribution of a's volume among its uses results in enough volume of d for its later use. Additionally, fluid volumes should not overflow (*i.e.*, exceed hardware capacity) or underflow (*i.e.*, fall below hardware transport resolution), and must be an *integer* multiple of the minimum hardware transport resolution. We call this volume-assignment problem *Integer Volume Management (IVol)*. At present, the complexity of IVol is unknown and is left as an open question for future work. However, we show that IVol can be cast as an integer linear programming (ILP) problem. This formulation is our first contribution. Because ILP is NP-Hard, and because the underlying chemistry in inherently tolerant of minor inaccuracies in mix ratios, we provide a rational formulation of the problem, called *Rational Volume Management (RVol)*, and round the resulting volume assignment to integers. To solve RVol, we employ linear programming (LP) instead of ILP. However, for L-bit inputs and n variables, LP has a worst-case asymptotic complexity of $O(n^3 L)$ [4]. While such complexity may be tolerable for modest-size assays at compile-time, there are cases where we have to solve this problem at run-time because volumes of some intermediate fluids may not be available at compile time (*e.g.*, the volume yielded by a separate-by-size step may not be known *a priori*). In such cases, LP's complexity may be a concern despite the fact that run-time computations can be done in the PLoC's fast electronic control. While one may think that LP solvers are fast in practice despite LP's worst-case complexity, we show that LP is significantly slow for real-world assays. Consequently, we wish to rely on LP at run-time only if there are no other alternatives.

We propose one such alternative approach for RVol, called *DAGSolve*, which artificially over-constrains the problem to achieve a linear-complexity solution. Because of its artificial constraints, DAGSolve may, in some uncommon cases, not find a solution even though LP may. In our experiments with real-world assays, DAGSolve was always successful, as we show in Section 4. Therefore, DAGSolve is a more suitable run-time option than LP. DAGSolve is our second contribution. For the uncommon case where DAGSolve fails to find a feasible solution, we fall back on LP, which itself may not find a solution and is backed up by BioStream's regeneration because it is better to provide a slow solution than no solution.

There are corner cases, such as extreme mix ratios (*e.g.*, mix ratio of 1:1000) or numerous uses of a fluid (*e.g.*, 7 uses of a fluid that has a volume of 6 multiples of minimum hardware transport resolution), that can defeat both LP and DAGSolve. For instance, a mix ratio of 1:399 using hardware with maximum and minimum capacities of 100 and 1 units, respectively, would cause either an underflow or an overflow. To handle extreme ratios, we employ *cascading*, a well-known idea in life sciences, to break an extreme ratio into two or more cascaded ratios (*e.g.*, break 1:399 into 1:19 followed by 1:19). Finally, because of hardware limit of maximum capacity, numerous uses of a fluid can cause underflow even if we produce as much volume of the fluid as possible without causing overflow. In such cases, we produce extra volume by replicating the backward slice of the fluid's production. The two optimizations of applying cascading and static replication to assays are our third contribution.

To implement our ideas, we build a simple compiler and a software run-time system which we describe in the paper. We evaluate our ideas on real-world assays and show that our techniques are effective in avoiding fluid overflow and underflow.

The rest of the paper is organized as follows. Section 2 presents some background on PLoCs and related work. Section 3 discusses the ILP formulation, DAGSolve, and its extensions. We show our compiled code and present our results in Section 4, and conclude in Section 5.

2 Background

There are two primary technologies for microfluidic devices: flow-based and droplet-based. Flow-based technologies rely on continuous or discrete fluid manipulation, while droplet-based technologies manipulate free-standing droplets of fluids. We focus on flow-based devices, though our techniques may be adapted for droplet-based LoCs.

Today's real-world application-specific LoCs typically comprise of channels (for fluid transport), valves (for flow control), reservoirs (for storage), input and output ports, and fluidic functional units, such as mixers, heaters, separators and sensors. Implementation details of these components can be found in [8]. The PLoC uses these pre-existing building blocks.

2.1 Baseline PLoC Architecture

We use AquaCore [2] as our baseline PLoC architecture, and implement our volume management techniques based on the AquaCore Instruction Set (AIS). Assays written in AIS are very similar to conventional computer assembly programs, where fluids correspond to variables, and operations such as mix, incubate, separate and sense manipulate existing fluids to generate new fluids. Though AIS may seem similar to their computer instruction sets, there are four key differences based on observations of real-world assays [2].

First, because intermediate fluids produced in assays are often used only once and usually immediately after their production, binding the fluids to storage results in unnecessarily moving the fluids from functional units to storage and back. To that end, AIS employs *storage-less operands* which are decoupled from storage so that fluids are transferred from one functional unit to another without any intervening storage (similar to [6]).

Second, assays are typically specified using *variable volumes*, unlike computational operands which use fixed-size data types and rely on padding to handle other sizes. However, fluidic operands cannot be padded easily to a fixed larger volume (by topping off) in cases where reagent concentrations have to be maintained. Accordingly, AIS instructions operate on variable volumes. All our pro-

Table 1: A subset of AIS

- **move** id1, id2, <rel vol>
- **move-abs** id1, id2, vol
- **mix** id1, time
- **separate.CE** id1, E_{sep}, len, time
- **separate.SIZE** id1, time
- **separate.AF** id1, time
- **incubate** id, temp, time
- **input** id2, id1
- **output** id2, id1
- **sense.OD** id1, senseval
- **sense.FL** id1, senseval[]
- **concentrate** id1, temp, time

active volume management techniques handles variable volumes.

Third, because most assays specify volumes only in a few steps and leave operand volumes implicit, forcing every instruction to specify its operand volume would be cumbersome. Accordingly, AIS allows operand volumes to be *implicit*. In the few steps when assays specify volumes, they usually use *relative volumes*. Therefore, AIS allows operands to optionally specify relative volumes (but also allows absolute volumes in the uncommon case). Because the use of relative volumes combined with the destructive nature of fluid usage can result in assays running out of a fluid, relative-volume operands give rise to the need for volume management.

Fourth, because microfluidics lacks the theoretical guarantees of universality (*i.e.,* the fluidic equivalent of Turing-Completeness), PLoCs must turn to experimental evaluation for coverage. While this aspect is orthogonal to this paper, we list this aspect for completeness.

A subset of the AquaCore Instruction Set (AIS) is shown in Table 1 (see [2] for the complete set). The operand *id* space includes not only reservoirs (analogous to registers), but also functional units, allowing one instruction to send its output directly to another without having to go to storage (*i.e.,* storage-less operands). Most instructions do not specify any volumes for their operands. For such instructions, the functional unit operates on the implicit volume available, and allows variable volume handling. The optional operand `<rel vol>` in `move` specifies relative volumes for transfer. The relative volumes are translated to implementation specific volumes at runtime, enabling variable volume support and increased code portability. The list includes different flavors of separate and sense (*e.g.,* `separate.CE` denotes electrophoresis-based separation, `separate.AF` separates using affinity to a reagent pre-loaded in the separator, `sense.OD` and `sense.FL` denote sensing of optical density and fluorescence, respectively).

Figure 1 shows a block diagram of the AquaCore microarchitecture consisting of heterogeneous components: a dry electronic control part and a wet fluidic datapath part. Because electronics is mature, reliable, inexpensive and can provide control functionality, AquaCore uses electronic control. The electronic dry part interprets AIS instructions and provides control signals to the wet part, and is implemented using a conventional microcontroller. As such, the dry electronic control is orders of magnitude faster than the fluidic wet part, a difference we leverage to perform some run-time computation in our volume management scheme without incurring performance overhead.

The wet part consists of fluidic functional units (FFUs) (*e.g.,* mixers, heaters, separators, and sensors), a set of reservoirs, and input and output ports. These components are connected by a set of channels. Furthermore, these components have a capacity limit, exceeding which would cause an overflow. Note that the number of reservoirs is fixed and limited, and current LoC technology does not provide a dense equivalent (such as DRAM or disk), hence careful compile-time allocation is required. Additionally, Aqua-Core uses microfluidic valves to control fluid flow which, much like electronic tri-state buffers, are central to enabling PLoC operation under program control. At each end of each channel is a microfluidic pump that effects fluid transfer from one component to another by peristalsis as described in [2]. These pumps may be used for accurate volume metering [12], which is required to handle variable volumes. Further, they impose a discrete, minimum volume transport unit, or *least count*. Finally, we note that, while several hardware techniques exist to achieve fluid transfer and metering [8], they all exhibit some form of least count.

3 Automatic Volume Management

Recall from Section 1 that volume management is needed because the use of a fluid results in the fluid's depletion, and if there are many uses of a fluid, the initial volume of the fluid must be distributed carefully among the uses to prevent execution from running out of fluid. Such distribution must obey the relative-volume proportions specified by the assay. Dealing with destructive uses is further complicated by low-level, implementation-dependent details of the fluidic hardware, such as maximum capacity and minimum transport resolution.

Fundamentally, this running out occurs because real implementations have a least count. Consider an example, where fluid A is partitioned into four portions and one of the portions is to be mixed with fluid B in the ratio 10:1 (A:B). Then if A's portion in this mix is small to start with then B may underflow (*i.e.,* the volume needed may be smaller than the least count). Burdening the programmer with the details of avoiding such underflow would defeat the purpose of PLoCs, and as such, an automatic scheme for volume management is required. However, because a fluid may have multiple uses like the above example, avoiding underflow requires automatic schemes to assign volumes such that all the uses of a fluid can be satisfied.

One may think that the example assumes four uses of A which is uncommon as fluids are usually used only once (Section 2.1). However, we note that multiple uses may lead to the correctness problem of fluid running out. As observed in computer systems research, the common case is not everything; the uncommon case must be correct.

While our concern is allocation of portions among uses to avoid running out of fluids, evaporation and residue in the fluidpath may also cause fluids to run out. Evaporation is alleviated by immersing the fluidpath in immiscible liquids (oils in most cases) [9]. Residue in the fluidpath is reasonably predictable and can be corrected by
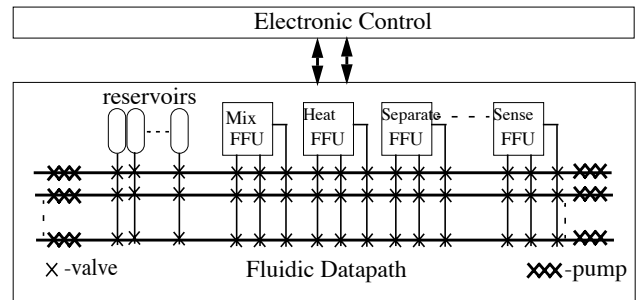

Figure 1: Microarchitecture

over-provisioning [12]. As such, our techniques ignore fluid loss.

Our problem is defined as follows. Given as assay which uses fluids in one or more instructions, in different ratios, find an absolute volume assignment for each use of each fluid, such that the assignment: 1) satisfies the usage ratio constraints as defined by the assay; 2) is an integer multiple of the least count hardware resolution; 3) does not underflow (*i.e.*, the minimum fluid use in any operation is greater than or equal to the least count); and 4) does not cause an overflow (*i.e.*, the total amount of fluid assigned in any operation is less than the hardware maximum capacity). We call this problem Integer Volume Management (IVol).

Before describing our volume management schemes, we note that in the absence of unlimited fluidic hardware resources our techniques can only minimize, but not eliminate, the possibility of underflow, just like a computer program can cause underflow in floating-point arithmetic. Such underflow cannot be prevented by the system and the assay must be rewritten to avoid underflow.

In general, there are two fundamental ways to address this problem: either by pro-actively and conservatively using fluids such that they last for the entire assay (if possible); or by reactively regenerating fluids that are exhausted. Reactive approaches may incur overhead due to regeneration's re-execution which runs on the slow fluidpath, as described in Section 1. We adopt the pro-active approach which reduces the chances of underflow and thus the number of times regeneration is needed. We fall back on regeneration when underflow is not avoided by our approach.

Volume management amounts to computing absolute volumes as long as two factors are known statically: (a) the number of uses of all fluids in an assay and (b) the output volumes (relative to the input) of all assay steps. However, if either of these factors is unknown statically and can be determined only at run-time, then computing the absolute volumes becomes more involved. The number of uses is statically known for straight line code and not known for assays with control flow. While volumes of fluids in *most* steps are known or are computable at compile time (*e.g.*, the volume in a mixture is in general the sum of volumes of the components and thermal expansion of fluids due to heating is usually too insignificant to alter volumes), there are exceptions that need online volume measurement. For instance, when mixing or heating alters the fluids' chemical nature, or, more commonly, in the case of separations where the volume of the output is unknown at compile time. Such exceptions are known to the compiler or assay writer, and we assume that the corresponding operation can be flagged (*e.g.*, using an opcode variant) so that the volumes can be measured at run-time [3].

We discuss our proposed solutions in three parts. First, we describe our DAG internal representation (Section 3.1) and our algorithms for the statically-known case (Section 3.2-Section 3.3). Second, we present extensions to DAGSolve that handle corner cases of in Section 3.4. Finally, we extend the base algorithms to support statically-unknown cases in Section 3.5.

3.1 Assay DAG Representation

We represent an assays as a simple directed acyclic graph (DAG) defined as follows. Nodes represent operations (typically volume-aggregating operations such as mixes) and edges represent true dependence among the operations. The edges are annotated with values to denote the ratio in which the source fluids is used in the operation. Input nodes have no in-bound edges and final output
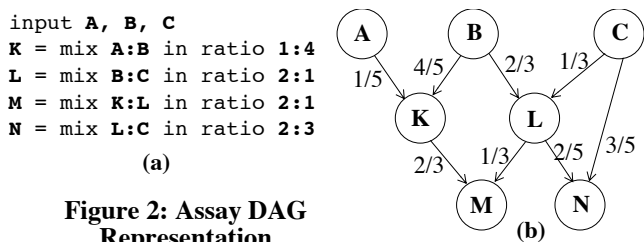
```
input A, B, C
K = mix A:B in ratio 1:4
L = mix B:C in ratio 2:1
M = mix K:L in ratio 2:1
N = mix L:C in ratio 2:3
```
(a)

Figure 2: Assay DAG Representation

nodes have no outbound edges. Figure 2 shows a simple assay and its corresponding DAG. Because all fluid uses are known and because there are no unknown output volumes (relative to input) in the assay, the DAG corresponds to the statically-known case. Assays with loops are also represented as DAGs whenever the loops can be unrolled completely, as we see later.

3.2 Integer Volume Management (IVol) and ILP Formulation

The problem of Integer Volume Management (IVol) is one of assigning volumes in integer multiples of least-count to the various uses of fluids to satisfy the assay constraints (mix ratios, number of uses) and the hardware constraints (maximum capacity and least-count). We have attempted to determine the complexity of IVol and tried casting it as some well-known problems. For example, one may think that volume management can be cast as a network flow problem. However, network flow problems have the key requirement of flow conservation at intermediate nodes (*i.e.*, the sum of all outputs of a node is equal to the sum of it's inputs), while in our case, each intermediate node potentially violates such requirement because all the produced fluid need not be used. We show later in Section 3.3 that imposing flow conservation still does not make the problem amenable to casting as a network flow problem. As such, determining the complexity of IVol is left as an open question for future work.

We show that IVol can be cast as an integer linear programming problem (ILP). We describe the ILP formulation (constraints and objective function) in the context of our DAG representation (say $G(V,E)$). The variables representing the volumes in nodes and edges are subject to the following classes of constraints with the number of constraints for each type shown parenthetically. (1) *Minimum volume* constraints ensure that no volume is smaller than the least count (one constraint per edge, $|E|$ constraints in all). In addition to the least count constraint, there may be additional minimum volume constraints for fluid functional units (e.g. separators). (2) *Maximum capacity* constraints limit the sum of volumes assigned to edges entering a node to the node's hardware capacity (one constraint per node, $|V|$ constraints in all). (3) *Non-deficit* constraints ensure that the use of a fluid (the sum of volumes of outbound edges) at a node does not exceed the volume of the fluid, which is the sum of volumes of inbound edges (one constraint for every non-output node, $|V|$ constraints at most). (4) *Ratio constraints* ensure that volumes of the inbound edges are in the specified mix ratio (one constraint for each node corresponding to a mix, $|V|$ constraints at most). (5) *Relative node output to input*, for nodes representing instructions such as separates, where the output volume of a node is not necessarily equal to the sum of its inputs, rather a relative fraction of the total input (one constraint for each node of the fractional output type, $|V|$ constraints at most). Though any feasible volume assignment that satisfies the above constraints is adequate to remove underflow, we set the objective function to
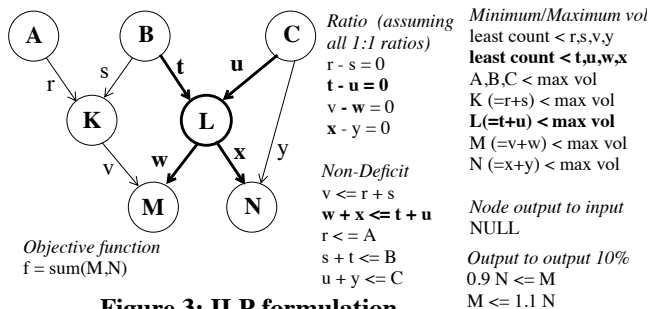
A   B   C
r s t u
K   L
v w x y
M   N

Objective function
f = sum(M,N)

Ratio (assuming all 1:1 ratios)
r - s = 0
t - u = 0
v - w = 0
x - y = 0

Non-Deficit
v <= r + s
w + x <= t + u
r <= A
s + t <= B
u + y <= C

Minimum/Maximum vol
least count < r,s,v,y
least count < t,u,w,x
A,B,C < max vol
K (=r+s) < max vol
L(=t+u) < max vol
M (=v+w) < max vol
N (=x+y) < max vol

Node output to input
NULL

Output to output 10%
0.9 N <= M
M <= 1.1 N

Figure 3: ILP formulation

maximize the sum of all output volumes to maximize the output production. In some cases, the solution may be skewed to produce very little of one output fluid and much more of another output fluid if such outputs maximize the sum of all output volumes. To avoid such skew, we add one optional set of constraints called *Relative output-to-output constraints*, where output volumes are set to be within a fixed percentage of each other (twice the number of constraints as output nodes, $|V|$ constraints at most). Note that for simplicity we bound the relative ratio of all outputs with respect to one output instead of bounding every possible pair which would result in $O(V^2)$ constraints). Figure 3 shows the optimization function and set of constraints derived from the assay shown in Figure 2.

One drawback to using ILP to solve IVol is that ILP is NP-Hard. An alternate approach to solving IVol is to use LP (polynomial complexity) to provide a non-integer solution and then round the result to an integer, which we call Rational Volume Management (RVol). One may think that practical ILP solvers are fast in practice and that the asymptotic worst-case complexity difference between ILP and LP may not necessarily translate to execution time reduction for our specific problems at typical problem sizes. However, we show later in Section 4.3 that the ILP solver we used is significantly slower than the LP solver for some of our benchmarks.

Simple rounding of the RVol results to the nearest integers may cause inaccuracies in mix ratios. Rounding up causes more input fluids to be consumed and rounding down causes less output fluids to be produced, both of which may lead to underflow. Fortunately, the underlying chemistry is inherently tolerant of small imprecisions in mix ratios, given that usual operating volumes are in the order of nanoliters and the least count is in the order of picoliters. We did not observe such underflow problems in practice and the errors for our benchmarks were below 2%. As such, we defer investigation of more sophisticated rounding techniques to the future.

While the LP-based solution to the RVol problem is feasible, we make two observations to motivate a more efficient algorithm for RVol. First, while the polynomial execution time of LP-based RVol may seem tractable for compile-time volume assignment, volume assignment must occur at run-time for statically-unknown cases, as mentioned in Section 1. LP has worst-case complexity of $O(n^3 L)$, where n is the number of variables and L is the number of bits in the binary representation of the variables [4]. Because worst-case asymptotic complexity may not be an accurate indicator of practical execution times, we show in Section 4.3 that LP can still be significantly slow for large assays. Second, as PLoCs become more widespread and their programmability matures,

composition of assays will lead to larger assays, resulting in a corresponding increase in the input to the LP problem. Next we describe our linear-complexity RVol algorithm that overcomes the above problems.

3.3 DAGSolve

To overcome the high complexity of our LP formulation to solve RVol, we make the key observation that, by adding some artificial constraints, the problem of fluid volume management can be solved in linear complexity, occasionally sacrificing a feasible solution that would be found by LP. To that end, we propose *DAG-Solve*, a linear time algorithm for fluid volume management, and fall back on LP when DAGSolve fails to find a feasible volume assignment.

DAGSolve over-constrains RVol by adding the two following constraints. First, we constrain the output volumes (say A, B and C) to be in some relative proportion to each other (say Va:Vb:Vc) while LP allows the outputs to be any volume. Note that this constraint does not fix the *absolute* volume of any output. Second, whereas LP imposes only the non-deficit constraint that can capture feasible solutions that may have excess fluids, we add a no-excess, flow-conservation constraint as well. Effectively, the flow-conservation constraint forces the generated volume for each intermediate fluid to be *equal* to the total volume of its uses.

Recall that the key distinction between the volume assignment problems and network flow problems was that flow conservation was not necessary in volume assignment. We reconsidered using the network flow approach after artificially introducing flow conservation. Some aspects of the problem, such as achieving a mix-ratio, can be expressed as multi-commodity network flow (MCNF) given absolute volumes. However, our problem requires the computation of absolute volumes from relative volumes. Further, each node in an assay creates a new fluid (commodity) due to mixing, heating or similar alteration. Therefore, each commodity traverses a single level in our DAG, unlike MCNFs. As such, we did no pursue the network-flow approach.

Our DAGSolve algorithm uses the above two additional set of constraints to reduce RVol to a simple propagation algorithm on the DAG. DAGSolve starts with the final relative output volumes (the first additional constraint), then performs a backward pass on the DAG to assign relative volumes to all nodes and edges, such that all the assay constraints (ratio constraints and relative output-to-input constraints) and the flow-conservation constraint (the second additional constraint) are satisfied. Then a forward pass assigns real absolute volumes which impose the hardware constraints (least count and maximum capacity) on the solution. The pseudocode for the algorithm is shown in Figure 4.

We define a node's V_{norm} as a measure of the fluid volume at the node relative to other nodes. Because V_{norm} is a relative measure, all nodes' V_{norm} value should be normalized to a common node or set of nodes. To that end, we choose this set of nodes to be the output nodes (leaf nodes), and apply the first artificial constraint of setting all output nodes' $V_{norm} = 1$ (*i.e.*, each output is normalized to itself and all the output volumes are equal). Though the V_{norm}s could be set to arbitrary values to produce outputs in arbitrary ratios without any other changes to the algorithm, unless we have information to prefer production of one output fluid over another, we initialize all output volumes to be equal. Similarly, we define an edge's V_{norm} as the volume of the fluid associated with

```
DAGSolve{
1. Build DAG from assay
2. Set leaf nodes' V_norm to 1 and others to 0
   //V_norm calculation
3. foreach node N in reverse topol. order of DAG
4.     foreach outbound edge E for N
5.         N.V_norm := N.V_norm + E.V_norm
6.       foreach inbound edge E for N
7.         E.V_norm := E.ratio * N.V_norm
   //Dispensing
8. Max_V_norm := maximum node V_norm
9. foreach node N and edge E in DAG
10.    N.volume := (N.V_norm*max_default)/Max_V_norm
11.    E.volume := (E.V_norm*max_default)/Max_V_norm
}
```

Figure 4: DAGSolve Algorithm

the edge normalized to the volumes of the final output fluids. V_{norm} calculation proceeds in reverse topological order of the DAG and updates the values for edges and nodes (Figure 4, lines 3-7). Each node is assigned a V_{norm} equal to the sum of its outbound edges' values (Figure 4, line 5) and then each inbound edge is updated as the product of the original edge value (the original ratio) and the node value (Figure 4, line 7). Note that setting each node's V_{norm} to the sum of the outbound edges' values is our second artificial constraint requiring flow-conservation of fluids at intermediate nodes.

Figure 5 shows the application of DAGSolve to the example given in Figure 2. Node *L* is assigned 1/3 + 2/5 = 11/15 and edges *B-L* and *C-L* are updated as 2/3*11/15 = 22/45 and 1/3*11/15 = 11/45. Figure 5(a) shows the updated values for all the nodes and edges. A node's V_{norm} may be less than one (*e.g.*, node *K*) or more (*e.g.*, node *B*) depending on their relative volumes with respect to output volume.

The next step is to assign absolute volumes based on nodes' V_{norm} (Figure 4, lines 8-11) and the hardware constraints. To avoid overflow (*i.e.*, to satisfy the maximum capacity constraint), we assign the node with the largest V_{norm} (node *B* in our example) a volume equal to the default machine-dependent maximum (*e.g.*, 100 nl). To ensure that the original ratios specified by the assay are honored, each other node and edge is assigned a fraction of the default maximum equal to the ratio of its V_{norm} to the largest V_{norm} (Figure 4, lines 10-11). Figure 5(b) shows the dispensed volumes for all the nodes and edges.

If the ratio of the largest V_{norm} to a smallest V_{norm} is less than the ratio of the default maximum to the least count, the smallest V_{norm} will underflow because its absolute volume will be less than the least count. However, this failure in DAGSolve could be the result of the additional constraints imposed by DAGSolve and does not imply that there is no feasible solution. An LP solver solving
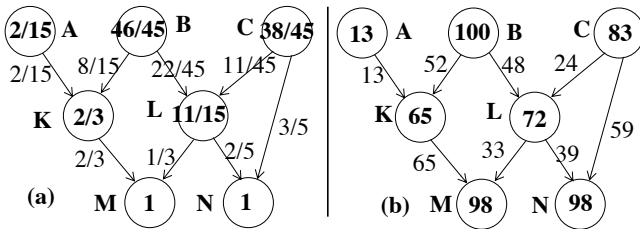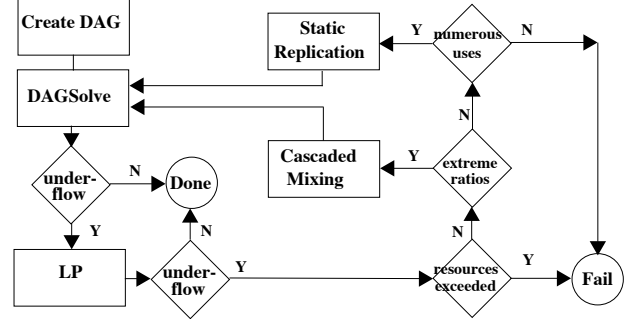

Figure 5: DAGSolve Example

Figure 6: Volume Management Flowchart

the original LP formulation (Section 3.2) without the additional constraints could still discover a feasible solution. Thus, we fall back on LP whenever our above approach leads to underflow.

In comparison with the worst-case $O(n^3L)$ complexity of LP, DAGSolve is of linear complexity as it visits each node and edge twice. Our results (Section 4) show that there is a significant difference in execution time (~80x). Thus, the use of DAGSolve as a preferred technique that may sometimes fail combined with the backing of a more robust, but slow LP solution can be viewed as a volume management hierarchy. One may think that adding the same artificial constraints to the LP formulation may, in practice, increase the efficiency of LP. However, we show in Section 4.3 that while LP does benefit in some cases from the additional constraints, the gap between DAGSolve and LP (with additional constraints) performance remains large (~60x).

3.4 Extensions to DAGSolve

Recall from Section 1, some corner cases of extreme mix ratios or numerous uses of a fluid may arise for which neither DAGSolve nor LP may produce feasible volume assignments. In this section, we discuss two techniques that modify the DAG to address the above two problems. The modified DAGs are re-processed through the same volume management hierarchy consisting of DAGSolve and LP. Figure 6 illustrates how the volume management hierarchy (on the left) interacts with our techniques to handle extreme mix ratios and numerous uses (on the right). We discuss extreme mix ratios first and numerous uses next.

3.4.1 Handling Extreme Mix Ratios — Cascading

Mix ratios that exceed the ratio of the least count to the maximum hardware capacity are infeasible to execute, and defeat DAGSolve, LP, and even regeneration. For instance, a mix ratio of *A* and *B* in a ratio of 1:1000, where the ratio of the least count to maximum capacity is 1:100 would cause either (1) an underflow of *A* if we set the volume of *B* to the hardware maximum, or (2) an overflow of *B* if we set *A* as the least count. Such extreme ratios have traditionally been handled by *cascaded mixing* where a desired mix ratio is achieved as a combination of two or more cascaded mix operations, each of which has a less skewed mix ratio. For example, a single *A-B* mixture of 1:99 can be achieved by first mixing *C= A-B* in the ratio 1:9, followed by *C-B* in the ratio of 1:9. (It may be more intuitive to think of the above example as creating a 1-part-in-100-parts mix by first creating a 1-part-*A*-in-10-parts mixture (say *C*) and then creating another 1-part-*C*-in-10-parts final mixture). Observe that the above example creates 10 parts of *C* and uses only 1 part which implies excess production, which is allowed in the LP formulation (non-deficit constraint in
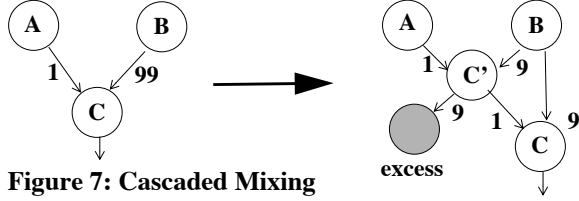
Figure 7: Cascaded Mixing

Section 3.2), but is expressly prohibited in the DAGSolve algorithm due to the flow conservation constraint. Without such excess production, cascading in DAGSolve would fail to avoid underflow in case of extreme mix ratios. However, introducing *a priori* unknown excess production leads to unknown amount of discarded output which would prevent the direct backward computation of V_{norm}s from known output volumes. In contrast, LP can handle excess production because such unknown discarded outputs correspond to independent slack variables in the LP formulation. Fortunately, we make the key observation that the fraction of the output volume to be discarded can always be computed *a priori*. In the example above, we know that exactly 9/10 parts of *C'* are discarded. Consequently, DAGSolve can incorporate this special case of excess production. The DAG transformation for excess handling in the case of the example is shown in Figure 7. We replace the original extreme-ratio mix node (*C*) with two cascaded mixes. At the intermediate node of the cascade (*C'*), we add an *excess* node and an edge from the intermediate node to the excess node that both have a V_{norm} equal to 0.9*V_{norm}(*C'*). DAGSolve is modified slightly to handle excess nodes as a special case. Unlike ordinary nodes/edges whose V_{norm}s are computed in a backward pass, the V_{norm}s of the excess edge and excess node are computed after their source node's V_{norm} is known.

In general, we use cascading only if the assay has no feasible volume assignment with the direct mix, as shown in Figure 6, When an extreme mix ratio (say 1:*R*) prevents feasible volume assignment, we apply cascading as follows. We initially attempt a cascade of two mixes, where each mix is equivalent to $1:\sqrt{R+1}-1$, and the amount discarded at the intermediate node is $\sqrt{R+1}-1 / \sqrt{R+1}$. If one level of cascading is insufficient to eliminate extreme mix ratios, we iteratively deepen the cascading by using three mixes each equal to $1:\sqrt[3]{R+1}-1$ and so on, until a suitable non-extreme mix-ratio is achieved. Finally, note that cascading has the negative side-effects of increasing the demand on the PLoC's fluid-path resources (*e.g.,* functional units and reservoirs), and of increasing the number of uses of the fluid with the larger contribution in the original mix. (In the example in Figure 7, *C'* requires an additional mix and uses of *B* increase to two from one.) In extreme cases, the increase of fluid uses due to cascading may cause underflow, which may require static replication (Section 3.4.2) as a solution as shown in Figure 6.

While Biostream [10] also relies on allowing excess production for their mix instructions, their approach is fundamentally different from ours in that they allow mixing only in a 1:1 ratio, and discard half of the output of the mix while we allow variable volume (and ratio) mixes. Because of their fixed-ratio mixing, achieving arbitrary mix ratios always requires cascading (except for 1:1 mixing), which executes on the slow fluid path, while our approach requires cascading only for uncommon cases of extreme mix ratios.

Finally, we note that though we allow cascading and excess production by default, there maybe cases where cascading mixes and producing/discarding excess fluid is disallowed because of safety, cost, regulation, or even correctness. For such fluids (identified by the programmer) we do not allow excess production.

3.4.2 Handling Numerous Uses — Static Replication

Because of the hardware limit of maximum capacity, numerous uses of a fluid (naturally or induced by cascading) can cause underflow even if we produce as much volume of the fluid without causing overflow. For such cases, where certain highly-used fluids must be produced in excess of hardware capacity, we employ static code replication to replicate part of the backward slice of the fluid's production. Various traditional compiler techniques for determining the backward slice of a variable exist [11].

In our context, static replication of the backward slice achieves pro-active generation of fluid in excess of what a single reservoir can hold by creating multiple instances of the same fluid. We replicate the numerous-usage node and distribute the original outbound uses as evenly as possible between the replicas. Once replication is complete, we re-run DAGSolve to produce new V_{norm} values for the new graph. If an underflow still occurs, we replicate another level in the DAG, effectively replicating the predecessor-nodes' predecessors. Replication continues in an iterative fashion until the underflow is eliminated. Because replication increases the demand on the PLoC's fluid-path resources (like cascading), the replicated code may exceed the PLoC's resources. In such cases, compilation fails. We follow this iterative procedure instead of one-shot replication of the entire backward slice because such one-shot replication may cause compilation failure even in cases where the iterative procedure succeeds. Finally, note that static replication is merely a graph transformation, and the LP formulation may be applied on the new DAG as well.

Because our replication is static, the additional demand for fluid-path resources is known *a priori* and accounted for at compile time. Thus, we can determine *statically* if the modified DAG fits within the PLoC's resources, as shown in Figure 6. In contrast, Biostream's reactive regeneration places a *run-time* demand for the resources that cannot be planned for and thus, may be hard to satisfy. Because cascading and static replication place extra demand on the resources, we fall back on these two schemes as a final option to avoiding underflow, as shown in Figure 6.

3.5 Statically-Unknown Case

Our volume management schemes so far assume that both the volumes output at every assay step and the number of uses are known statically. However, either of these quantities may be known only at run time, as discussed before. We extend DAGSolve to handle each such uncertainty — unknown volume first and unknown use next, and show how these alterations easily apply to LP.

To handle statically-unknown volumes, we delay the volume assignment step from compile time to run time while keeping V_{norm} calculation at compile time to reduce run-time overhead. To compute V_{norm}, we cut the outbound edges of the unknown-volume nodes. Consequently, these nodes become similar to final output nodes, while the sink of each cut edge becomes similar to an input node. While the natural input nodes are unconstrained in that the input volumes can be anything up to the default maximum, these artificial input nodes' volumes are constrained to be equal to the output of the unknown-volume instruction, which is measured at run time. The edge cutting may partition the DAG into several

partitions to each of which we apply DAGSolve. While we calculate the V_{norm} of the nodes and edges of all the partitions as before, there is one difference in the final step of absolute volume assignment (*i.e.,* application of the hardware constraints) due to constrained inputs. Assigning the default maximum to the node with the largest V_{norm} may require more than the available volume at a constrained input. To handle that case, we compute the minimum ratio of each input's V_{norm} and the available input volume for each constrained input once the volume is measured. Consequently, nodes and edges are assigned volumes by scaling their V_{norm} with this ratio.

While the above method handles unknown-volume nodes, such nodes may prevent applying DAGSolve to normal, known-volume instructions. Consider the output of a normal, known-volume instruction, node X in Figure 8(a). Node X has two uses, one of which transitively feeds an unknown-volume instruction (node U, shown as a hashed node). At the time of X's execution U's execution has not even begun, and as such, U's outbound edge volumes cannot yet be determined. However, an earlier use of X (the X-Y edge in Figure 8(a)) needs to be assigned some volume without waiting for the later unknown use. Thus, unknown-volume instructions prevent us from assigning volumes on the basis of transitive use. To handle this problem, we partition the DAG at compile time by cutting the outbound edges of nodes that transitively lead to an unknown-volume node and marking the edges as constrained inputs. As before, the known-volume instruction node is treated as if it were a final output node, and we compute the V_{norm} for the partitioned DAG. At run time, we divide conservatively the known-volume instruction output into N equal portions, if there are N uses of the output fluid (here we assume all the uses are known and handle unknown-uses next). We then apply the above method for constrained inputs to assign volumes. Figure 8(b) shows the DAG in Figure 8(a) after cutting Y-U and the outbound edges of X, with X', X'', and U' representing constrained inputs. X' and X'' each get half the volume of X and U' gets the run-time output of U. One slight refinement to the above conservative strategy is that if a partition gets more than one of the N uses, say m uses, then we can replace the m constrained inputs (of $1/N$ each) with a single constrained input of m/N.

Note that the above conservative strategy (including the refinement) with its potential for underflow is needed only for the instructions whose transitive use DAG includes unknown-volume instructions. All other instructions can be handled better via DAGSolve. If the programmer can provide hints on approximate output volume relative to input volume at the unknown-volume instruction, the portion per use can be adjusted accordingly instead of the default of equal portions across uses. In DAGSolve we model such a hint as a node whose output shrinks the input volume in the specified ratio. We also note that such a node is trivially compatible with the LP formulation mentioned before.
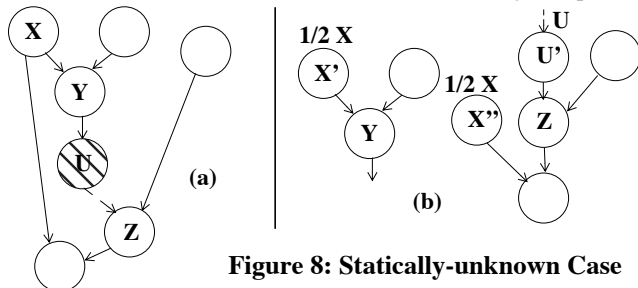
Next, we handle unknown uses, which occur due to control-flow in the assay (*if-then-else* and loop constructs). To handle *if-then-else*, we conservatively include both *if* and *else* paths in our DAG and apply DAGSolve (similar to standard dataflow analyses in modern compilers). Loops with statically-known number of iterations can be unrolled that many times and handled by DAGSolve. One may think that instead of unrolling, one could apply DAGSolve to the body of the loop and then scale the input volumes by the number of iterations. However, if the loop has loop-carried dependencies where fluids from different iterations are mixed in ratios, such simple scaling will not work.

For loops whose iterations are not known even at loop entry (*e.g., while* loops), there are two options: (1) The programmer provides a hint of the upper bound on the number of iterations for a loop. We simply unroll the loop that many times and apply DAGSolve. (2) While the programmer may not know the number of iterations, she may know the minimum volumes that the loop should output for successful completion of the assay. These minimum volumes can be used in the case of loops with independent iterations. For such loops, we make two changes to DAGSolve before applying it to the loop body. First, instead of assigning the largest V_{norm} to the default maximum, we pick the *output node* with the smallest V_{norm} and assign it the programmer-specified volume. All other nodes and edges are scaled as per the ratio of their V_{norm} and that of the chosen output node. This process gives us the volumes of the input fluids needed for one iteration to produce the specified output volumes in that iteration. The assumption here is that as much of the input fluids is produced as possible (over-provisioning may be achieved by static replication), and each iteration takes as much as needed from this initial volume. This strategy, however, is a departure from DAGSolve where intermediate nodes produced only as much fluid as needed. Accordingly, the second change is that we break up the assay at loops and treat nodes outputting fluids to loops as if they were final output nodes.

Finally, the per-iteration input volumes are valid only if the loop iterations are independent. In the presence of loop-carried dependencies, the specified output volumes would be reached after multiple iterations and as such, input volumes cannot be calculated from one iteration as done above. For such loops, we fall back on the first option of the programmer specifying an upper bound on the number of iterations. Note that programmer-provided hints or bounds, could be from any source including, but not limited to, human expertise, profiling runs and prediction.

While we have described our approach for handling the statically-unknown cases in DAGSolve, the same approach works if we need to use LP, by applying the same DAG transformations and adding the same constraints. Like DAGSolve, LP can also reduce its run-time overhead by solving all but the constrained inputs at compile time and then incrementally solving the constrained inputs at run time.

4 Results

We evaluate our volume management scheme in two parts: (1) avoiding underflow in RVol using DAGSolve on some real-world assays and (2) the error of our simple rounding. We also compare the execution times of DAGSolve and LP, and show the number of times regeneration is triggered assuming no volume management.

Figure 8: Statically-unknown Case

```
ASSAY glucose START

fluid Glucose, Reagent, Sample;
fluid a, b, c, d, e;
VAR Result[5];
a = MIX Glucose AND Reagent IN RATIOS 1 : 1 FOR 10;
SENSE OPTICAL it INTO Result[1];
b = MIX Glucose AND Reagent IN RATIOS 1 : 2 FOR 10;
SENSE OPTICAL it INTO Result[2];
c = MIX Glucose AND Reagent IN RATIOS 1 : 4 FOR 10;
SENSE OPTICAL it INTO Result[3];
d = MIX Glucose AND Reagent IN RATIOS 1 : 8 FOR 10;
SENSE OPTICAL it INTO Result[4];
e = MIX Sample AND Reagent IN RATIOS 1 : 1 FOR 10;
SENSE OPTICAL it INTO Result[5];
END                                                    (a)
```

```
glucose{                        move mixer1, s2, 4
input s1, ip1;Glucose           mix mixer1, 10
input s2, ip2;Reagent           move sensor2, mixer1
input s3, ip3;Sample            sense.OD sensor2, Result3
                                move mixer1, s1, 1
move mixer1, s1, 1              move mixer1, s2, 8
move mixer1, s2, 1              mix mixer1, 10
mix mixer1, 10                  move sensor2, mixer1
move sensor2, mixer1           sense.OD sensor2, Result4
sense.OD sensor2, Result1      move mixer1, s3, 1
move mixer1, s1, 1              move mixer1, s2, 1
move mixer1, s2, 2              mix mixer1, 10
mix mixer1, 10                  move sensor2, mixer1
move sensor2, mixer1           sense.OD sensor2, Result5
sense.OD sensor2, Result2      }
move mixer1, s1, 1                                      (b)
```

Figure 9: Glucose Assay

We first describe the assays and our experimental infrastructure, and then present our results.

4.1 Infrastructure and Benchmarks

We define a simple high-level language to specify the assays. Our syntax is similar to the specification format used in conventional assays. We use the names of common operations (*e.g.*, mix, separate, incubate) as key words which are accompanied by the relevant parameters such as operand fluids, temperature, and time duration of operation. Figure 9(a) through Figure 11(a) show three assays written in our language. The variable *it* corresponds to the output of the previous statement. We do not describe our language in any more detail due to space limitation. We construct a compiler to translate the high-level assays into the AquaCore Instruction Set (AIS) (Section 2.1). The usual steps of parsing, intermediate representation, register allocation, and code generation are similar to those of a conventional compiler. We show the compiler-generated AIS code in Figure 9(b) through Figure 11(b). Our compiler implements DAGSolve (the statically-known case) without cascading and replication which we perform manually. For LP, we use Matlab's *linprog* command, which is based on LIPSOL (Linear Interior Point Solver) [14].

Figure 9 shows the source and the AIS code for detecting the concentration level of glucose in a given sample using an optical sensor [9]. The assay uses a calibration step to obtain a best-fit curve of known amounts of dilutions of a standard glucose concentration. The sample's reading is placed on this curve to obtain the concentration. The numbers in the move and mix instructions are the relative volumes and number of seconds of mixing, respectively (as discussed in Section 2.1).

Figure 10 shows the source and AIS code for a glycomics assay (study of sugar molecules in proteins) [7]. The assay first concentrates the proteins containing glycans (sugar molecules) by an

```
ASSAY glycomics START

fluid buffer1a, buffer1b, buffer2; --buffer2 has PNGanF
fluid buffer3a, buffer3b, buffer4, buffer5;
fluid sample, lectin, C_18, NaOH;
fluid effluent,effluent2,effluent3, waste,waste2,waste3;

MIX buffer1a AND sample FOR 30;
SEPARATE it MATRIX lectin USING buffer1b FOR 30 INTO effluent
AND waste;
MIX effluent AND buffer2 FOR 30;
INCUBATE it AT 37 FOR 30;
MIX it AND buffer3a IN RATIOS 1:10 FOR 30;
LCSEPARATE it MATRIX C_18 USING buffer3b FOR 30 INTO
effluent2 AND waste2;
MIX effluent2 AND buffer4 AND NaOH IN RATIOS 1:100:1 FOR 30;
MIX it AND buffer3a FOR 30;
LCSEPARATE it MATRIX C_18 USING buffer3b FOR 2400 INTO
effluent3 AND waste3;                                   (a)
MIX effluent3 AND buffer5 FOR 30
END
```

```
glycomics{                      move heater1, mixer1
input s1, ip1 ;buffer1a         incubate heater1, 37, 30
input s2, ip2 ;sample           move mixer1, heater1, 1
input s3, ip3 ;lectin           move mixer1, s6, 10
input s4, ip4 ;buffer1b         mix mixer1, 30
input s5, ip5 ;buffer2          move separator2.matrix, s7
input s6, ip6 ;buffer3a         move separator2.pusher, s8
input s7, ip7 ;C_18             move separator2, mixer1
input s8, ip8 ;buffer3b         separate.LC separator2, 30
input s9, ip9 ;buffer4          move mixer1,separator2.out1,1
input s10, ip10 ;NaOH           move mixer1, s9, 100
input s11, ip11 ;buffer5        move mixer1, s10, 1
                                mix mixer1, 30
move mixer1, s1                 move mixer1, s6
move mixer1, s2                 mix mixer1, 30
mix mixer1, 30                  move separator2.matrix, s7
move separator1.matrix, s3      move separator2.pusher, s8
move separator1.pusher, s4      move separator2, mixer1
move separator1, mixer1         separate.LC separator2, 2400
separate.AF separator1, 30      move mixer1,separator2.out1
move mixer1,separator2.out1     move mixer1, s11
move mixer1, s5                 mix mixer1, 30
mix mixer1, 30                  }                       (b)
```

Figure 10: Glycomics Assay

affinity separation using lectin as the affinity matrix. Then, the glycans are cleaved from the proteins by mixing with a solution of PNGan F enzyme, and the extracted glycans are once more separated from the proteins using liquid chromatography (we add `separate.LC` to AIS for this purpose). To enhance future analysis and measurements, the effluent is permethylated using sodium hydroxide. Finally, the effluent undergoes chromatography, making it ready for external measurements such as mass spectrometry.

Figure 11 shows an assay to study the impact of inhibitors on enzyme kinetics [5]. Four different dilutions of enzyme, substrate and inhibitor are created. Then, all combinations of the dilutions are mixed and incubated and the optical density is sensed. The optical sensor gives an indication of enzyme activity at that specific dilution, for that specific substrate, in the presence of that specific inhibitor dilution. Because the number of loop iterations are known at compile time, we fully unroll the loops.

4.2 DAGSolve's Solution Quality

We evaluate DAGSolve on our benchmarks in terms of DAGSolve's goal of avoiding overflow and underflow. For all of our benchmarks, we assume a default maximum of 100 nl. Recent results on polydimethylsiloxane (PDMS) valves (a common type of valve used in microfluidic devices) show that a least count of 100 pl is feasible [12].

```
ASSAY enzyme_test START
VAR inhibitor_diluent, enzyme_diluent, substrate_diluent;
VAR i, j, k, temp, RESULT[4][4][4];
fluid Diluted_Inhibitor[4], Diluted_Enzyme[4];
fluid Diluted_Substrate[4];
fluid inhibitor, enzyme, diluent, substrate;

inhibitor_diluent = 1; enzyme_diluent = 1;
substrate_diluent = 1; temp = 1;
FOR i FROM 1 TO 4 START              --inhibitor
 Diluted_Inhibitor[i] = MIX inhibitor AND diluent
               IN RATIOS 1:inhibitor_diluent FOR 30;
 temp = temp * 10;
 inhibitor_diluent =  temp - 1;
ENDFOR
temp = 1;
FOR j FROM 1 TO 4 START              --enzyme
 Diluted_Enzyme[j] = MIX enzyme AND diluent
               IN RATIOS 1:enzyme_diluent FOR 30;
 temp = temp * 10;
 enzyme_diluent = temp - 1;
ENDFOR
temp = 1;
FOR k FROM 1 TO 4 START              --substrate
 Diluted_Substrate[k] = MIX substrate AND diluent
               IN RATIOS 1:substrate_diluent FOR 30;
 temp = temp * 10;
 substrate_diluent = temp - 1;
ENDFOR
FOR i FROM 1 TO 4 START              --inhibitor
 FOR j FROM 1 TO 4 START             --enzyme
  FOR k FROM 1 TO 4 START            --substrate
    MIX Diluted_Inhibitor[i] AND Diluted_Enzyme[j] AND
               Diluted_Substrate[k] FOR 60;
    INCUBATE it AT 37 FOR 300;
    SENSE OPTICAL it INTO RESULT[i][j][k];
  ENDFOR
 ENDFOR
ENDFOR
END
```

(a)

```
enzyme_test{                    dry-mov temp, 1
input s1, ip1;inhibitor         loop2: index k: 1->4
input s2, ip2;diluent            move mixer1, s6, 1
input s4, ip3;enzyme             move mixer1, s2, subs_dil
input s6, ip4;substrate          mix mixer1, 30
                                 dry-mov r0, temp
dry-mov inh_dil, 1               dry-mul r0, 10
dry-mov enzyme_dil, 1            dry-mov temp, r0
dry-mov subs_dil, 1             dry-sub r0, 1
dry-mov temp, 1                  dry-mov sub_dil,r0
loop0: index i: 1->4             move s7(k), mixer1
 move mixer1, s1, 1
 move mixer1, s2, inh_dil        loop3: index i:1->4
 mix mixer1, 30                  loop4: index j:1->4
 dry-mov r0, temp                loop5: index k:1->4
 dry-mul r0, 10                   move mixer1, s3(i)
 dry-mov temp, r0                 move mixer1, s5(j)
 dry-sub r0, 1                    move mixer1, s7(k)
 dry-mov inh_dil,r0              mix mixer1, 60
 move s3(i), mixer1              move heater1, mixer1
 dry-mov temp, 1                 incubate heater1, 37, 300
loop1: index j:1->4             move sensor2, heater1
 move mixer1, s4, 1              dry-mov t4, i
 move mixer1, s2, enzyme_dil    dry-mul t4, 16
 mix mixer1, 30                  dry-mov t5, j
 dry-mov r0, temp                dry-mul t5, 4
 dry-mul r0, 10                  dry-add t5, t4
 dry-mov temp, r0                dry-mov t6, k
 dry-sub r0, 1                   dry-add t6, t5
 dry-mov enzyme_dil, r0          sense.OD sensor2,RESULT(t6)
 move s5(j), mixer1             }
```

(b)

Figure 11: Enzyme Assay

The glucose assay (Figure 9) has five uses of the reagent, four of glucose and one of the sample. We show the DAG for this assay and the V_{norm} generated by DAGSolve in Figure 12(a) and the actual volumes in Figure 12(b). The smallest volume dispensed is
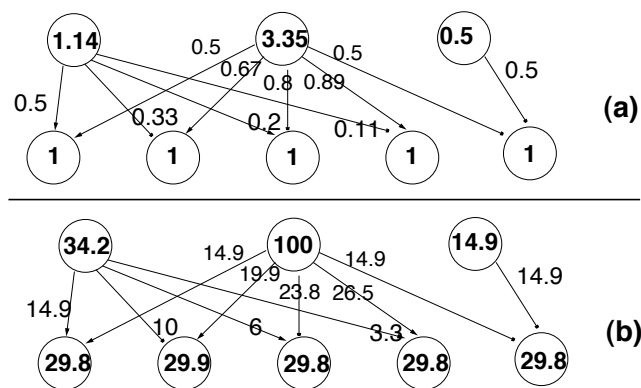

Figure 12: Glucose Assay DAG

3.3 nl which is well above the least count. Because all the volumes and fluid uses are statically known, not only the V_{norm} calculations but also the volume assignments occurs at compile time and thus, there is no run-time overhead for this assay.

The glycomics assay (Figure 10) consists of a sequence of mix and separate operations. Because the assay has three separations which produce statically-unknown volumes (Section 3.5), DAG-Solve's final volume assignment step is done at run time. The DAG is partitioned at the unknown-volume nodes resulting in four partitions, as shown in Figure 13. Buffer *3a* (in the second and third partitions) is used in two partitions and hence the corresponding input node is split into two constrained-input nodes each of which gets half the default maximum (*i.e.*, 50 nl). The three nodes *X1*, *X2*, and *X3* represent the constrained inputs corresponding to the unknown volumes generated by the separates. Because the separate's output volumes are unknown, we show only the V_{norm} and not the final volumes. While most of the V_{norm} are reasonably large numbers, the *X2* input to the third partition has a V_{norm} of 1/204 which may be a concern. If *X2*'s separation output volume in the second partition is high then there would be no underflow; otherwise, we would need more of *X2's* separation output, for which BioStream's regeneration would have to be triggered.

Due to the unknown-volume instructions, only the first partition's volumes may be computed at compile time, while the remaining partitions' volumes must be calculated at run-time. We show later that the run-time overhead of this computation is a few milliseconds on a 750-MHz processor, and is acceptable given that fluidic instructions take seconds to execute.

The enzyme assay (Figure 11), creates four different dilutions for each of the enzyme, substrate and inhibitor using the same diluent. Thus, the diluent is used twelve times. All combinations of the resultant dilutions are mixed so that each dilution is used sixteen times. Because the DAG for this assay is large and symmetric, we show only the enzyme-diluent part of the DAG in Figure 14(a). With the output V_{norm} set to 1, all the dilutions' V_{norm} equal 16/3 ~ 5.3 (the dilutions are the middle nodes in Figure 14(a) and each dilution is used in 16 mixes in the ratio of 1:1:1). The maximum V_{norm} is for the diluent fluid (V_{norm} = 54). DAGSolve results in 9.8 nl for all the dilutions (V_{norm} of 5.3 normalized to the maximum V_{norm} of 54 and scaled to 100 nl). Each dilution is split 16 times into 0.6 nl volumes to give the final volumes of 1.8 nl.

One of the dilutions corresponds to the 1:999 mix of the enzyme and diluent. As stated above, the dilution's volume is 9.8
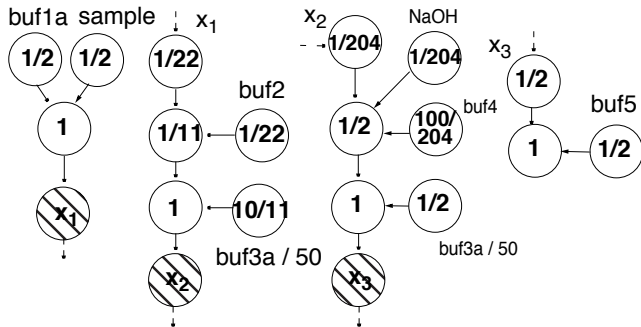
Figure 13: Glycomics assay DAG

nl and the enzyme input to the mix is 9.8 pl which is the minimum dispensing volume in this assay. While this dispense step underflows, all other steps dispense about 100 pl or more staying at or above the least count. The underflow is not surprising considering (1) the extreme mix ratio of 1:999 used in this assay, and (2) the many uses of the diluent fluid. In fact, we found that LP also fails to avoid this underflow. To address this underflow, we apply the optimizations of cascading and static replication (Section 3.4). As we discuss below, both optimizations are needed to avoid the underflow because both problems of extreme ratios and many uses are there.

We cascade each of the 1:999 mixes to three 1:9 mixes (shown in bold in Figure 14(b)). Each of the newly-created intermediate nodes is assigned a V_{norm} of 16/3, equal to that of the original extreme ratio node. For each cascaded mix, the diluent is used two more times (increasing the total number of uses from 12 to 18), and its V_{norm} increases to 81. The underflowing volume of 9.8 pl initially assigned to the 1:999 mix changes to 123 pl, corresponding to the volume assigned to the 1:9 mix for the first node in the cascade. Though the initial underflow is eliminated, there is a new underflow corresponding to the 1:99 mix ratio where the minimum

(a)

(b)

Figure 14: Enzyme Assay DAG

dispensed volume is 65.6 pl now. This underflow occurs due to the extreme mix ratio and due to the increased use of the diluent, and may be removed by increasing the diluent volume via static replication, or by cascading the 1:99 mix. We first examine static replication and replicate the diluent input node three times so that the problematic mix receives higher volume (similar to using three input instructions to three different reservoirs). Each replica is used in one set of dilutions — one for enzyme, one for substrate, and one for inhibitor. In this dilution, the diluent's V_{norm} reduces by a factor of 3 from 81 with no replication (Figure 14(a)) to 81/3 = 27 with replication (Figure 14(b)). Accordingly, the minimum dispensed volume increases by a factor of 3 from 65.5 pl to 196 pl, eliminating all underflow in the assay. Note that because the diluent's V_{norm} is the maximum, reduction in the V_{norm} results in increase in actual volumes due to the inverse relationship of volumes with the maximum V_{norm}.

We also tried the option of using replication without cascading which resulted in underflow with the minimum dispensed volume of 29.5 pl corresponding to the 1:999 mix ratio.

Recall from Section 3.3 that DAGSolve (and LP) provide a solution to RVol. To achieve our initial goal of solving the IVol problem, we round the results of the rational volume assignment to the closest integer multiple of the least-count. Such rounding did not cause any overflow/underflow for our assays. However, because such rounding can introduce errors in mix ratios, we evaluate its effect on our benchmarks using a maximum volume of 100 nl and least count of 0.1 nl. Averaged across the glucose and enzyme assays, the error was no more than 2%. (We do not include rounding error for the glycomics assay as the assay depends on statically unknown volume.)

4.3 Run Times

Table 2 shows the execution times of DAGSolve and LP, the number of constraints generated by LP, and the number of times regeneration is triggered assuming no volume management. Each execution time includes the processing time of generating the constraints from the internal DAG representation and for executing the algorithm (DAGSolve or LP). The execution times are from runs on a 750 MHz Intel PIII processor with 256 MB memory (each number is averaged over 10 runs to account for OS-related variations). For the glycomics assay, we show the total run time of all the four partitions of the DAG (resulting from the statically unknown separation steps in this assay).

For our assays, while DAGSolve is about 80 times faster than LP, even LP takes less than one second. This run-time is acceptable not only for compile-time solvable assays but also for run-time solutions like the glycomics assay which has separation steps with unknown volumes. Thus, both schemes incur little overhead compared to fluidic instruction execution times of several seconds.

To explore DAGSolve's complexity advantage over LP, we increase the enzyme assay's problem size. Instead of four dilutions, we perform ten dilutions for each of the enzyme, substrate and inhibitor. The results are shown under the assay name *Enzyme10* in Table 2. LP's run time increases to more than 20 minutes, while DAGSolve remains under 2 seconds, confirming that DAGSolve scales better than LP for large problem sizes. Though the LP solution for this example can be computed at compile time, such overhead when incurred at run time would be significant, even if fluidic instructions take several seconds to execute.
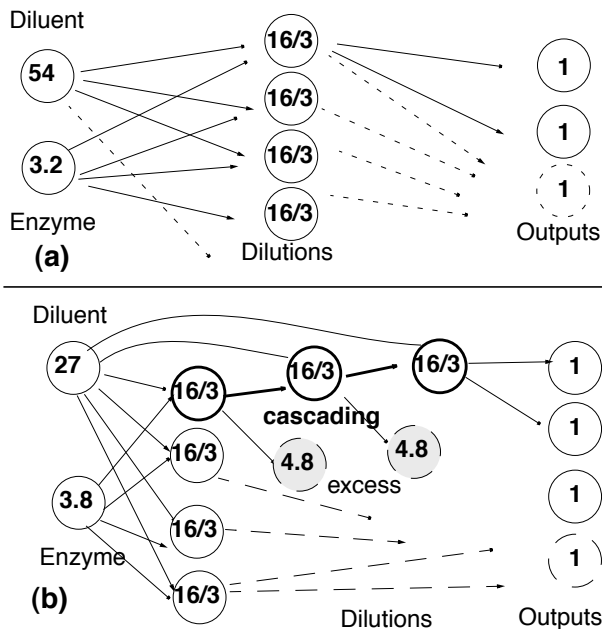
Table 2: DAGSolve, LP, and Regeneration

Assay	DAGSolve (s)	LP (s)	LP constraints	Regen. count
Glucose	~ 0	0.08	49	2
Glycomics	0.003	0.28	84	--
Enzyme	0.016	0.73	872	85
Enzyme10	1.57	1211	11258	1313

To eliminate the possibility that DAGSolve's speedups were solely due to the additional constraints, we evaluate the effects of adding DAGSolve's additional constraints (flow conservation and output equalization) to the LP formulation. Though the additional constraints result in some improvement in LP's run time, even in the best case (glucose assay), LP remained significantly slower than DAGSolve with a minimum slowdown of 60x (as compared to 80x without the additional constraints).

Without volume management, regeneration is required twice for the simple glucose assay, and 85 times for the enzyme assay. With DAGSolve, there are no regenerations. The number of regenerations for the glycomics assay is not shown because the number depends on the volume output at each separation. However, if the separation steps in glycomics produce low volumes, then regeneration may be required even with DAGSolve. Overall, our pro-active approach reduces the need for regeneration which executes on the slow fluidic path, and may require extra fluidic resources. Thus, our approach achieves low run-time overhead and low resource requirement for volume management.

Finally, we compare LP and ILP. For ILP, we used the integer mode for the Matlab extension of LP Solve 5.5 [1]. Though the ILP solver achieved similar execution times as the LP solver for the glucose assay, the ILP solver ran for hours without generating a solution for the enzyme assay, whereas the LP solver completed in 0.73 seconds.

5 Conclusions

We identified and addressed a practical issue, namely, fluid volume management, in programmable labs-on-a-chip (PLoC). Volume management addresses the problem that the use of a fluid depletes it and unless the given volume of a fluid is distributed carefully among all its uses, execution may run out of the fluid before all its uses are complete. Additionally, fluid volumes should not overflow (*i.e.*, exceed hardware capacity) or underflow (*i.e.*, fall below hardware resolution). We showed that the problem can be formulated as a linear programming problem (LP). Because LP's complexity and slow execution times in practice may be a concern, we proposed another approach, called DAGSolve, which over-constrains the problem to achieve linear complexity while maintaining good solution quality. We also proposed two optimizations, called cascading and static replication, to handle cases involving extreme mix ratios and numerous fluid uses which may defeat both LP and DAGSolve. We showed that our techniques produce good solutions for some real-world assays while being faster than LP.

Our techniques relieve the PLoC programmer from not only assay-specific details of mix ratios and number of uses but also low-level hardware details of maximum capacity and resolution. Our goal is to raise the level of abstraction in lab-on-a-chip technology as modern programming languages and compilers have done for computer technology.

References

[1] MILP LP_Solve 5.5, http://sourceforge.net/project/show-files.php?group_id=145213.

[2] A. M. Amin, M. Thottethodi, T. N. Vijaykumar, S. Wereley, and S. C. Jacobson. AquaCore: A Programmable Architecture for Microfluidics. In *Proceedings of the 34th ISCA*, pages 254–265, 2007.

[3] R. Go' mez, R. Bashir, A. Sarikaya, M. Ladisch, J. Sturgis, J. Robinson, T. Geng, A. Bhunia, H. Apple, and S. Wereley. Microfluidic Biochip for Impedance Spectroscopy of Biological Species. *Biomedical Microdevices*, 3(3):201–209, 2001.

[4] C. C. Gonzaga. An algorithm for solving linear programming programs in O(n3L) operations. In *Progress in Mathematical Programming Interior-point and related methods*. Springer-Verlag New York, Inc., 1988.

[5] A. Hadd, D. Raymond, J. Halliwell, S. Jacobson, and J. Ramsey. Microchip Device for Performing Enzyme Assays. *Analytical Chemistry*, 69(17):3407–3412, 1997.

[6] J. Janssen and H. Corporaal. Partitioned register file for TTAs. In *Proceedings of the 28th MICRO*, pages 303–312, 1995.

[7] Y. Mechref and M. Novotny. Structural Investigations of Glycoconjugates at High Sensitivity. *Chemical Reviews*, 102(2):321–370, 2002.

[8] N. Nguyen and S. Wereley. *Fundamentals and Applications of Microfluidics*. Artech House, 2002.

[9] V. Srinivasan, V. Pamula, M. Pollack, and R. Fair. A digital microfluidic biosensor for multianalyte detection. In *Proceedings of the 16th Annual IEEE International Conference on Micro Electro Mechanical Systems*, pages 327–330, 2003.

[10] W. Thies, J. P. Urbanski, T. Thorsen, and S. Amarasinghe. Abstraction layers for scalable microfluidic biocomputing. *Natural Computing*, 2007.

[11] F. Tip. A survey of program slicing techniques. *Journal of programming languages*, 3:121–189, 1995.

[12] M. A. Unger, H.-P. Chou, T. Thorsen, A. Scherer, and S. R. Quake. Monolithic Microfabricated Valves and Pumps by Multilayer Soft Lithography. *Science*, 288(5463):113–116, 2000.

[13] J. P. Urbanski, W. Thies, C. Rhodes, S. Amarasinghe, and T. Thorsen. Digital microfluidics using soft lithography. *Lab on a Chip*, 6(1):96–104, Jan 2006.

[14] Y. Zhang. Solving large–scale linear programs by interior–point methods under the MATLAB environment. Technical Report 96–01, Baltimore, MD 21228–5398, USA, 1996.