Puddle: A Dynamic, Error-Correcting, Full-Stack Microfluidics Platform

Max Willsey
University of Washington

Ashley P. Stephenson
University of Washington

Chris Takahashi
University of Washington

Pranav Vaid
University of Washington

Bichlien H. Nguyen
Microsoft

Michal Piszczek
University of Washington

Christine Betts
University of Washington

Sharon Newman
Stanford University

Sarang Joshi
University of Washington

Karin Strauss
Microsoft

Luis Ceze
University of Washington

Abstract

Microfluidic devices promise to automate wetlab procedures by manipulating small chemical or biological samples. This technology comes in many varieties, all of which aim to save time, labor, and supplies by performing lab protocol steps typically done by a technician. However, existing microfluidic platforms remain some combination of inflexible, error-prone, prohibitively expensive, and difficult to program.

We address these concerns with a full-stack digital microfluidic automation platform. Our main contribution is a runtime system that provides a high-level API for microfluidic manipulations. It manages fluidic resources dynamically, allowing programmers to freely mix regular computation with microfluidics, which results in more expressive programs than previous work. It also provides real-time error correction through a computer vision system, allowing robust execution on cheaper microfluidic hardware. We implement our stack on top of a low-cost droplet microfluidic device that we have developed.

We evaluate our system with the fully-automated execution of polymerase chain reaction (PCR) and a DNA sequencing preparation protocol. These protocols demonstrate high-level programs that combine computational and fluidic operations such as input/output of reagents, heating of samples, and data analysis. We also evaluate the impact of automatic error correction on our system's reliability.

1 Introduction

Microfluidic technology facilitates the automation of chemical and biological protocols. These devices manipulate small quantities of liquid at smaller scales and with higher precision than humans. Laboratories can use these devices to save time, labor, and supplies. Outside of the lab, microfluidic automation also promises to advance fields like medicine, education, and molecular computation/storage.

Despite these promises, microfluidics are still not universal. Different microfluidic technologies (covered in Section 2) have different drawbacks including high cost, inflexibility, high error rate, and difficulty to users. Until a system stack addresses these concerns, general-purpose microfluidic automation will remain inaccessible to all but experts in resource-rich labs.

Aside from the above concerns, most existing work on microfluidics has focused on automating individual protocols, i.e. given a fixed set of inputs, manipulate them in some way to produce an output. While this is certainly an important component, we picture a greater role for microfluidics. These devices are the bridge between computation and the world of chemistry and biology. However, the role of computation in these systems has typically been limited to that of a microcontroller or synthesis tool.

If we instead view microfluidic devices as part of a heterogeneous computer system, we can begin to close loops that otherwise require human intervention. Instead of executing a pre-determined list of operations and reporting the output, the heterogeneous system can use information from

sensors to dynamically make decisions. This combination of fluidic manipulation and computation is critical for emerging applications which depend on biology "in the loop", e.g., molecular data storage and computation [10, 17, 38, 45, 52] or automated experimentation [33–35, 48, 51].

We present a full-stack, open-source[1] microfluidics system with a new high-level programming model that allows unrestricted combination of computation and fluidics. Instead of a new programming language, we introduce Puddle, a runtime system that provides microfluidic manipulations through an API. Puddle controls PurpleDrop, an affordable general-purpose microfluidic device with novel capabilities that executes the fluidic portions of programs. Users write programs against the Puddle API using Python (or the language of their choice), and Puddle dynamically manages the fluidic resources and provides transparent error correction using a computer vision system.

Puddle's key innovation is its dynamic approach to resource (i.e., fluidic) management, a fundamental problem in microfluidic programming. Existing work takes a more static approach, trading off programming expressiveness for the ability to statically plan microfluidic execution. Solutions compete on metrics such as synthesis time, placement and routing efficiency, and simulation ticks to completion. This static approach comes at the cost of excluding or restricting programming features such as data structures, loops, functions. Puddle comes from the other side of the design space; we maximize expressiveness and ease-of-use while trading off some efficiency and ahead-of-time guarantees.

Our dynamic approach is more flexible than previous work, allowing both the system and the user to gather and react to data from the fluidic domain. These data-driven decisions occur at three levels (see Figure 1):

- *Execution-level decisions* ensure that the program is run as intended, e.g., error detection and correction.
- *Protocol-level decisions* allow protocols to conditionally take action, e.g., replenishing a liquid that may have evaporated
- *Application-level decisions* allow high level user code to make decisions based on protocol output, e.g., deciding what experiment to run next based on data analysis.

We demonstrate all three types of decisions in our evaluation.

Puddle and PurpleDrop constitute a complete system stack for microfluidic programming. While the design of the runtime system is our main contribution, the full-stack nature of the work necessitates advances at all levels:

- We present Puddle, a runtime system that provides a high-level microfluidic API. Puddle allows for more expressive programs than previous work, including the unrestricted combination of fluidic manipulation and computation.
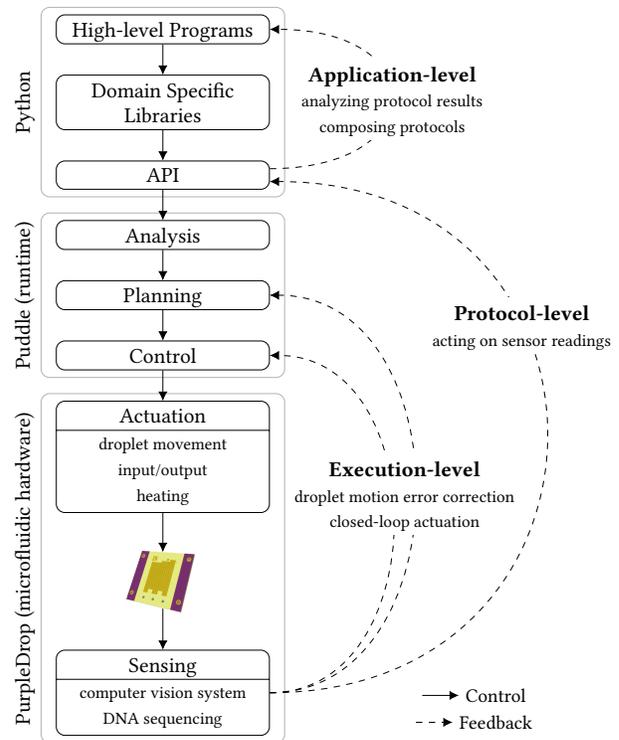
Figure 1. Puddle and PurpleDrop provide a full stack programming system for digital microfluidics. Users program in Python where they can combine Puddle's primitives into higher-level, domain specific operations. Programs go through the syscall-like API, allowing the Puddle system to plan, optimize, and control execution on the microfluidic device. PurpleDrop provides a low-cost target device. Feedback gives the system and the user flexibility at the execution, protocol, and application levels.

- We present PurpleDrop, a simple and affordable digital microfluidic device. Together with peripherals for heating, fluidic input/output, and volume measurement, PurpleDrop more is capable than comparable designs from the literature.
- We implement and evaluate a computer vision system for error-correction that is more flexible than previous work.
- We demonstrate fully automated, closed-loop execution of two important protocols in synthetic biology. These include protocol-level conditional action and application-level decisions.

Section 2 covers the relevant background on microfluidic hardware and software. Section 3 introduces the Puddle API with an example. Section 4 describes the implementation and our custom microfluidic device. In Section 5, we evaluate the error correction component of our system in isolation, and we demonstrate the end-to-end system with the first fully-automated execution of two synthetic biology protocols on a

---

digital microfluidic device. Section 6 discusses related work. Section 7 concludes with a discussion of the benefits and trade-offs of our dynamic approach and finally a mention of future work.

2 Background

2.1 Applications for Microfluidics

Microfluidic devices have broad applications, from medical devices [14] to tools for education or entertainment [2, 57]. Instead of breadth of applications, we want to highlight the different levels of complexity at which microfluidics can play a role, and how that role impacts the requirements of a microfluidic system.
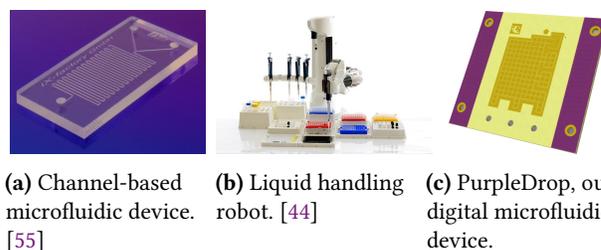
Protocol automation As mentioned in the introduction, the primary (and most obvious) use for microfluidic devices is to automate well-specified chemical or biological protocols. These protocols are typically specified in natural language, but they are precise enough to be executed by technicians other than the authors [31]. In fact, many of these protocols are distributed with the supplies (e.g., here are the steps to prepare sample X for process Y).

These encapsulated, well-specified protocols are ripe for expression as programs. Others have noted that automation offers not only convenience and savings, but it can prevent human error stemming from lack of precision and natural language ambiguity [47]. Most existing work on microfluidic software focuses on this level of application complexity (see Section 2.3).

Automated experimentation Performing individual protocols on a microfluidic device saves a lab technician time and effort. However, just like in computer systems, many tasks in the lab involve many repetitive actions coordinated by data-driven decisions. These situations call for a programming model that can compose fluidic protocols with general-purpose computation.

The field of automated experimentation has aspired to produce systems that can propose hypotheses, design and run experiments, and analyze data [33, 34, 48, 51]. These efforts rely on liquid handling robots and have therefore focused on drug discovery, a domain with the resources to afford high-throughput automation. While microfluidics could be a promising replacement for the execution layer, the programming model must be able to handle the computation side. These systems rely on a broad variety of techniques from logic programming [33] to program synthesis [35] to design experiments. Current microfluidic programming systems (covered in Section 2.3) lack the flexibility to combine fluidic control with such broad computational needs.

Molecular data storage and computing The field of computer science has taken notice of synthetic biology as a future substrate for data storage and computation. By interpreting molecules as data, molecular operations promise

(a) Channel-based microfluidic device. [55]

(b) Liquid handling robot. [44]

(c) PurpleDrop, our digital microfluidic device.

Figure 2. Microfluidic technologies come in many shapes and sizes, each offering different advantages.

massive storage density [12, 17, 38] or parallel computation [45, 52, 63].

Integrating these molecular components into heterogeneous computer systems requires microfluidics. The full life cycle of the data, including encoding, writing (into molecules), operating, reading back, and decoding must be fully automated.

2.2 Microfluidic Hardware

Microfluidic technologies all center around the manipulation of small fluid volumes, but different approaches offer different trade-offs between cost, flexibility, and reliability.

On one end of the spectrum, channel-based devices (Figure 2a) offer high precision and low cost at scale. These devices move liquids through a fixed set of channels, so they are single-purpose by nature. Similar to ASICs in computer systems, channel-based devices are used in situations where the application is static enough to overcome the initial design and manufacturing cost. Some channel-based devices incorporate configurable valves for some degree of flexibility [5, 42].

Liquid handling robots (Figure 2b) are more general, aiming to emulate a lab technician anthropomorphically with robotic arms controlling pipettes. In theory, these systems can be programmed to do anything a human can, but their flexibility comes at a size and monetary cost (thousands up to hundreds of thousands of dollars).

Digital microfluidic (DMF) technology (Figure 2c) offers flexibility at small size and potentially at low cost. DMF devices manipulate individual droplets of liquids on a grid of electrodes, taking advantage of a phenomenon called *electrowetting on dielectric* [43]. Activating electrodes in certain patterns can move, mix, or split droplets anywhere on the chip. Figure 3 shows how our DMF device, PurpleDrop, moves droplets by activating electrodes in sequence. The droplets can move through either an oil or air medium.

The discrete nature of handling individual droplets makes DMF devices more flexible than channel-based devices; think of a CPU compared to fixed-function ASICs. Unfortunately, the hardware itself can suffer from high failure rates [11]. The physics of electrowetting rely on a hydrophobic surface that can wear out, and it also involves high voltage electronic
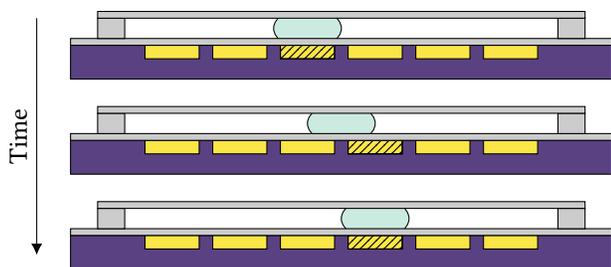
Figure 3. Side view of the PurpleDrop DMF device. Electrodes (yellow) sit at the top of the PCB (purple) in a grid. The PCB is topped with a dielectric and hydrophobic layer. The top plate sits on risers and is also hydrophobic and conductive. Droplets are attracted to the activated electrodes (shown with diagonal lines). Activating the neighboring electrode will move a droplet.

components, which can be prone to shorts or contact issues, and also cause dielectric layer breakdown. Either of these complications can result in partial system failures where regions of the board become inaccessible or unusable.

2.3 Controlling DMF devices

The "machine language" that controls DMF devices is little more than turning individual electrodes on or off. To move a droplet from one location to another, a controller must activate the electrodes along that path in sequence. This sequence of electrode actuations explicitly refers to locations on the board, and there is no notion of the identity, properties, or even the existence of the droplets. A better programming model allows high-level operations such as those shown in Figure 4, and the tool plans where those operations should occur and how the droplet operands get there.

Existing work has tackled these problems with place-and-route techniques from VLSI [8, 19, 20, 27, 32, 46, 61, 64]. The input to these tools is a directed acyclic graph that encodes the data dependencies of the operations. Figure 4 shows a pseudo-code snippet along with the corresponding DAG. Tools can take such a DAG and, together with the layout of the DMF chip, automatically determine when and where to execute each operation. The tool then plans routes for every droplet so that they do not collide on the way. Importantly, this is all done *ahead of time*: the tool statically determines if executing a given DAG on a given chip is possible. However, DAGs do not naturally express constructs found in programming like conditionals, loops, or functions.

Other approaches have proposed domain specific languages (DSLs) to increase expressiveness [6, 15, 20, 21, 39]. A DSL can support features like data-dependent control flow, which is necessary if a fluidic program is expected to act on the value of a sensor reading. However, compiling or interpreting microfluidic programs in these languages introduces a fundamental trade-off between the flexibility of


```
a = input(substance_A)
b = input(substance_B)
ab = mix(a, b)

heat(ab)
```
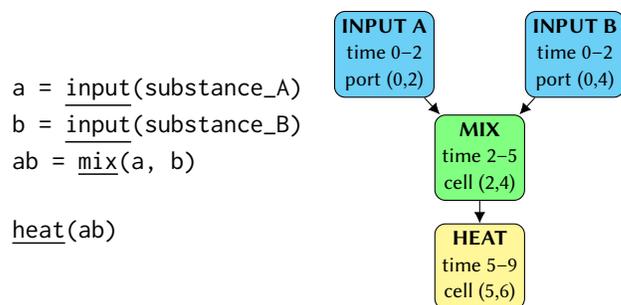
Figure 4. Pseudocode for fluidic a program fragment and the corresponding DAG where the operations have been scheduled and placed.

available programming constructs and the ability to statically reason about the program. In order to statically place and route a protocol, a system must be able to prove the protocol only uses resources that the hardware can supply. To meet this requirement, existing work has limited or eliminated programming features that allow potentially unbounded resource usage (e.g., loops, data structures, recursive functions, and error detection and recovery).

3 Dynamic Microfluidic Programming

The vision put forward in this paper calls for a microfluidics platform that can combine computation and fluidic manipulation in an unrestricted, high-level programming model. Our microfluidic programming system, Puddle, realizes this vision by making a different trade-off than previous work in microfluidics and doing everything dynamically. This decision is based on the following key insight: when it comes to resource management, expressiveness comes from dynamism. As a result, Puddle is not a programming language but a runtime system that provides a high-level API for microfluidic manipulations. The runtime system dynamically manages the fluidic resources (droplets), imposing no restrictions on the user's programming model.

This section serves as an introduction to the Puddle API, detailing what it provides to the programmer and the runtime system implementer. Section 4 explains the implementation, and Section 7 discusses the benefits and trade-offs of our approach.

3.1 Example

While the Puddle API is language-agnostic (we provide frontends in both Rust and Python), we focus on the Python frontend for this paper. Python immediately satisfies many of the requirements for an effective microfluidic programming solution. Python is popular with beginning programmers [22], and it is also the language of choice for many scientists due to the wealth of libraries for scientific computation [30] and bioinformatics [13]. Python also has a read-eval-print-loop

```python
1   # reduce is a functional fold over lists
2   from functools import reduce
3
4   def mix_n_heat(droplets):
5       vol_before = sum(
6           d.volume for d in droplets
7       )
8       stuff = reduce(mix, droplets)
9       stuff.heat(temp=90, seconds=60)
10      if stuff.volume < vol_before:
11          print("we lost some volume!")
12      return stuff
13
14  a = input("substance a", volume=1.0)
15  b = input("substance b", volume=2.0)
16  c = input("substance c", volume=1.5)
17  abc = mix_n_heat([a, b, c])
```

Figure 5. A example Python program interfacing with Puddle. Puddle API calls are underlined.

(REPL), allowing users to interactively write (microfluidic or conventional) programs.

Consider the program snippet in Figure 5. Users write regular Python programs that interface with Puddle through a simple library that implements the Puddle API (fully described in Section 3.2). Python snippets shown throughout the paper will have calls into Puddle underlined.

Starting at line 14, the user inputs three samples of various volumes. These inputs take the substance name and requested volume of the sample. Larger droplets may span multiple electrodes, but Puddle handles that automatically. The inputs return opaque handles to the new droplets called *droplet id*s. The user then calls mix_n_heat with a list containing the three droplet ids as an argument. Note that mix_n_heat is regular Python function and thus is completely transparent to the Puddle system; it could be recursive, exported or imported from a library, or passed around as a first-class value.

Inside mix_n_heat, the user calculates the expected volume of mixing all the droplets together by adding their individual volumes. The droplets are then all mixed together by reducing the list with mix, resulting in a new droplet id stuff. The call to heat then mutates stuff. The user then compares the droplet's new volume (which may have shrank due to evaporation) to the expected one, conditionally prints something, and then returns a handle to the mixed, heated droplet.

The API calls to input, mix, and heat were *non-blocking*. They immediately returned new droplet ids without actually manifesting the droplets that those ids represent, which

Fluidic I/O
input(name, volume) $\rightarrow d$
output(name, d)

Sensing
volume(d) $\rightarrow$ volume of d
temperature(d) $\rightarrow$ temp. of d

Fluidic Manipulation
mix(d_1, d_2) $\rightarrow d$
split(d) $\rightarrow (d_1, d_2)$
heat(d, temp, time) $\rightarrow d'$

Other
flush($d_1, d_2, ...$)

Figure 6. The Puddle API. ds are droplet ids, opaque handles to droplets. All calls are non-blocking except for those under Sensing.

might take several seconds. All manipulations and actuations are non-blocking in Puddle, because the user has no way to inspect their progress except through other API calls. Non-blocking calls give the runtime system more flexibility in how to implement operations on the microfluidic device. For example, even though mix is binary in the API, the runtime system can see that the user intends to mix many droplets together, which could lead to more efficient execution.

In contrast, the accesses to volume on lines 5 and 10 are blocking; they are also the only points at which something happens on the microfluidic device. The volume of a droplet is a dynamic property: it cannot (in general) be known statically, as precision errors in input and actuations like heat might change it. These calls must block and wait for the system to produce the relevant droplets and take the sensor reading, because the return value is just a number that the user's program (which Puddle knows nothing about) can manipulate and branch on.

3.2 Programming Interface

The API's most important feature is that it deals in opaque handles to droplets called *droplet ids*. The user cannot introspect on these ids (they are just numbers), so all queries and manipulations of droplets must go through the API. Therefore, Puddle is free to reorder, optimize, or delay performing the requested operations, allowing many calls in the API to be non-blocking. This opacity also allows Puddle to provide automatic error correction and process-like isolation for concurrency.

The complete Puddle API is listed in Figure 6. The calls for fluidic I/O and manipulation are non-blocking; they immediately return a fresh droplet id. The fluidic I/O calls are indexed by a name, which refers to an input pump based on a configuration file with the hardware details. The fluidic manipulation functions are self-explanatory. Note that they are functional, consuming their droplet id arguments and returning new ones. The frontend, however, is free to wrap the API calls to provide an idiomatic interface: for example, heat was used imperatively in Figure 5.

The sensing API calls force the system to "flush the queue" by performing the operations necessary to make the input droplet. The `volume` operation reads the volume using the camera (see Section 4.3). The `temperature` operation measures the temperature using an onboard sensor. In our Python frontend, `volume` and `temperature` are getters.

The `flush` operation allows the user to manually force to realize the specified droplets (or all of them, if none are given), which is useful in interactive programming. This can also be useful to inform Puddle of parallelism it would otherwise not see, as seen at the end of Section 7.2.

Users can easily extend the Puddle API with their own actuation and sensing primitives. The mechanism for allocating space on the device (Section 4.2.2) is sufficiently general to implement any primitive that the hardware might support.

3.3 Handling Errors

API calls can fail instantaneously for two reasons: invalid arguments or using a consumed droplet id. These failures happen as soon as the API call is made and are recoverable; the error propagates back to the user (in the form of an exception in the Python frontend). Consider the following code snippets:

```
input("water", 1e-10)
```
```
ab = mix(a, b)
mix(ab, a)
```

The left snippet demonstrates an invalid argument by trying to `input` too small a volume of fluid; the pumps do not have that level of precision. The right snippet reuses droplet id a after it has been consumed on the first line. Droplets are physical resources that can only be consumed once. All API calls under "I/O" and "manipulation" consume their droplet arguments. It is the programmer's responsibility to not reuse droplets ids that have been consumed. An imperative interface (like `heat` in Figure 5) can help prevent this problem by changing the droplet id that the wrapper object refers to. The Rust frontend statically prevents droplet reuse through its ownership-based type system.

Additionally, hardware failures may occur during execution, making droplets not move or actuate as planned. Most of these are automatically detected and corrected by Puddle's error correction system (described in Section 4.3). In rare cases, however, an error can result in a situation that is unrecoverable (e.g. the number of failed electrodes prevents routing). Because actuation API calls are non-blocking, the program may have progressed with the assumption that the promised droplets will be actually produced. Therefore, Puddle treats this case as unrecoverable and throws an exception to the user.

4 Implementation

Our implementation spans three levels of the stack shown in Figure 1: a frontend that facilitates high-level microfluidic programming against the Puddle API, the Puddle runtime
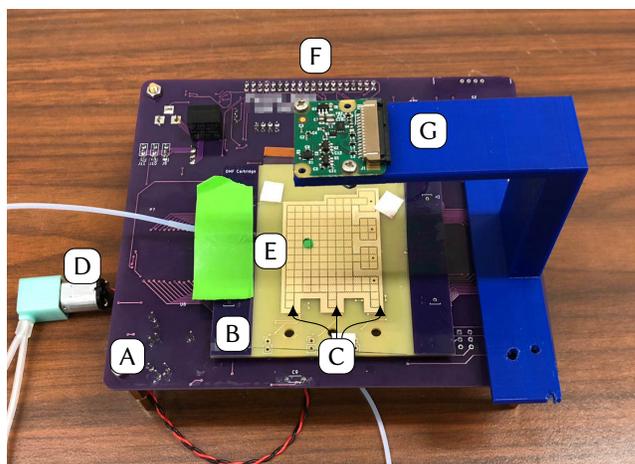
Figure 7. PurpleDrop, our digital microfluidic device. The motherboard PCB (A) contains the electronic components, and the daughterboard (B) contains the electrodes and the hydrophobic surface detailed in Figure 3. The device supports heaters on the bottom three electrodes (C). We can drive up to three pumps (D) that can input or output fluids on the edge of the device (E). PurpleDrop is controlled by a Raspberry Pi over the 40-pin connector (F). The Raspberry Pi connects to a camera on a 3D-printed mount (G).

system which implements the API detailed in Section 3.2, and PurpleDrop, the DMF device which Puddle controls. The interface and programming model were covered in the previous section; here we detail the implementation of the hardware and runtime system.

4.1 PurpleDrop DMF Device

We designed our digital microfluidic device, PurpleDrop, with simplicity and accessibility in mind. All together, the components cost on the order of $300, orders of magnitude less than most other microfluidic systems. Furthermore, the design uses commodity components and does not require a clean room, so anyone with electronics experience could assemble PurpleDrop on their own or have it assembled by a PCB assembly service. Figure 7 shows the device and enumerates its components.

DMF hardware The DMF portion of PurpleDrop is responsible for holding and manipulating the droplets. The daughterboard contains the electrodes that hold the droplets, and the motherboard contains the electrical components such as high-voltage controllers, shift registers, etc. Both mother and daughterboard are PCBs. The daughterboard is removable, allowing different configurations of electrodes with the same motherboard. The modularity is also useful for durability, as the daughterboard is the most prone to wear.

PurpleDrop's design was inspired by OpenDrop [2], an existing open-source DMF platform. Unlike OpenDrop, PurpleDrop runs without an oil medium for easier setup. Figure 3 shows a side view of the PurpleDrop daughterboard.

Electronic control   Instead of using a microcontroller to drive the electronics needed for electrowetting, we use a Raspberry Pi 3B [1] single-board computer. The Raspberry Pi runs Linux and sports a quad-core 1.2 GHz ARMv7 processor as well as GPIO pins, allowing PurpleDrop to be a self-sufficient microfluidics platform. The control software actuates the electrodes on the daughterboard with the GPIO pins through some shift registers on the motherboard. The execution planning (Section 4.2) and computer vision for error correction (Section 4.3) also run on-device, so no host machine is needed.

Peripherals   Microfluidic devices can move and mix liquid samples, but possible applications are limited without the ability to sense and manipulate properties of the droplets and get droplets to and from the device. PurpleDrop includes a heater, temperature sensor, and the ability to do both input and output of droplets. Input and output are driven by small peristaltic pumps which carry droplets to/from test tube reservoirs or other devices. Section 5.4 demonstrates using fluidic IO to interface with a DNA sequencer.

PurpleDrop also uses a camera mounted on top of the device as a multi-purpose sensor. The camera detects the locations of the droplets for error correction. Other approaches like capacitive sensing [7] would increase the hardware cost, where as the camera is relatively cheap. The camera can also sense the volume of droplets, which is useful for certain protocols like that shown in Section 5.3. Droplet volume is computed by multiplying the area of the droplet in the image by the fixed distance between the board and the top plate (see Figure 3).

4.2   Planning and Execution

Because all the complications of a programming language (loops, function calls, conditionals) are handled at the user level, the internals of Puddle are concerned only with the planning and execution of API calls. Note that we do not use any of the existing algorithms that might guarantee optimal routing or extract parallelism. This decision was made primarily to simplify implementation, but also because none of our microfluidic needs demand that level of efficiency. Section 7.3 discusses how we could incorporate this body of work into our system.

Figure 8 shows the entire lifetime of an API call. The first step is reification into a *command*, the object used internally to represent a request from the user. The remainder of the flow operates on these command objects. All types of commands (input/output, sensing, actuation) go through the same flow, so a user can extend Puddle with a new primitive
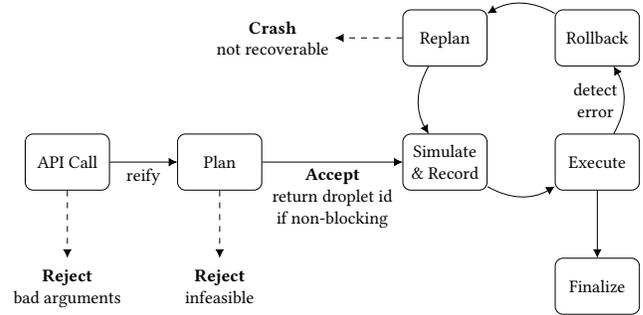

Figure 8. The life of a command. Dashed edges end the flow immediately. Commands may be rejected if the API call was malformed or a feasible execution plan cannot be found. The user can recover from rejected commands, but not from failures to replan accepted ones.

without modifying the planning and execution infrastructure.

4.2.1   API Calls to Commands

Commands store the operations' arguments, input droplet ids, and freshly created output droplet ids for droplet manipulation operations. These output droplet ids correspond to the droplets that command will make if successfully executed. After the command is created and planned, the system can return these ids if the API call was non-blocking, allowing the program to proceed without waiting on execution.

The core of planning is placement and routing, which occur after the API call has been reified into a command. Commands can form a DAG where the edges are their droplet id dependencies. For simplicity, however, we store them in a queue and place and route them serially.

4.2.2   Planning

Each command makes an *allocation request* for space on the microfluidic board. For example, mix requests a rectangle slightly larger than the resulting combined droplet so it has space to move the droplet in a circle, agitating the mixture. The allocation request can also place constraints on the features of the space, e.g. heat requests a space with a heater. Our simple placement algorithm directly checks for the best of all possible ways to satisfy a location. This is linear in the size of the hardware, and the hardware is small, so practically we can still do this in real-time.

The allocation request also specifies the (relative) desired locations of the input droplets. In the case of mixing two 1×1 droplets, mix will request a 3×2 rectangle and ask that the input droplets start at coordinates $(0, 0)$ and $(1, 0)$. The adjacent droplets will combine, leaving a 2×1 droplet inside the 3×2 rectangle. After the allocation request is placed, the system routes the input droplets to their specified locations using a modified A* algorithm [8, 24]. Both placement and

routing ensure that droplets stay at least 1 space apart to avoid collisions.

If either placement or routing fails, the command is rejected as infeasible. Because planning happens right after command creation, the user gets an immediate, recoverable error. If planning succeeds, then non-blocking API calls can return droplet id(s) knowing that their successful execution is at least feasible (although not guaranteed in the face of hardware errors).

4.2.3 Simulation and Recording

Regardless of whether the API call is blocking, successfully planned commands are then simulated. Each time step in the simulation is recorded, resulting in a *record* of where each droplet is (and thus which electrodes to activate) at every moment. The record provides a view into the future state of the microfluidic device assuming that no hardware errors occur during execution.

Puddle only simulates droplet movement to check for errors and determine the presence and location of droplets on the DMF device at each timestep. Chemical results of actuations on the droplets or mixing them are not simulated. Simulation does, however, record when actuations occur, so execution can perform them when it "replays" the record.

The record allows multiple commands to be planned ahead-of-time without executing any of them. Consider the following program snippet:

```
for i in range(1000000):
    input("water", volume=3.0)
```

This program clearly cannot run on our microfluidic hardware; it requires more space than any DMF device available today. And since Puddle knows nothing about the program structure, it will begin to accept these operations as they are submitted. However, commands are planned with respect to the latest state in the record, i.e. commands are planned under the assumption that all previous commands execute exactly as planned. Eventually, one of the inputs in the above program will be infeasible, since the DMF device will be full of the previous droplets. Puddle will reject the command as infeasible, returning an error to the user even though none of the commands have been executed yet.

4.2.4 Execution, Monitoring, and Rollback

The simulation record allows execution to be asynchronous with planning. Execution simply consists of popping the earliest state from the record, and activating the electrodes (and any peripherals) according to the droplets' position in that state. After a short delay, Puddle uses its computer vision system (described in Section 4.3) to detect the actual state of droplets on the device. If the actual state does not match the expected state, the system triggers a rollback. This "check and correct" flow is similar to previous work [26] that uses capacitance sensing instead of computer vision.
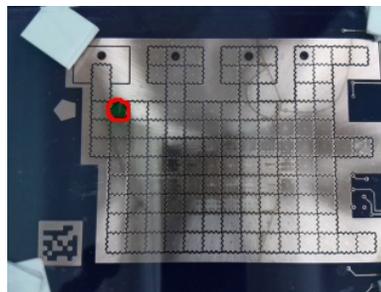
Figure 9. A computer vision system identifies droplets in real time for error detection and volume measurement.

A rollback consists of deleting the record and replanning all commands which have not been completed. Replanning is identical to planning, except that failure to *re*plan is unrecoverable (Figure 8). Non-blocking API calls may have already returned with a droplet id that essentially promises that new droplet. Since Puddle had no knowledge of the program, we have no choice but to terminate.

During the rollback, Puddle can also mark any electrodes that failed to move a droplet as dead. Otherwise, the rollback would replan the same route over the same electrode, and the error would occur again. Section 5.2 demonstrates how this allows execution on a DMF with faulty electrodes. The user can tune this behavior, forcing Puddle to retry an electrode a certain number of times before marking it as dead.

Replanning can fail in one of two cases: the number/identity of the droplets on the board has suddenly changed (something accidentally mixed), or the new arrangement of droplets is impossible to place and route. The first case is relatively unlikely, as hardware errors tend to be a failure to move a droplet (due to some defect with the DMF device) rather than an errant move in some other direction. The second case is also unlikely in the face of few errors: the main constraints to placement and routing are the number of droplets on the device, and the same droplets (albeit in a different arrangement) were successfully planned before. However, as errors accumulate, Puddle avoids more and more regions of the board. If enough errors build up, routing can become impossible, and replanning will fail.

The record also keeps track of the states that commands were completed in. When execution reaches such a state, the command is finalized. For most commands, this does nothing, but for blocking commands, this sends the result back to the user.

4.3 Error Detection via Computer Vision

The previous section discussed how Puddle corrects errors via the rollback and replan mechanism, but not how we detect them. Like other works, we use a computer vision system to localize the droplets on the DMF device. However, our system is more flexible. Past work has required either a

template image for a droplet [36] or a reference background image of the electrode array [50, 58]. The template image approach does not scale to droplets of different shapes and sizes, and the background difference approach is sensitive to changes in lighting and therefore requires a highly controlled environment.

We detect droplets based on color. We tint all the input fluids with green dye, and then calibrate the vision system to that hue. Any object within that hue range is recognized as a droplet. Because we are using the hue-saturation-value color space, our system is resistant to changes in lighting. We require only a simple paper shade to reduce glare off of the reflective surface of the chip.

The shape detection portion of error detection is implemented in OpenCV [9]. The result of the hue filtering is a binary image indicating which pixels are the desired shade of green. A series of morphological operations (erosions and dilations) suffices to remove any noise. A contour finding algorithm [53] then generates shapes that represent the droplets. We finally use a projective transform to map those shapes from the image's coordinate space to that of the DMF device.

Once the shapes are detected, Puddle must determine if the set of shapes constitutes an error. The first step is to match the shapes with the expected droplets. We use a distance metric to measure the difference in their locations and sizes. The pairwise distances form a bipartite graph, and we find a matching using the Kuhn-Munkres algorithm [37]. Each shape is then compared with its expected droplet. If they are all similar enough, execution proceeds. If there is a significant difference between the expectation and reality according to the camera, we convert the shapes into the new expected state and trigger a rollback, which replans unfinished commands starting from the new state.

The result is an error detection system that is more robust to lighting changes and handles droplets of any shape or size, an improvement over previous work.

5 Evaluation

We evaluate our system both quantitatively and qualitatively. First we evaluate the computer vision system in isolation, then we evaluate it in the context of error detection and correction. We then demonstrate Puddle's ability to write high-level programs and interface with other computer systems with two case studies.

5.1 Computer Vision

We evaluated our computer vision system on a dataset of 57 images taken from the Raspberry Pi camera at 320x240 pixels, the resolution we use for real time tracking. We manually labeled the droplets in the image and also specified which electrodes they occupied. The images cover a wide variety of droplet counts, droplet locations, and droplet areas

Image Segmentation Metrics

Mean Precision	0.9851
Mean Recall	0.9147
Mean False Positive Rate	0.0001

Droplet Metrics

Mean accuracy of droplet area	0.9220
Incorrect droplet count	1 / 57
Droplet occupation overestimates	14 / 56
Droplet occupation underestimates	0 / 56

Table 1. Results from evaluating our computer vision system on 57 manually labeled images of the DMF device. The first set of metrics come from image segmentation, and the second set is specific to droplet recognition. Droplet occupation is explained in Section 5.1.

on the microfluidic board, and are thus representative of the microfluidic setup, and the variability expected to arise within the setup during normal operation of the device. We used a single piece of white paper mounted above the device to reduce the glare off the PCB, otherwise the lighting was uncontrolled and varies throughout the images.

Table 1 shows the results of our evaluation. Since droplet recognition is similar to image segmentation, we first calculate some metrics from that domain. Notably, the precision is higher than recall, indicating that our recognition system is relatively conservative. This comes from the fact that our system uses more erosion than dilation to remove noise from the image. Erosion results in slightly smaller recognized areas in the image, whereas too much dilation would combine nearby droplets. The area underestimate is consistent and thus could be calibrated out.

We also compute a number of metrics specific to droplet recognition. Droplet count is the most important metric for error correction. If the system observes the wrong number of droplets, it cannot match the expected state to the actual state. We only saw one of these errors in the dataset: in an image with droplets on neighboring electrodes, the system confused two droplets as one. Puddle prevents droplets from being on neighboring electrodes to avoid accidental mixing, so this error is unlikely to occur in practice. Of the remaining 56 images, we calculated the electrodes the droplets occupied. Puddle takes a conservative approach here, preferring to turn on more electrodes rather than fewer to ensure droplet movement. The results reflect this with 14 overestimates but no underestimates. Overall, these results indicate that the vision system allows for correct and reliable droplet detection.

5.2 Error Correction

In Puddle, we use the droplet position and size information as part of a larger error correction system including droplet
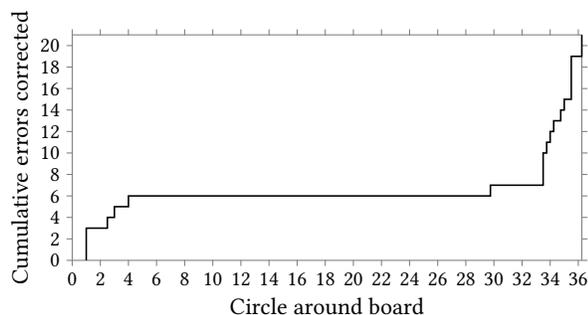
Figure 10. To test error correction, we moved a droplet in circles until failure. Total experiment time was 2 hours and 11 minutes. Along the way, the computer vision system detected and corrected errors, marking regions of the board as faulty and avoiding them in the future. The errors at the beginning correspond to faulty electrodes; those at the end were caused as the droplet evaporated.

matching, rollback, and replanning (detailed in Section 4.2). To evaluate these mechanisms and their impact on DMF reliability, we staged an endurance test on the microfluidic device.

DMFs can suffer from either inherent or use-induced failure. Flaws inherent to the device itself, e.g., surface flaws or poor electrical contact in the wiring for some electrodes, lead to failure early in execution. Use-induced defects, e.g., droplet evaporation or surface wear, lead to failure later in execution. Our endurance test demonstrates that Puddle's error correction can extend the life of a DMF device, allowing it to run longer protocols in the face of both types of failure. We specify four points on the chip near the corners and route a droplet between them over and over until the system eventually marks so many electrodes as faulty that routing fails.

Figure 10 shows the results of our endurance test. Note that six errors occur relatively soon in the test, before the completion of the fourth loop. Without error correction, a protocol would be forced to terminate here. These errors were due to poor electrical contact, resulting in a weaker electrowetting force that failed to pull the droplet to that electrode. Our error correction system identified these electrodes and avoided them in later loops. The later errors (starting around loop 34) were due to evaporation, leaving the droplet too small to move reliably. The next section demonstrates how automatic replenishment can deal with evaporation.

5.3 PCR and Thermocycling

Many chemical or biological protocols include thermocycling, or repeated heating and cooling, to speed up a reaction or denature a reagent. Thermocycling poses a challenge to current DMF systems that operate in air (as opposed to oil):

```python
1   min_volume = 10 * microliters
2
3   def thermocycle(droplet, temps_and_times):
4       for temp, time in temps_and_times:
5           heat(droplet, temp, time)
6           if droplet.volume < min_volume:
7               # '+=' is a mutating mix
8               droplet += input("water", min_volume)
9
10  def pcr(droplet, n_iter):
11      thermocycle(droplet, n_iter * [
12          (95, 3 * minutes),
13          (62, 30 * seconds),
14          (72, 20 * seconds),
15      ])
```

Figure 11. Python code for polymerase chain reaction (PCR) and thermocycle. The heating in thermocycling can evaporate droplets, so the code replenishes with water if necessary. Note that list multiplication in Python is concatenation, e.g. `2 * [1] == [1, 1]`

The heating portion of thermocycling could evaporate the small droplets being manipulated on the DMF device.

From a programming perspective, the natural way to express thermocycling is with a loop. Moreover, thermocycling is not a protocol in itself, but rather it is an important part of many other protocols. Ideally, we would write the code for thermocycling once, and its behavior would be parameterizable and reusable.

Figure 11 shows our implementation of thermocycling in Puddle. The use of functions, data structures, and data-dependent control-flow put this implementation out of reach for any other high-level microfluidic programming system that we know of.

We also implemented polymerase chain reaction (PCR) using `thermocycle` as subroutine. PCR is an important protocol in synthetic biology that selectively amplifies DNA in a solution. We validated the experiment by using the Qubit system [49] to quantify the amount of DNA before and after amplification. We performed 8 cycles of PCR which required 2 replenishments to avoid evaporation. The procedure doubled the amount of DNA in our 10 microliter sample. While commercial PCR instruments achieve more efficient amplification, our PCR protocol was successful and can be improved with more precise heaters and temperature sensors. To our knowledge, this is the first fully-automated execution of PCR with replenishment on a DMF device in air.

5.4 DNA Sequencing

To further highlight the combination of microfluidic manipulation and computation, we performed DNA sequencing using Puddle and the MinION sequencer [40]. The MinION

```python
1  def sequence(droplet):
2      # actual prep protocol
3      droplet += input("buffer", 65 * microliters)
4      output("sequencer", droplet)
5      # pseudocode to get and process the data
6      data = get_data_from_device()
7      seq = process(data)
8      return seq
```

Figure 12. A Python function that takes in a droplet and returns the sequence of DNA contained in that droplet.

requires some fluidic steps to prepare a sample for sequencing, and the raw output data requires processing to return the DNA sequence. Because of the computation needed for data processing, the sequencer is connected directly to a laptop instead of the Raspberry Pi. The Python code (including data processing) runs on the laptop, but the Puddle system still runs on the Raspberry Pi. The two sides communicate over the network using the same Puddle API, which is encoded in JSON format.

The code for the sequencing protocol is listed in Figure 12. `get_data_from_device` and `process_data` are pseudocode Python functions that get and process the data from the MinION; the former returns the raw data and the latter returns the actual DNA sequence. We use pseudocode because the actual process for sequencing and processing involves a closed-source GUI application. Given an API to perform these operations, automating the entire protocol including data acquisition and analysis is a matter of implementation. We validated the DNA sequencing results from the experiment by comparing the reported sequences with the known sequences used as input.

The protocol is simple, but it still demonstrates the power of the Puddle programming model. In one sense, the function `sequence` is no more exciting than one that reads the contents of a file from disk. However, existing microfluidic programming models cannot express both the fluidic and computational aspects of this protocol, nor can they export this functionality as an easy-to-use (and reuse) Python function. To our knowledge, this is the first time that computation and protocol execution are merged in this way. We envision encouraging the use microfluidics for complex experiment automation.

6 Related Work

DMF hardware DMF devices can be built on different substrates including silicon, or on a printed circuit board (PCB). While the former two technologies offer great control over the surface topology (an important property for electrowetting), PCBs are much more flexible and accessible, as they are cheaper and offer multi-layer wire routing [18]. We chose a

PCB substrate and operate in air because it makes the device easier to manufacture and use. Our error correction system compensates for faults which may occur due to the more affordable substrate.

Jebrail et al. have presented a system for replenishing liquid on a DMF device [29], but the intervention is not automated (a user manually pushed a syringe).

DMF routing Extensive work in this area has led to algorithms that can exploit parallelism [8, 19], reduce contamination [27, 61, 64], and minimize overall droplet travel [32, 46]. Our place and route algorithms in Puddle are intentionally very simple and do not provide these features, as these are outside the scope of this work. Section 7.3 discusses how such features could be implemented within our system.

Microfluidic Programming Previous work has improved upon static DAG place and route with domain specific languages for protocol specification and execution. Grissom et al. propose a dynamic interpretation approach that allows control flow [20], but the programming model is still centered around explicit DAG construction. Dynamic interpretation also suffers from sub-optimal place and route, as the system only considers one basic block at a time. Others have proposed static compilers capable of handling control flow, e.g., Curtis et al. [15] and Ott et al. [39]. These approaches offer a static guarantee that the program (ignoring errors) can be successfully routed, but they are all less expressive than Puddle. In particular, programs cannot (in general) use recursion, manipulate data structures, or allocate in loops. The example given in Figure 11 exhibits the latter two of these characteristics.

Previous approaches using general purpose languages either offer little abstraction, requiring the user to manually address droplet locations [2, 16], or require re-expression in a restricted embedded DSL or manual DAG construction [6, 15].

Amin et al. propose an instruction set for a channel-based microfluidic architecture [5]. Puddle could feasibly target channel-based devices through such an ISA.

Autoprotocol [56] and Aquarium [31] focus on off-premises execution where users send samples and protocols to be executed by pipetting robots (Autoprotocol) or by human technicians (Aquarium). Antha [54] targets various lab automation technologies for on-premises execution but not DMF. The intended use case for these platforms is high-throughput fluidics rather than writing and running expressive programs. In principle, Puddle could be used as a backend for these frameworks to allow execution on DMF devices like Purple-Drop.

Error Correction for DMFs Section 4.3 mentioned that previous work on optical error detection is not as flexible as our approach in terms of droplet size/shape [36] or lighting conditions [50, 58]. We do require that droplets are dyed green. We are working to lift this restriction, as it limits the

use of the camera as a colormetric sensor in the visible light spectrum. Other work has proposed impedence sensing as a means of detecting droplets [23].

Alistar et al. [3] present an ahead-of-time solution for operations that take variable time, accounting for some classes of precision errors. An online solution has also been presented [4], but it has an offline component and only works for programs represented as DAGs. Previous work has also proposed error correction for DMF devices [25, 26, 28, 59, 60], but these were implemented outside of a larger microfluidic programming solution, relied on more expensive sensors, or were only simulated.

Lazy dataflow processing Our approach of allowing the programmer to lazily manipulate opaque ids through an API bears a similarity to an approach used in data processing. In particular, Apache Spark [62] and PyTorch [41] provide a similar abstraction for the domains of data processing and machine learning. Both allow the user to build up sets of operations in a general-purpose host language, and then (upon request for data) a runtime system optimizes and executes the dataflow graph of pending operations.

7 Discussion

Our main contribution is the notion of a microfluidic runtime system that dynamically manages fluidic resources. This advances upon previous work by allowing unrestricted flexibility and expressiveness, at the cost of some static guarantees. Below we further detail the benefits of and reasons for this approach, discuss the drawbacks, and propose future work based on this key idea.

7.1 Benefits of Dynamism

Existing work has approached microfluidic programming from the perspective that fluidic resources should be statically managed. This provides guarantees that a program will "fit" on the microfluidic device, but limits the expressive power of the programming model. Specifically, any programming feature that allows for potentially unbounded fluidic resource usage must be either prohibited or restricted, as the tool cannot statically perform place and route.

General-purpose computation has taken the opposite tack. Nearly all of the abstractions that make up the modern computing stack (virtual memory, stack, heap, processes, file systems, etc.) provide the illusion of exclusive access and unlimited resources. The systems under these abstractions cannot statically promise that enough resources exist, but their popularity demonstrates the value of favoring expressiveness over static safety. Barring the development and use of sufficiently complex static analyses, existing fluidic programming systems cannot coexist with general purpose computation: one side wants to make promises that the other has given up on.

Puddle's design draws more inspiration from general purpose computer systems than synthesis tools for microfluidics. Instead of trying to provide a monolithic solution, Puddle instead defines a small API that decouples fluidic management from the act of programming. This design leads to many benefits that, to our knowledge, no other DMF programming system has.

Most importantly, Puddle abstracts away fluid management without restricting the programming model. This flexibility is important since our target domains (automated experimentation, molecular computer systems, etc.) are new and have not yet settled on abstractions. The separation of concerns in our approach allows the frontend(s) and the runtime system to advance independently.

The notion of opaque handles (droplet ids) representing dynamically managed resources (droplets) lets Puddle provide many features familiar to computer systems with very little effort. Like the abstractions mentioned above, Puddle lets the user pretend they have unlimited resources (space on the microfluidic board) and exclusive access to it. Puddle can provide process-like isolation by simply name-spacing droplet ids with a UUID, so multiple programs can run on a single device concurrently. The Puddle API can also be used locally (on the Raspberry Pi) or remotely, useful for interactive programming or protocols that require heavy data analysis, respectively.

Unlike static approaches to digital microfluidic programming [15, 20, 39], dynamism also allows for metaprogramming. Users can use features of their favorite languages (reflection, macros, first-class functions, etc.) to programmatically construct protocols.

Error correction is also closely tied to our dynamic approach. Puddle implements error detection and correction using a computer vision system (details in Section 4.3). Abstract droplet ids allow the runtime system to freely move droplets around to correct and avoid errors. Since the nature of a hardware failure cannot be known ahead of time, even static approaches to microfluidic programming that include this kind of error correction must resort to dynamic replanning, limiting the static promises that can be made in the face of errors.

7.2 Drawbacks of Dynamism

These benefits come with a trade-off, of course: we give up all hope of statically reasoning about the program. One concern is that the lack of static analysis allows programs to run that will crash due to space constraints. This is technically true, but the problem can be mitigated by simulation. Section 4.2.3 discusses the role of simulation in planning execution, but it can also be used at the user level. When running in simulation-only mode, Puddle plans execution (including place and route), returning dummy values for sensor readings if needed. If the fluidics portion of the program

is "straight-line" code (i.e., with no branches), then the simulation will tell the user if their program is infeasible before executing on the DMF device, just like static approaches.

For more complex programs, it is intractable to explore all the possible sequences of fluidic operations in simulation. Since Puddle has no knowledge of the client's program, it is entirely possible to get to a state where resource requests cannot be met, forcing Puddle to crash. We hope to address this in future work, but we also view it as part of the expressiveness trade-off. Consider the thermocycle code in Figure 11: it loops over a data structure of potentially unbounded size and potentially inputs a new droplet each time. Even though this program is practical and useful, an ahead-of-time compiler would see it as requiring unbounded resources and fail to generate a static plan[2].

A final concern is that the dynamic approach leads to inefficient placement and routing. Our prototype implementation is simple, but taking advantage of more efficient solutions from the literature [8, 19, 32, 46] is a matter of implementation. Because the droplet manipulation API is functional and "lazy", unfinished operations can form a DAG. The operations themselves are the nodes and the droplet ids are the edges, connecting operations based on their data dependence. DAGs formed this way could contain even more information that those from static approaches, as dynamically constructed DAGs can easily capture inter-procedural droplet dependencies. These DAGs are analogous to those studied in the literature, so we could use all the same placement and routing algorithms developed by existing work without giving up the ability to write rich programs.

Despite the laziness of the API, it is still possible for users to write programs that Puddle would fundamentally not be able to optimize, even assuming the unimplemented ability to do parallel planning. Consider the following snippet where prep does a lot of manipulation and sense takes some kind of sensor reading:

```
prep(a); prep(b); sense(a); sense(b);
```

When droplet a is sensed, Puddle will execute the planned work from prep(a), and likewise for when b is sensed. Even though the two calls to prep are independent, Puddle would not exploit this fact. This is necessary because when sense(a) is called, Puddle still thinks that b might be manipulated further, and Puddle aims to perform dependent operations with minimal delay between them. The programmer could solve this by inserting flush(a, b) before the calls to sense, telling Puddle to realize those droplet now. Another approach would be to sense the independent droplets concurrently on the client side, so one does not block the other. On the server side, Puddle could coalesce the two requests and possibly plan them in parallel.

---

[2] Assuming the static compiler is not smart enough to reason about the volume comparison. In reality, the droplet should not grow too far beyond that min_volume.

7.3 Future Work

The complexity of new applications provides challenges for future work at all levels of the stack: programming model, runtime system, and hardware. Our dynamic approach maximizes expressiveness, trading off the ability to statically determine if a given program is feasible on a particular DMF device. While simulation suffices for now, future, more complex protocols would benefit from a static guarantee, but only if it does not compromise the programming model. The rich domains in which fluidic systems are used also pose a challenge to programming languages: can we statically reason about the contents of the fluids? Recent work has approached this from a chemical safety perspective, preventing accidental dangerous reactions [39].

At the runtime system level, we hope to incorporate more advanced place and route techniques from the literature into Puddle. As Puddle already provides process-like concurrency, algorithms that extract parallelism and prevent contamination (by keeping certain paths separate) are especially promising. With further advances in DMF hardware, a larger DMF system combined with Puddle could enable a multi-user system where many protocols are running on "virtualized" DMFs that share common resources like fluidic input/output, heaters, etc.

7.4 Conclusion

We have presented Puddle, a runtime system that provides a high-level API for microfluidic programming. By taking a more dynamic approach than past work, Puddle allows an unrestricted programming model in a general-purpose language as well as real-time error correction. Puddle facilitates decisions at the execution, protocol, and application level, all of which are necessary for robust execution of complex protocols. Puddle allows scientists to develop protocols (even interactively) in a familiar language.

We have also presented PurpleDrop, an accessible DMF hardware platform with features like fluidic input/output necessary for hands-off automation of many protocols. PurpleDrop's design is simple and cheap enough for labs to assemble their own. Together, Puddle and PurpleDrop comprise the first full-stack microfluidics platform that is accessible and flexible enough to enable complex applications requiring a combination of computation and fluidic manipulation.

References

[1] [n. d.]. Raspberry Pi 3B. https://www.raspberrypi.org/products/raspberry-pi-3-model-b/

[2] Mirela Alistar and Urs Gaudenz. 2017. OpenDrop: An Integrated Do-It-Yourself Platform for Personal Use of Biochips. *Bioengineering* 4, 2 (2017), 45.

[3] Mirela Alistar, Elena Maftei, Paul Pop, and Jan Madsen. 2010. Synthesis of biochemical applications on digital microfluidic biochips with operation variability. In *2010 Symposium on Design Test Integration and Packaging of MEMS/MOEMS (DTIP)*. 350–357.

[4] Mirela Alistar, Paul Pop, and Jan Madsen. 2012. Online synthesis for error recovery in digital microfluidic biochips with operation variability. In *2012 Symposium on Design, Test, Integration and Packaging of MEMS/MOEMS (DTIP)*. IEEE, 53–58.

[5] Ahmed M Amin, Mithuna Thottethodi, TN Vijaykumar, Steven Wereley, and Stephen C Jacobson. 2007. Aquacore: a programmable architecture for microfluidics. In *ACM SIGARCH Computer Architecture News*, Vol. 35. ACM, 254–265.

[6] Vaishnavi Ananthanarayanan and William Thies. 2010. Biocoder: A programming language for standardizing and automating biology protocols. *Journal of Biological Engineering* 4, 1 (2010), 13.

[7] Biddut Bhattacharjee and Homayoun Najjaran. 2012. Droplet sensing by measuring the capacitance between coplanar electrodes in a digital microfluidic system. *Lab on a Chip* 12, 21 (2012), 4416–4423.

[8] Karl F. Bohringer. 2006. Modeling and controlling parallel tasks in droplet-based microfluidic systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 25, 2 (Feb 2006), 334–344. https://doi.org/10.1109/TCAD.2005.855958

[9] Gary Bradski and Adrian Kaehler. 2000. OpenCV. *Dr. Dobb's journal of software tools* 3 (2000).

[10] Douglas Carmean, Luis Ceze, Georg Seelig, Kendall Stewart, Karin Strauss, and Max Willsey. 2019. DNA data storage and hybrid molecular-electronic computing. *Proc. IEEE* 107, 1 (Jan 2019), 63–72. https://doi.org/10.1109/JPROC.2018.2875386

[11] Kihwan Choi, Alphonsus H.C. Ng, Ryan Fobel, and Aaron R. Wheeler. 2012. Digital Microfluidics. *Annual Review of Analytical Chemistry* 5, 1 (2012), 413–440. https://doi.org/10.1146/annurev-anchem-062011-143028

[12] George M. Church, Yuan Gao, and Sriram Kosuri. [n. d.]. Next-generation digital information storage in DNA. 337, 6102 ([n. d.]), 1628–1628. https://doi.org/10.1126/science.1226355

[13] Peter J.A. Cock, Tiago Antao, Jeffrey T. Chang, Brad A. Chapman, Cymon J. Cox, Andrew Dalke, Iddo Friedberg, Thomas Hamelryck, Frank Kauff, Bartek Wilczynski, et al. 2009. Biopython: freely available Python tools for computational molecular biology and bioinformatics. *Bioinformatics* 25, 11 (2009), 1422–1423.

[14] Beatriz Coelho, Bruno Veigas, Elvira Fortunato, Rodrigo Martins, Hugo Águas, Rui Igreja, and Pedro V. Baptista. 2017. Digital microfluidics for nucleic acid amplification. *Sensors* 17, 7 (2017), 1495.

[15] Christopher Curtis, Daniel Grissom, and Philip Brisk. 2018. A compiler for cyber-physical digital microfluidic biochips. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*. ACM, 365–377.

[16] Ryan Fobel, Christian Fobel, and Aaron R. Wheeler. 2013. DropBot: An open-source digital microfluidic control system with precise control of electrostatic driving force and instantaneous drop velocity measurement. *Applied Physics Letters* 102, 19 (2013), 193513.

[17] Nick Goldman, Paul Bertone, Siyuan Chen, Christophe Dessimoz, Emily M. LeProust, Botond Sipos, and Ewan Birney. 2013. Towards practical, high-capacity, low-maintenance information storage in synthesized DNA. *Nature* 494, 7435 (2013), 77.

[18] Jian Gong and Chang-Jin Kim. 2008. Direct-referencing two-dimensional-array digital microfluidics using multilayer printed circuit board. *Journal of Microelectromechanical Systems* 17, 2 (2008), 257–264.

[19] Daniel Grissom and Philip Brisk. 2012. Path scheduling on digital microfluidic biochips. In *Proceedings of the 49th Annual Design Automation Conference (DAC '12)*. ACM, New York, NY, USA, 26–35. https://doi.org/10.1145/2228360.2228367

[20] Daniel Grissom, Christopher Curtis, and Philip Brisk. 2014. Interpreting assays with control flow on digital microfluidic biochips. *ACM Journal on Emerging Technologies in Computing Systems (JETC)* 10, 3 (2014), 24.

[21] Daniel Grissom, Christopher Curtis, Skyler Windh, Calvin Phung, Navin Kumar, Zachary Zimmerman, O'Neal Kenneth, Jeffrey McDaniel, Nick Liao, and Philip Brisk. 2015. An open-source compiler and PCB synthesis tool for digital microfluidic biochips. *INTEGRATION, the VLSI journal* 51 (2015), 169–193.

[22] Philip Guo. 2014. Python is now the most popular introductory teaching language at top US universities. *Communications in ACM, Blogs* (2014).

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

[24] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. 1968. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics* 4, 2 (7 1968), 100–107. https://doi.org/10.1109/TSSC.1968.300136

[25] Yi-Ling Hsieh, Tsung-Yi Ho, and Krishnendu Chakrabarty. 2014. Biochip synthesis and dynamic error recovery for sample preparation using digital microfluidics. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 33, 2 (Feb. 2014), 183–196. https://doi.org/10.1109/TCAD.2013.2284010

[26] Kai Hu, Bang-Ning Hsu, Andrew Madison, Krishnendu Chakrabarty, and Richard Fair. 2013. Fault detection, real-time error recovery, and experimental demonstration for digital microfluidic biochips. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '13)*. EDA Consortium, San Jose, CA, USA, 559–564. http://dl.acm.org/citation.cfm?id=2485288.2485426

[27] Tsung-Wei Huang, Chun-Hsien Lin, and Tsung-Yi Ho. 2010. A contamination aware droplet routing algorithm for the synthesis of digital microfluidic biochips. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 29, 11 (Nov 2010), 1682–1695. https://doi.org/10.1109/TCAD.2010.2062770

[28] Christopher Jaress, Philip Brisk, and Daniel Grissom. 2015. Rapid online fault recovery for cyber-physical digital microfluidic biochips. In *2015 IEEE 33rd VLSI Test Symposium (VTS)*. 1–6. https://doi.org/10.1109/VTS.2015.7116246

[29] Mais J. Jebrail, Ronald F. Renzi, Anupama Sinha, Jim Van De Vreugde, Carmen Gondhalekar, Cesar Ambriz, Robert J. Meagher, and Steven S. Branda. 2015. A solvent replenishment solution for managing evaporation of biochemical reactions in air-matrix digital microfluidics devices. *Lab on a Chip* 15, 1 (2015), 151–158.

[30] Eric Jones, Travis Oliphant, and Pearu Peterson. 2014. SciPy: open source scientific tools for Python. (2014).

[31] Ben Keller, Justin Vrana, Abraham Miller, Garrett Newman, and Eric Klavins. 2019. Aquarium: The Laboratory Operating System. https://doi.org/10.5281/zenodo.2535715

[32] Oliver Keszocze, Robert Wille, Krishnendu Chakrabarty, and Rolf Drechsler. 2015. A general and exact routing methodology for digital microfluidic biochips. In *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 874–881. https://doi.org/10.1109/ICCAD.2015.7372663

[33] Ross D. King, Jem Rowland, Stephen G. Oliver, Michael Young, Wayne Aubrey, Emma Byrne, Maria Liakata, Magdalena Markham, Pinar Pir, Larisa N. Soldatova, Andrew Sparkes, Kenneth E. Whelan, and Amanda Clare. 2009. The automation of science. *Science* 324, 5923 (2009), 85–89. https://doi.org/10.1126/science.1165620

arXiv:http://science.sciencemag.org/content/324/5923/85.full.pdf

[34] Ross D. King, Kenneth E. Whelan, Ffion M. Jones, Philip G.K. Reiser, Christopher H. Bryant, Stephen H. Muggleton, Douglas B. Kell, and Stephen G. Oliver. 2004. Functional genomic hypothesis generation and experimentation by a robot scientist. *Nature* 427, 6971 (2004), 247.

[35] Ali Sinan Koksal, Yewen Pu, Saurabh Srivastava, Rastislav Bodik, Jasmin Fisher, and Nir Piterman. 2013. Synthesis of Biological Models from Mutation Experiments. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '13)*. ACM, New York, NY, USA, 469–482. https://doi.org/10.1145/2429069.2429125

[36] Yan Luo, Krishnendu Chakrabarty, and Tsung-Yi Ho. 2013. Error recovery in cyberphysical digital microfluidic biochips. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 32, 1 (2013), 59–72.

[37] James Munkres. 1957. Algorithms for the assignment and transportation problems. *J. Soc. Indust. Appl. Math.* 5, 1 (1957), 32–38.

[38] Lee Organick, Siena Dumas Ang, Yuan-Jyue Chen, Randolph Lopez, Sergey Yekhanin, Konstantin Makarychev, Miklos Z. Racz, Govinda Kamath, Parikshit Gopalan, Bichlien Nguyen, Christopher N. Takahashi, Sharon Newman, Hsing-Yeh Parker, Cyrus Rashtchian, Kendall Stewart, Gagan Gupta, Robert Carlson, John Mulligan, Douglas Carmean, Georg Seelig, Luis Ceze, and Karin Strauss. 2018. Random access in large-scale DNA data storage. *Nature Biotechnology* 36, 3 (Mar 2018), 242–248. https://doi.org/10.1038/nbt.4079

[39] Jason Ott, Tyson Loveless, Chris Curtis, Mohsen Lesani, and Philip Brisk. 2018. BioScript: programming safe chemistry on laboratories-on-a-chip. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 128.

[40] Oxford Nanopore. [n. d.]. MinION. https://nanoporetech.com/products/minion

[41] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in PyTorch. In *NIPS-W*.

[42] John Paul Urbanski, William Thies, Christopher Rhodes, Saman Amarasinghe, and Todd Thorsen. 2006. Digital microfluidics using soft lithography. *Lab on a Chip* 6, 1 (2006), 96–104. https://doi.org/10.1039/B510127A

[43] Michael G. Pollack, Richard B. Fair, and Alexander D. Shenderov. 2000. Electrowetting-based actuation of liquid droplets for microfluidic applications. *Applied Physics Letters* 77, 11 (2000), 1725–1726.

[44] Pzucchel. [n. d.]. https://commons.wikimedia.org/wiki/File:Automated_pipetting_system_using_manual_pipettes.jpg

[45] Lulu Qian and Erik Winfree. 2011. Scaling up digital circuit computation with DNA strand displacement cascades. *Science* 332, 6034 (2011), 1196–1201.

[46] Pranab Roy, Hafizur Rahaman, and Parthasarathi Dasgupta. 2010. A novel droplet routing algorithm for digital microfluidic biochips. In *Proceedings of the 20th Symposium on Great Lakes Symposium on VLSI (GLSVLSI '10)*. ACM, New York, NY, USA, 441–446. https://doi.org/10.1145/1785481.1785583

[47] Michael I. Sadowski, Chris Grant, and Tim S. Fell. 2016. Harnessing QbD, programming languages, and automation for reproducible biology. *Trends in biotechnology* 34, 3 (2016), 214–227.

[48] Gisbert Schneider. 2017. Automating drug discovery. *Nature Reviews Drug Discovery* 17, 2 (2017), 97.

[49] Thermofisher Scientific. [n. d.]. Qubit Fluorometric DNA Quantitation. https://www.thermofisher.com/us/en/home/industrial/spectroscopy-elemental-isotope-analysis/molecular-spectroscopy/fluorometers/qubit.html

[50] Yong-Jun Shin and Jeong-Bong Lee. 2010. Machine vision for digital microfluidics. *Review of Scientific Instruments* 81, 1 (2010), 014302. https://doi.org/10.1063/1.3274673 arXiv:https://doi.org/10.1063/1.3274673

[51] Andrew Sparkes, Wayne Aubrey, Emma Byrne, Amanda Clare, Muhammed N. Khan, Maria Liakata, Magdalena Markham, Jem Rowland, Larisa N. Soldatova, Kenneth E. Whelan, Michael Young, and Ross D. King. 2010. Towards robot scientists for autonomous scientific discovery. *Automated Experimentation* 2, 1 (2010), 1.

[52] Kendall Stewart, Yuan-Jyue Chen, David Ward, Xiaomeng Liu, Georg Seelig, Karin Strauss, and Luis Ceze. 2018. A content-addressable DNA database with learned sequence encodings. In *24th International Conference On DNA Computing and Molecular Programming (DNA 24)*. https://homes.cs.washington.edu/~kstwrt/pubs/dna24.pdf

[53] Satoshi Suzuki et al. 1985. Topological structural analysis of digitized binary images by border following. *Computer vision, graphics, and image processing* 30, 1 (1985), 32–46.

[54] Synthace. 2018. Antha. https://synthace.com/introducing-antha

[55] Sven Tombrink and iX factory. [n. d.]. https://commons.wikimedia.org/wiki/File:Microfluidic_Chip_iX-factory.jpg

[56] Transcriptic. 2018. Autoprotocol. http://autoprotocol.org/

[57] Udayan Umapathi, Patrick Shin, Ken Nakagaki, Daniel Leithinger, and Hiroshi Ishii. 2018. Programmable droplets for interaction. In *Extended Abstracts of the 2018 CHI Conference on Human Factors in Computing Systems*. ACM, VS15.

[58] Philippe Q. N. Vo, Mathieu C. Husser, Fatemeh Ahmadi, Hugo Sinha, and Steve C. C. Shih. 2017. Image-based feedback and analysis system for digital microfluidics. *Lab on a Chip* 17, 20 (2017), 3437–3446.

[59] Tao Xu and Krishnendu Chakrabarty. 2008. Integrated droplet routing and defect tolerance in the synthesis of digital microfluidic biochips. *Journal of Emerging Technologies in Computing Systems* 4, 3, Article 11 (Aug. 2008), 24 pages. https://doi.org/10.1145/1389089.1389091

[60] Tao Xu, Krishnendu Chakrabarty, and Fei Su. 2008. Defect-aware high-level synthesis and module placement for microfluidic biochips. *IEEE Transactions on Biomedical Circuits and Systems* 2, 1 (March 2008), 50–62. https://doi.org/10.1109/TBCAS.2008.918283

[61] Hailong Yao, Qin Wang, Yiren Shen, Tsung-Yi Ho, and Yici Cai. 2016. Integrated functional and washing routing optimization for cross-contamination removal in digital microfluidic biochips. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35, 8 (Aug 2016), 1283–1296. https://doi.org/10.1109/TCAD.2015.2504397

[62] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. 2016. Apache Spark: a unified engine for big data processing. *Commun. ACM* 59, 11 (Oct. 2016), 56–65. https://doi.org/10.1145/2934664

[63] David Yu Zhang and Georg Seelig. [n. d.]. Dynamic DNA nanotechnology using strand-displacement reactions. 3, 2 ([n. d.]), 103. https://doi.org/10.1038/nchem.957

[64] Yang Zhao and Krishnendu Chakrabarty. 2012. Cross-contamination avoidance for droplet routing in digital microfluidic biochips. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 31, 6 (June 2012), 817–830. https://doi.org/10.1109/TCAD.2012.2183369