Scaling Microfluidics to Complex, Dynamic Protocols

Invited Paper

Max Willsey*, Ashley Stephenson*, Chris Takahashi*, Bichlien Nguyen†, Karin Strauss†, and Luis Ceze*
*Paul G. Allen School of Computer Science & Engineering, University of Washington †Microsoft
Email: mwillsey@cs.washington.edu

Abstract—**Microfluidic devices promise to automate wetlab procedures by manipulating small chemical or biological samples. We are developing a full-stack microfluidic automation platform that allows and allows users to scale up the complexity of microfluidic programming, encouraging them to mix fluidic manipulations with traditional programming.**

Puddle 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, resulting in more expressive programs. It also provides real-time error correction through a computer vision system, allowing robust execution on cheaper digital microfluidic hardware.

We have been running Puddle on PurpleDrop, a new digital microfluidic device that is affordable and has novel features such as fully automated input/output of fluids. With this combination, we have demonstrated PCR with automated replenishment, a DNA sequencing preparation protocol, and the complete retrieval of digital data stored in dehydrated spots of DNA on the device's surface.

Going forward, we see Puddle and PurpleDrop as part of a platform for further research. PurpleDrop is affordable and extensible, which makes a compelling case for adding new periferials or even scaling out by connecting multiple devices. And Puddle provides a flexible and abstract programming model that could enable microfluidic programs to run on different hardware targets (DMF or liquid handling robots), or even a combination thereof.

I. 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.

Puddle [1] is a a full-stack, open-source[1] microfluidics system with a high-level programming model that allows unrestricted combination of computation and fluidics. Instead of a new programming language, Puddle is a runtime system that provides microfluidic manipulations through an API. Users write programs against the Puddle API, and Puddle dynamically manages the fluidic resources.

The relatively small Puddle API abstracts away low-level details like fluidic location, providing many benefits to users:

- Users can write high-level programs that dynamically combine fluidics and computation in Python (or the general-purpose programming language of their choice).
- Users can program interactively and design, share, and reuse domain-specific libraries.
- Multiple users can safely and simultaneously run protocols on the same device.
- Protocols can run on different hardware with little to no modification.

The same abstraction allows users who would like the extend Puddle flexibility along many axes:

- Users can define their own operations for Puddle to plan and execute.
- Researchers can modify Puddle to use new, advanced placement and routing algorithms.
- Researchers or hardware designers can even define new hardware backends.

These benefits stem from Puddle's key innovation: dynamically managing resources (i.e., fluids) in response to API calls. 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 current hardware backend is PurpleDrop [1], an affordable general-purpose microfluidic device with capabilities such as fully-automated fluidic input/output, heating, and a camera to power our computer vision error correction system. Puddle and PurpleDrop constitute a complete system stack for microfluidic programming.

This paper will describe the design of Puddle and how it allows for the power and flexibility described above. We will also document some of the benchmarks and case studies we have performed. Finally, we will conclude with ideas for future work that builds on Puddle.

II. Dynamic Microfluidic Programming

We aim to enable users to combine computation and fluidic manipulation in an unrestricted, high-level programming

[1]Both the Puddle software and the PurpleDrop hardware are open-source and available at `http://puddle.bio`

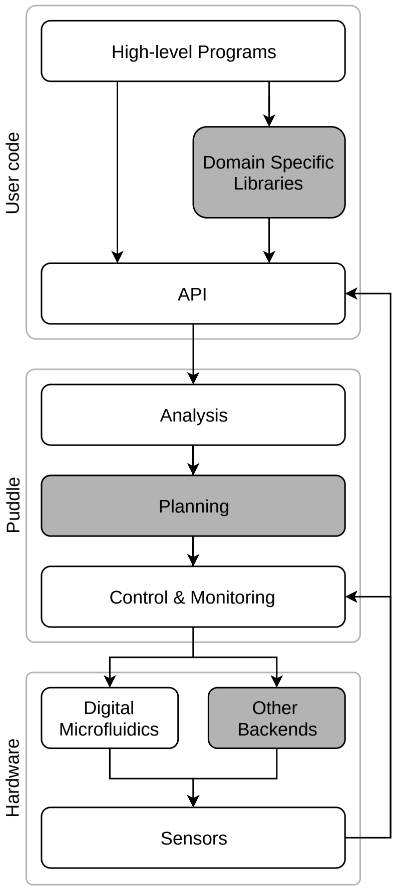

Fig. 1: Users program in a high-level, general-purpose language where they can combine Puddle's primitives into domain-specific libraries. Programs call into the Puddle runtime, which plans execution and controls the hardware. Sensor data can be returned to the user through the API, and it also informs the execution. Darkened boxes indicate potential for expansion in future work.

model. We picture programs not only reacting to sensor data, but using metaprogramming, cloud resources, machine learning, and other techniques from the full range of computing in conjunction with microfluidics. Accomplishing this hinges on allowing the user to program in a general-purpose programming language.

Unlike other solutions for microfluidics programming [2], [3], [4], [5], 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. As a result, Puddle has no knowledge of the control structure of user's program, but it does discover the data flow through non-blocking API calls. This lack of knowledge is the key trade-off of Puddle's design: it allows a completely unrestricted programming model in any programming language, but it prevents Puddle from making static guarantees about the (error-free) execution of a protocol

```
min_volume = 10 * microliters

def thermocycle(droplet, temps_and_times):
   for temp, time in temps_and_times:
      heat(droplet, temp, time)
      if volume(droplet) < min_volume
         # '+=' is a mutating mix
         droplet += input("water", min_volume)

def pcr(droplet, n_iter):
   iters = n_iter * [
      (62, 30 * seconds),
      (72, 20 * seconds),
   ]
   params = [(95, 30 * seconds)] + iters
   thermocycle(droplet, params)
```

Fig. 2: 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. `3 * [1] == [1, 1, 1]`

like some synthesis tools could.

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 snippets shown throughout the paper will have calls into Puddle underlined.

A. *Example*

Figure 2 shows some example Puddle code in the Python frontend. It defines two regular Python functions. The `thermocycle` function repeatedly heats up a given droplet based on a given Python list of temperatures and times. The `pcr` function performs polymerase chain reaction, which duplicates DNA by thermocycling the droplet with a specific set of parameters.

Importantly, all of the control is handled by Python. The functions, loops, lists, and if statements all work as expected. While the Puddle API is functional (see below), each frontend is free to wrap the API in a way that is idiomatic for the language. In Python, the `heat` API call is mutating, and the += operator is overloaded to perform a mutating mix.

When the `pcr` function is called, `thermocycle` is called and a `heat` API call is made. That and all other fluidic actuations are *non-blocking*, as described below, so Puddle quickly returns an opaque token. Non-blocking calls give the runtime system more flexibility in how to implement operations on the microfluidic device.

On the other hand, the `volume` API call is a sensor reading, and is therefore blocking. The volume of a droplet is a dynamic property; it cannot (in general) be known statically. 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) could manipulate and branch on.

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`($d_1, d_2, \ldots$)

Fig. 3: The Puddle API. ds are droplet ids, opaque handles to droplets. All calls are non-blocking except for those under Sensing and the special `flush` call.

B. 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 Puddle API is listed in Figure 3. 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; they mix, split, or heat their arguments. Note that they are functional, consuming their droplet id arguments and returning new ones.

The sensing API calls force the system to "flush" by performing the operations necessary to actually produce the droplet to be sensed. The `volume` operation reads the volume using the camera. The `temperature` operation measures the temperature using an onboard sensor.

The `flush` operation allows the user to manually force Puddle to realize the specified droplets (or all of them, if none are given), which is useful in interactive programming.

Users can easily extend the Puddle API with their own actuation and sensing primitives. The mechanism for allocating space on the device (Section III-B2) is sufficiently general to implement other primitives that the hardware might support.

C. Handling Errors

API calls can fail instantaneously for two reasons: invalid arguments or using a consumed droplet id, as droplets are physical resources that can only be consumed once. 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). It is the programmer's responsibility to not reuse droplet ids that have been consumed. An imperative interface (like `heat` in Figure 2) 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 III-C). 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.

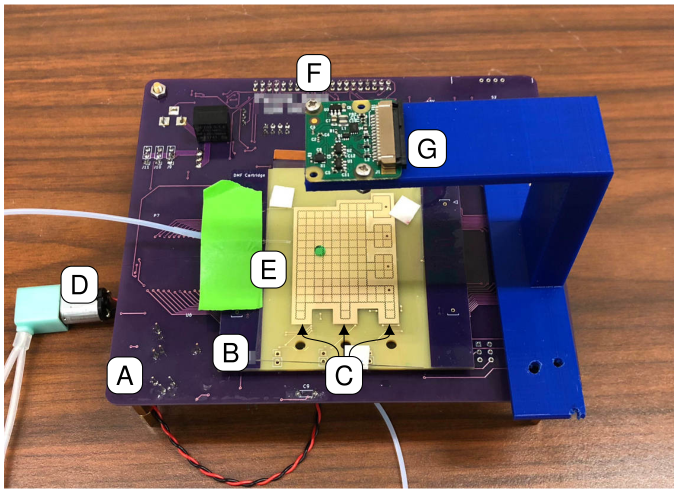

Fig. 4: PurpleDrop, our digital microfluidic device. The parent PCB (A) contains the electronic components, and the child (B) contains the electrodes and the hydrophobic surface. The device supports heaters on the bottom three electrodes (C). We can drive up to three pumps (D) that can input or output 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).

III. 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 system which implements the API detailed in Section II, 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.

A. 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 many 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 4 shows the device and enumerates its components.

PurpleDrop runs in air (without an oil medium) for easier setup. Input and output are driven by small peristaltic pumps which carry droplets to/from test tube reservoirs or other

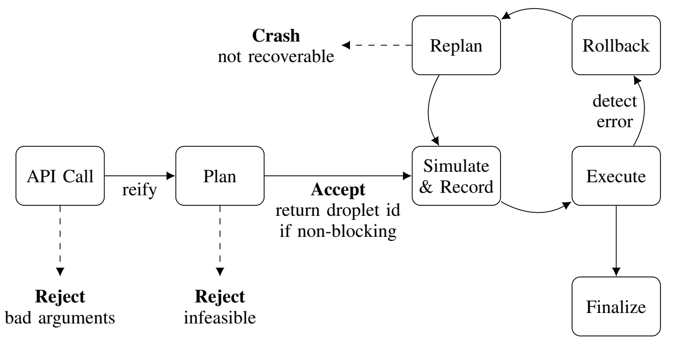

Fig. 5: The life of a command. Dashed edges end the flow immediately. Commands may be rejected if the API call was malformed or a feasible execution plan cannot be found. The user can recover from rejected commands, but not from failures to replan accepted ones.

devices. PurpleDrop also includes heaters and temperature sensors. A camera mounted on top of the device serves as a multi-purpose sensor; it can detect the volume of droplets, and it also powers our error detection system.

B. 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.

Figure 5 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 without modifying the planning and execution infrastructure.

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.

2) Planning: Reified commands form a DAG where the edges are their droplet id dependencies. A scheduler chooses which pending commands to plan and execute.

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. A placement algorithm finds a way to satisfy the request. The allocation request also specifies the (relative) desired locations of the input droplets. In the case of mixing two 1×1 droplets, for example, `mix` will request a 3×2 rectangle and ask that the input droplets start at coordinates $(0, 0)$ and $(1, 0)$. After the allocation request is placed, the router finds paths for the input droplets to their specified locations. 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. Planning happens right after command creation, so 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).

Puddle is modular over its placements and routing algorithms. Researchers could easily be plug in different algorithms without modifying the frontend, commands, or hardware backends.

3) Simulation and Recording: Once a blocking command comes in, before execution begins, the planned commands are first 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.

4) Execution, Monitoring, and Rollback: Execution simply consists of popping the earliest state from the simulation 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 III-C) 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 [6] that uses capacitance sensing instead of computer vision.

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 5). 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 IV-A 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.

C. Error Detection via Computer Vision

The previous section discussed how Puddle corrects errors via the rollback and replan mechanism, but not how we detect

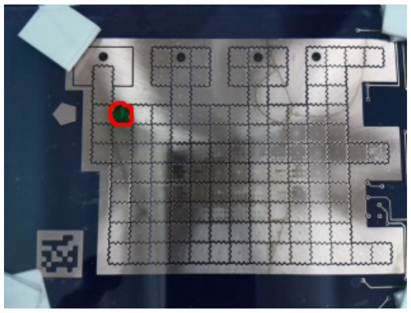
Fig. 6: A computer vision system identifies droplets in real time for error detection and volume measurement.

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 [7] or a reference background image of the electrode array [8], [9].

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. Compared to other approaches, ours scales to different sizes of droplets and is less sensitive to light changes.

The shape detection portion of error detection is implemented in OpenCV [10]. Once the shapes are detected, Puddle must determine if the set of shapes constitutes an error. Distances between expected and actual form a bipartite graph, and we find a matching using the Kuhn-Munkres algorithm [11]. 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.

IV. Case Studies

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.

A. Error Correction

In Puddle, we use the droplet position and size information as part of a larger error correction system including droplet matching, rollback, and replanning (detailed in Section III-B). 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

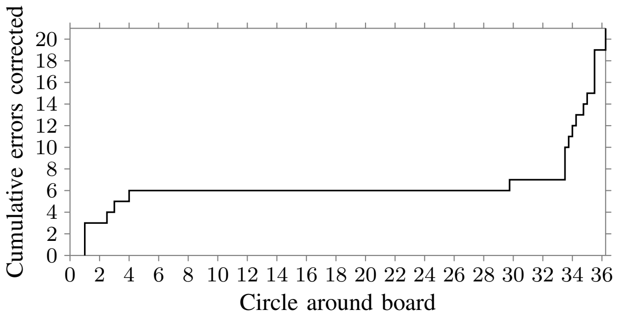

Fig. 7: To test error correction, we moved a droplet in circles until failure. Total experiment time was 2 hours and 11 minutes. Along the way, the computer vision system detected and corrected errors, marking regions of the board as faulty and avoiding them in the future. The errors at the beginning correspond to faulty electrodes; those at the end were caused as the droplet evaporated.

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 7 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.

B. 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): 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 2 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

```
def dna_lookup(key):
  spot = SPOTS[key]
  d = input(...)
  d.mix_at(spot)
  sleep(60 * seconds)

  output("sequencer", d)

  data = get_data()
  seq = process(data)
  return seq
```

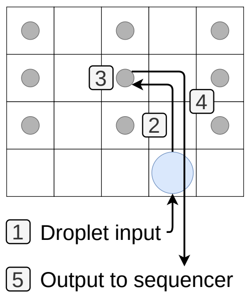

Fig. 8: Code and diagram for DMF retrieval of dehydrated DNA samples. The gray dots represent the DNA samples dried onto the glass top-plate. To retrieve a sample, (1) a droplet is added to the board, (2) the droplet navigates to the sample of interest and (3) sits under the dehydrated spot, rehydrating the sample in the droplet. The droplet can then (4) proceed the edge of the board, and (5) get outputted to the DNA sequencer.

protocol in synthetic biology that selectively amplifies DNA in a solution. 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.

C. Key DNA

In prior work [12], we demonstrated the ability to store and retrieve dehydrated samples of DNA on the glass top-plate of our DMF device. Figure 8 describes how the storage and retrieval works. The prior work focuses on characterizing the capabilities of such a storage scheme. In short, retrieval and successful sequencing of the samples is possible, and any contamination was well within noise of the inherently stochastic sequencing process.

We can write the protocol from that work in Puddle to further demonstrate the power of Puddle's programming model. Figure 8 also includes code that would perform the protocol as described in the figure, with the addition of looking up the spot location in a table (SPOTS). The snippet also includes pseudo-code for performing the actual sequencing. We have demonstrated that we can output samples from PurpleDrop directly onto DNA sequencers, but we did not find an API to automate the sequencing, a human still had to initiate it. But given such an API, the code in the figure represents a single Python function that takes in data (a key), returns data (the sequence of the DNA at that spot), and uses microfluidics in between.

V. Conclusion

We presented Puddle, an open-source system that provides a high-level API for microfluidic manipulations. Using this API from a general-purpose programming language enables unprecedented flexibility, allowing programs to freely combine traditional and microfluidic programming. We also described PurpleDrop, and we enumerated some uses of the Puddle/PurpleDrop stack that highlight this style of programming.

Presently, we use PurpleDrop exclusively as the hardware backend. We also use relatively simple algorithms for placement and routing. But, as shown in Figure 1, Puddle is rich in opportunties for future work. More operations, more advanced placement and route algorithms, and even new hardware backends could be implemented without breaking the API. The flexiblity over backend is particularly exciting, as it could provide the foundation for a unified microfluidic programming model over many different devices. This could also be a basis for combining backends, creating a heterogeneous microfluidic platform that combines the best of several devices.

References

[1] M. Willsey, A. P. Stephenson, C. Takahashi, P. Vaid, B. H. Nguyen, M. Piszczek, C. Betts, S. Newman, S. Joshi, K. Strauss, and L. Ceze, "Puddle: A dynamic, error-correcting, full-stack microfluidics platform," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19. New York, NY, USA: ACM, 04 2019.

[2] D. Grissom, C. Curtis, S. Windh, C. Phung, N. Kumar, Z. Zimmerman, O. Kenneth, J. McDaniel, N. Liao, and P. Brisk, "An open-source compiler and pcb synthesis tool for digital microfluidic biochips," *INTEGRATION, the VLSI journal*, vol. 51, pp. 169–193, 2015.

[3] C. Curtis, D. Grissom, and P. Brisk, "A compiler for cyber-physical digital microfluidic biochips," in *Proceedings of the 2018 International Symposium on Code Generation and Optimization*. ACM, 2018, pp. 365–377.

[4] J. Ott, T. Loveless, C. Curtis, M. Lesani, and P. Brisk, "Bioscript: programming safe chemistry on laboratories-on-a-chip," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, p. 128, 2018.

[5] V. Ananthanarayanan and W. Thies, "Biocoder: A programming language for standardizing and automating biology protocols," *Journal of Biological Engineering*, vol. 4, no. 1, p. 13, 2010.

[6] K. Hu, B.-N. Hsu, A. Madison, K. Chakrabarty, and R. Fair, "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*, ser. DATE '13. San Jose, CA, USA: EDA Consortium, 2013, pp. 559–564. [Online]. Available: http://dl.acm.org/citation.cfm?id=2485288.2485426

[7] Y. Luo, K. Chakrabarty, and T.-Y. Ho, "Error recovery in cyberphysical digital microfluidic biochips," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 32, no. 1, pp. 59–72, 2013.

[8] Y.-J. Shin and J.-B. Lee, "Machine vision for digital microfluidics," *Review of Scientific Instruments*, vol. 81, no. 1, p. 014302, 2010. [Online]. Available: https://doi.org/10.1063/1.3274673

[9] P. Q. N. Vo, M. C. Husser, F. Ahmadi, H. Sinha, and S. C. C. Shih, "Image-based feedback and analysis system for digital microfluidics," *Lab on a Chip*, vol. 17, no. 20, pp. 3437–3446, 2017.

[10] G. Bradski and A. Kaehler, "Opencv," *Dr. Dobb's journal of software tools*, vol. 3, 2000.

[11] J. Munkres, "Algorithms for the assignment and transportation problems," *Journal of the Society for Industrial and Applied Mathematics*, vol. 5, no. 1, pp. 32–38, 1957.

[12] S. Newman, A. P. Stephenson, M. Willsey, B. H. Nguyen, C. N. Takahashi, K. Strauss, and L. Ceze, "High density dna data storage library via dehydration with digital microfluidic retrieval," *Nature Communications*, vol. 10, no. 1, p. 1706, 4 2019.