Virtualizing Existing Fluidic Programs

CALEB WINSTON, MAX WILLSEY, and LUIS CEZE, University of Washington, USA

Fluidic automation, the practice of programmatically manipulating small fluids to execute laboratory protocols, has led to vastly increased productivity for biologists and chemists. Most fluidic programs, commonly referred to as protocols, are written using APIs that couple the protocol to specific hardware by referring to the physical locations on the device. This coupling makes isolation impossible, preventing portability, concurrent execution, and composition of protocols on the same device.

We propose a system for virtualizing existing fluidic protocols on top of a single runtime system without modification. Our system presents an isolated view of the device to each running protocol, allowing it to assume it has sole access to hardware. We provide a proof-of-concept implementation that can concurrently execute and compose protocols written using the popular Opentrons Python API. Concurrent execution achieves near-linear speedup over serial execution since protocols spend much of their time waiting.

CCS Concepts: • **Hardware → Emerging technologies**; • **Computer systems organization → Embedded and cyber-physical systems**.

Additional Key Words and Phrases: virtualization, fluidics, protocols, pipetting robots, cloud lab, biofoundry

ACM Reference Format:
Caleb Winston, Max Willsey, and Luis Ceze. 2022. Virtualizing Existing Fluidic Programs. 1, 1 (August 2022), 14 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Fluidic automation is revolutionizing the automation of chemical and biological protocols by manipulating small quantities of liquid at higher speeds and precision than humans. Laboratories use these devices to save time, labor, and supplies.

Fluidic automation is performed at widely different scales by a variety of hardware, each of which is backed by a different software stack (Figure 1a-b). This has led to a range of paradigms for writing fluidic programs, or *protocols*. These programming systems [1, 3, 4, 6–8, 12–14, 16, 18, 19, 21] have successfully allowed researchers and users to automate genome editing in zebrafish [10], machine-learning-based discovery of novel biological synthesis, pathways [9], and RT-qPCR COVID-19 testing [11].

Unfortunately, existing fluidic programming systems suffer from a lack of portability and compositionality (Figure 1c). Protocols written in one paradigm with a specific hardware device in mind can seldom be used on a different device without manually porting the protocol. In the same vein, it is nearly impossible to compose protocols that make different assumptions about the underlying system. Even if two protocols are written in the same system for the same machine, composition can be difficult due to hard-coded resource usage. Furthermore, simultaneous execution of protocols on the same hardware is largely unaddressed.

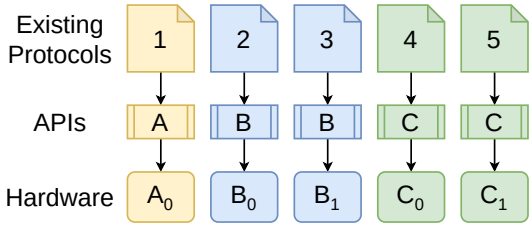
Authors' address: Caleb Winston, calebwin@cs.washington.edu; Max Willsey, mwillsey@cs.washington.edu; Luis Ceze, luisceze@cs.washington.edu, University of Washington, Paul G. Allen Center, Seattle, Washington, USA, 98195.

XXXX-XXXX/2022/8-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

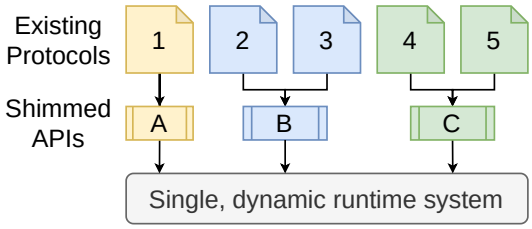
(a) Opentrons OT-2 pipetting robot [12].

```
pip = pipettes["left"]
pip.pick_up_tip()
pip.aspirate(plates[1]["A1"])
pip.dispense(plates[2]["A1"])
pip.aspirate(100, plates[1]["B1"])
pip.dispense(100, plate[2]["A1"])
pip.drop_tip()
```

(b) A simple mixing protocol written in Python using the Opentrons API.



(c) Currently, protocols are typically written using APIs that expose features of the underlying hardware. This locks protocols to that API/hardware and prevents portability and concurrent execution.



(d) We propose that API shims intercept the fluidic "system calls", so a single, dynamic runtime system can support concurrent execution on any hardware that supports the protocols' resource requirements.

Fig. 1. Fluidic automation uses devices such as pipetting robots (a) to execute laboratory procedures. The hardware is controlled programmatically by a *protocol* (b) that typically refers to specific locations or resources on the device, coupling the protocol to that API/hardware (c). We propose virtualization as a solution to allow compositional, concurrent execution (d).

We observe that these issues in the burgeoning space of fluidic programming look similar to the well-understood problem of isolation in computer systems. In both settings, processes (or protocols) would like to assume isolated, complete control over a set of resources on the system. Programs will frequently hard-code memory locations (as protocols will hard-code locations on the fluidic device), preventing simultaneous execution. The solution to both is virtualization on top of single host. By intercepting the right parts of guest programs, the host can present an isolated view of the hardware to each running program. This view can be tweaked to match what the program expects, even if that differs from what the underlying system looks like.

	Static Control Flow	*Dynamic Control Flow*
Physical Addressing	BioBlocks [8], Autoprotocol [5]	Opentrons [12], OpenDrop [1], Antha [14]
Virtual Addressing	AIS [3], BioCoder [4], Roboliq [19]	BioScript [13], Puddle [21]

Fig. 2. Existing fluidic programming systems organized by their addressing mode and control flow support

In this paper, we introduce methods for the virtualization of existing, unmodified fluidic protocols. We propose that a single, dynamic runtime system manages the underlying host hardware (Figure 1d). Previous work [21] has already shown that protocols written against such an API are portable and composable. However, most existing fluidic programs (such as those written for the Opentron platform) are not written against such an API that would directly allow portability and composability. Our key insight is that *existing fluidic programs can be virtualized on top of a dynamic runtime* by intercepting the fluidic "system calls" of the guest protocols.

We provide a proof-of-concept implementation that demonstrates how a single, dynamic runtime can virtualize unmodified protocols written for the popular Opentrons OT-2 pipetting robot [12]. Our system can concurrently execute protocols that would otherwise demand access to the same resources. Concurrent execution can even lead to a performance improvement since protocols can spend a lot of time waiting for reactions to occur or oversubscribed resources such as thermocyclers to become available. With concurrent execution, different protocols can wait simultaneously.

In summary, the contributions of this work are:

- Identifying how API/hardware lock-in leads to lack of portability and composability of existing fluidic protocols
- The concept of virtualizing fluidic protocols on top of a single, dynamic runtime
- A proof-of-concept implementation that demonstrates this technique provides concurrent, compositional execution

2 BACKGROUND

Fluidic programming systems come in many shapes and sizes. Some are entirely custom languages [3, 8, 13, 19], some are deeply embedded domain specific languages [4, 7, 16, 18], and others are simply APIs in a general purpose programming language [1, 6, 12, 14, 21]. Most fluidic programming systems can be organized along two axes (Figure 2): whether they address resources physically or virtually (Section 2.1), and whether they support static or dynamic control flow (Section 2.2).

2.1 Addressing Modes for Fluidic Resources

Physical addressing. Many popular fluidic programming systems (including Autoprotocol [17], Opentrons [12], and Antha [15]) use the *physical addressing mode*, wherein users hard-code physical locations of resources in their programs. These resources include wells in well plates, tubes in tube racks, tips in tip racks, heaters, and other labware and hardware modules. Each of these resources are placed at a precise location on the deck of a pipetting robot and this precise location is akin to a physical address in a computer system. Since the particular geometry of these labware and the properties of the hardware modules may differ in ways that affect protocol results, physical addressing of resources may be desirable.

Consider the simple mixing protocol in Figure 1b. This protocol is implemented using the Opentrons API which requires fluids as well as pipettes and hardware modules to be addressed by their physical locations on the deck of the Opentrons robot. `plates[2]["A1"]` accesses the fluid stored in the well in row A and column 1 of a well plate in slot 2 while `pipettes["left"]` accesses the pipette mounted on the left of the Opentrons robot's robotic arm.

However, physical addressing has its downsides. For many fluidic applications, minute differences in labware and hardware modules are insignificant. Hard-coded physical locations (much like memory addresses in computing) imply an assumption that the running protocol has exclusive access to the hardware, thereby preventing portability, concurrent execution, and composition of protocols.

Virtual addressing. By contrast, consider the protocol in Figure 3. In this protocol, all fluids are referred to by abstract fluid IDs. The reagents `a` and `b` have fluid IDs produced by the operation `In`. These IDs are passed to `Mix` as inputs, rendering the IDs invalid for future use and returning a new fluid ID representing the resulting mixed fluid. `In` and `Out` optionally accept a hard-coded location or allow the system to find an available one. By virtualizing both fluidic I/O (`In` and `Out`) and fluidic manipulation (`Mix` and `WithTemp`), physical addresses are abstracted away.

With physical addresses eliminated from the programming interface, this program may be executed on different pipettes or labware without modification. The underlying runtime is responsible for later determining when and where to execute the fluidic operation on the hardware. Multiple instances may be executed concurrently with physical addresses dynamically allocated for storing the 2 reagents and the intermediate result of the `Mix`.

Virtual addressing of resources is supported by some programming systems coming from the research community (Figure 2), but most protocols in practice are written for platforms that use physical addressing.

2.2 Control Flow in Fluidic Protocols

Static control flow. Many protocols can be expressed as a directed acyclic dataflow graph (DAG) with fluidic operations (e.g. mixing, heating) as nodes and the edges representing the data dependencies, i.e., the fluids produced by one operation and consumed by another. Therefore, some fluidic systems are designed to take such a DAG as input rather than a full program. The DAG input can facilitate virtual addressing, since the DAG can be lowered (via place-and-route) to physical hardware manipulations at compile time. However, this *static control flow* approach limits how protocols can make decision based on real-world data. Prior work [2] found that many protocols are conditioned on simple settings that do not require built-in control flow constructs and can instead be simply toggled at protocol compilation time.

Dynamic control flow. Alternatively, some fluidic programming systems simply offer APIs within a general-purpose programming language like Python. The popular Opentrons system takes this approach (Figure 1a-b), allowing programmers to use *dynamic control flow* to specify fluidic manipulations that may depend on sensor data or other non-static information.

Recent work [13, 21] has developed fluidic programming systems that are both dynamic and virtually-addressed. Consider for example the theromcycling protocol in Figure 3 written in the Puddle [21] framework. This protocol is fully dynamic as it is embedded in Python and all API calls are executed *lazily*. The non-blocking calls to operations such as `In` and `Mix` build up DAGs of fluid IDs and operations which are executed only on calls to `Flush` or other operations that perform sensing and thus require blocking for intermediate fluid IDs to materialize. The higher abstraction level of the dynamic, virtualized approach has already led prior work to discuss the potential for

```
droplet = In(initial_volume)
for temp, time in TEMPS_AND_TIMES:
    droplet = WithTemp(droplet, temp)
    sleep(time)
    if Volume(droplet) < MIN_VOLUME:
        droplet = Mix(droplet, In(MIN_VOLUME))
Flush(Out(droplet))
```

Fig. 3. An auto-replenishing thermocycling protocol using dynamic control flow with virtual addressing [21].

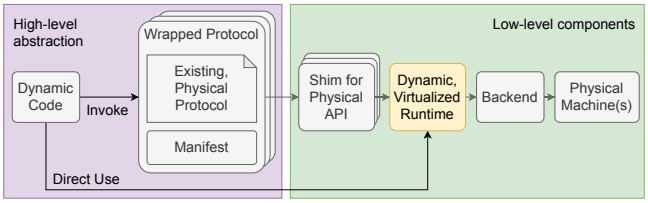


Fig. 4. Architecture for end-to-end virtualization and execution of existing, unmodified fluidic protocols.

isolated, cloud-scale execution of fluidic protocols [20], but these concepts would only apply to protocols written in that framework.

3 METHODOLOGY

We propose methods for the virtualization of existing, unmodified Opentrons [12] protocols. We choose Opentrons for its popularity and the fact that physical, dynamic protocols are the hardest to virtualize.[1]

Our proposed system architecture (Figure 4) relies on a single runtime system that supports dynamic, virtualized fluidic execution (Section 3.1). Shimmed APIs (Section 3.2) can then provide a physically addressed interface to the runtime system. Then, an existing protocol can be paired with a *manifest* (Section 3.3) that specifies its resource requirements. The result is a *wrapped protocol* that can finally be invoked by dynamic, virtualized user-level code (Section 3.4). This wrapped protocol performs the same fluidic manipulation as the original existing protocol but is executed in a virtualized manner.

3.1 Single Dynamic, Virtualized Runtime

In our proposed system architecture, the underlying system that virtualized protocols make calls to is a dynamic, virtualized runtime for fluidic automation. This runtime is similar to the Puddle runtime [21] with some example supported fluidic operations shown in Figure 5. Fluidic operations may be dynamically called and lazily executed as described in Section 2.2. Since operations are lazily executed, a flush operation forces execution and materializes given fluid IDs.

Fluidic I/O operations optionally have the location of the fluidic input or output, a message, or a prompt. If a message is provided, the system will wait until an administrator confirms that a fluidic input is placed at the specified location or that a fluidic output has been removed from its location. If a prompt is provided, the system will display the prompt and wait for an administrator to confirm and also specify the location to use for the input or output. If no message or prompt is

[1]Unlike physically addressed systems, virtually addressed systems are already isolated since they cannot refer to hardware locations. Systems that support dynamic control flow trivially support static control flow.

specified, fluidic inputs are assumed to already be at the specified location and fluidic outputs are left at the specified location without waiting for removal.

A single runtime may interface with multiple backends. For example, while our underlying runtime implementation targets pipetting robots through the Opentrons Python library, Puddle targets digital microfluidic devices [21]. These backends are responsible for the eventual execution of commands on physical machine(s).

Fluidic I/O
`In`(volume, location, message, prompt) $\rightarrow f$
`Out`(volume, location, message, prompt) $\rightarrow f$

Fluidic Handling
`Mix`(f_1, f_2) $\rightarrow f$
`Take`(f, volume) $\rightarrow (f_t, f_r)$

Hardware Control
`WithTemp`(f, temperature) $\rightarrow f'$
`WithoutTemp`(f) $\rightarrow f'$

Sensing and Flushing
`Temp`(f) $\rightarrow$ temperature of f
`Flush`($f_1, f_2, ...$)

Fig. 5. Example fluidic operations supported by a dynamic, virtualized runtime that virtualized protocols may make calls to. fs are fluid IDs, virtually representing fluid at an arbitrary location. Optional arguments are underlined.

3.2 Shims for Physical APIs

Shim libraries present a physically-addressed API for guest protocols and translate incoming calls into virtually-addressed calls to the underlying runtime in Section 3.1. While we develop a single shim for the popular Opentrons API, APIs for other pipetting robots, such as the pyhamilton Python API for the Hamilton STAR, could also have shims developed. A shim for the pyhamilton library would allow pyhamilton programs as guest protocols for virtualized execution.

Physical-to-virtual mapping. During the execution of a guest protocol, the shim maintains a one-to-one mapping between physical locations referenced by the guest protocol and virtual fluid IDs. Each physical location is associated with a distinct fluid ID. These locations include wells, tubes, and—importantly—pipettes. Each pipette that the protocol references can virtually store fluid at any point in execution and we keep track of the fluid stored in each pipette with a fluid ID.

Manifests (described further in Section 3.3) are what actually initialize this location-to-fluid-ID mapping. When a user executes a wrapped protocol, they must provide a mapping from each named input reagent to a fluid ID as described in Section 3.4. The name-to-fluid-ID mapping (named input arguments) and the name-to-location mapping (manifest) together initialize the location-to-fluid-ID mapping at the start of virtualized execution.

Intercepted API calls. Using the location-to-fluid-ID mapping, a shim can intercept commands from the physical API and submit fluidic operations to the dynamic, virtualized runtime (described in Section 3.1) for physical execution.

Examples. An example shim for intercepting Opentrons API calls to `aspirate` is shown in Figure 6. Example shims for other functions in the Opentrons API are shown in Figure 7.

3.3 Manifests for Physical Protocols

To motivate the need for manifests for specifying protocol inputs and outputs, consider the example protocol in Figure 1b that uses physical addressing of fluids. The protocol performs a simple mixing procedure. The two input reagents are assumed to be in the locations `plates[1]['A1']` and

```
def aspirate(pipette, volume, location):
    # First, we (1) take the fluid we want to aspirate and
    # (2) update `LOC_TO_ID[location]` (the fluid ID
    # associated with the location aspirated from) with
    # `remaining` (the remainder of the splitting)
    aspirated, remaining = Take(LOC_TO_ID[location], volume)
    LOC_TO_ID[location] = remaining

    # Then, we mix with the fluid already in the pipette
    # and update the fluid ID associated with the pipette
    # to be the result of the mixing
    LOC_TO_ID[pipette] = Mix(aspirated, LOC_TO_ID[pipette])
```

Fig. 6. Shim for intercepting an aspiration command. `LOC_TO_ID` is the mapping from physical locations to virtual fluid IDs described in Section 3.2.

`plates[1]['B1']`, respectively. The one product will be placed in `plates[2]['A1']` after execution. It cannot be statically known what locations referenced by this protocol hold the input and output fluids.

In general, it is not possible to statically know the input and output locations for a protocol. Some protocols depend on configurations which are programmatically supplied at run time. More complex protocols have intermediate fluids at locations which are used for neither input nor output and instead may be discarded. Since virtualized execution of an existing protocol requires knowledge of the input and output locations in the protocol, existing protocols must first be wrapped with a manifest that actually specifies the locations of input reagents and output products. Notably, a manifest does not specify the locations of *all* the fluids used in a protocol but only the inputs and outputs.

Definition. A protocol manifest is a mapping from named parameters to locations of reagents and products for a given protocol. Manifests are provided by a human annotator who understands the unmodified original implementation of the protocol. In our implementation described in Section 4.1, the manifest is provided with a Python function returning a dictionary mapping from string parameter names to Opentrons well locations.

Example. An example manifest for the protocol in Figure 1b is shown in Figure 8. The reagents stored at `plates[1]["A1"]` and `plates[1]["B1"]` are assigned names `A` and `B` respectively while the product located at `plates[2]["A1"]` is assigned the name `0`. In this example, a static manifest is sufficient but for the PCR prep protocol in Section 4.3, an external configuration is queried at run time to dynamically construct a manifest.

Another example is in Figure 9 which demonstrates the process of creating a manifest for a PCR protocol.

Effort of writing manifests. Manifests must be manually authored by an *annotator*. Since the mappings that manifests contain are basically just the reagents and products of the protocol, the annotator is not necessarily the original protocol author. This is because the reagents and products are typically already separately documented for most protocols. Indeed, in our experience annotating protocols for demonstration in Section 4, knowledge of the implementation of protocols was not necessary for writing manifests and we relied solely on the graphics and text documentation accompanying the protocols on the Opentrons website.

```
aspirate(pip, loc, vol):
  aspirated, remaining = Take(loc_to_id[loc], vol)
  loc_to_id[pip] = Mix(aspirated, loctoid[pip])
  loc_to_id[loc] = remaining
```

(a) Shim for aspiration

```
dispense(pip, loc, vol):
  dispensed, remaining = Take(loc_to_id[pip], vol)
  loc_to_id[loc] = Mix(dispensed, loctoid[loc])
  loc_to_id[pip] = remaining
  if loc in temp_module.locations:
    loc_to_id[loc] =
      WithTemp(loc_to_id[loc], temp_module.temp)
```

(b) Shim for dispension

```
set_temperature(temp_module, temperature):
  for loc in temp_module.locations ∪ loc_to_id:
    loc_to_id[loc] =
      WithTemp(loc_to_id[loc], temperature)
```

(c) Shim for controlling temperature

```
deactivate():
  for loc in temp_module.locations ∪ loc_to_id:
    loc_to_id[loc] =
      WithoutTemp(loc_to_id[loc])
```

(d) Shim for deactivating temperature

```
delay(time):
  for loc, id in loc_to_id:
    Flush(id)
  wait(time)
```

(e) Shim for delaying

Fig. 7. Pseudocode for shims of other fluidic commands.

```
1 Manifest( reagents={ "A": plates[1]["A1"],
2                      "B": plates[1]["B1"] },
3           products={ "O": plates[2]["A1"] })
```

Fig. 8. A manifest for the mixing protocol in Figure 1b

3.4 Dynamic, User-level Code

Finally, existing protocols that have been wrapped with a manifest may be invoked from dynamic code written by end users in high-level languages such as Python. This code may dynamically use the underlying runtime for virtualized fluidic I/O (using the `In` and `Out` operations described in Section 3.1).

```
# Original, unmodified PCR protocol implementation
def run(protocol):
    temp_mod = protocol.load_module('temperature module', 1)
    plates = protocol.loaded_labwares

    # Transfer to a plate on the temperature module
    pip.transfer( vol, plates[0]["A1"], plates[1]["A1"]) )

    # Thermocycle and deactivate                          1
    for iter in range(N_ITER):
        for temp, time in TEMPS_AND_TIMES:
            temp_mod.await_temperature(temp)
            protocol.delay(time)
    temp_mod.deactivate()                                 2

    # Transfer to the result plate
    pip.transfer( vol, plates[1]["A1"], plates[2]["A1"]) )

# Manifest added to annotate the original protocol
# with information about locations of inputs and outputs  3
def manifest(p):
    return Manifest( reagents={ "I": p.loaded_labwares[0]["A1"] },
                     products={ "O": p.loaded_labwares[2]["A1"] })
```

Fig. 9. The process of annotating an existing, unmodified protocol: (1) reading documentation describing the function of the protocol, (2) identifying referenced locations that contain inputs and outputs, and (3) writing a manifest that specifies the input and output locations.

The fluid IDs returned by I/O operations can then be passed as inputs when invoking a wrapped protocol. Both the input and the output of invocation is a mapping from names to fluid IDs. These mappings are based on the manifest wrapped with the original protocol. So in order to invoke a protocol successfully, the input mapping must provide a fluid ID for each named reagent in the manifest.

We believe this represents a more productive abstraction where dynamic code using high-level operations (such as `Mix` and `Take`) can be used in conjunction with existing protocols. As shown in prior work [21], the underlying runtime may also be used exclusively for writing entire new protocols. However, the novelty of this work is that it allows *existing* protocols to be seamlessly executed in a virtualized manner, providing the benefits of concurrency, portability, and composition without rewriting protocols using a different, less widely adopted API.

4 DEMONSTRATION

In this section, we demonstrate two key benefits of virtualizing existing protocols: *(1)* concurrent execution and *(2)* composition of separate protocols.

4.1 Implementation

We base our demonstration on a proof-of-concept implementation that can virtualize existing, unmodified Opentrons protocols. Opentrons protocols are defined in Python modules with a `run` function that makes API calls to load labware, pipettes, hardware modules and to execute commands to aspirate, dispense, set temperature, etc.

To provide a manifest for an existing Opentrons protocol, an annotator must add a Python `manifest` function to the protocol module. This `manifest` function returns a `Manifest` as shown in Figure 8.

In user-level Python code, users may import protocol modules that have been annotated with a `manifest` and pass the module into a `VirtualProtocol`. The `VirtualProtocol` can then be executed using `Run` which returns a Python `Future`. Returned futures can then be executed asynchronously using standard library functions.

For demonstration, we run the protocols described in this section on the Opentrons simulator which asserts correct physical usage based on robot geometry. The simulator also accurately simulates waiting, which dominates the execution time of protocols where concurrency would be useful.

4.2 Concurrent Protocol Execution

In our implementation, each instance of an existing, physically-addressed protocol is run on a separate Python thread with a shim of the Opentrons API redirecting fluidic commands to a shared dynamic, virtualized runtime. The shared runtime lazily executes commands sent from each thread, prioritizing commands that don't require scarce resources. Scarce resources, such as thermocyclers which only support 1 thermal cycle at a time, are prone to resource contention. Protocols must concurrently wait for their desired hardware configuration (e.g., a thermal cycle which is a sequence of specific temperatures and durations to keep the fluid at each temperature) and so protocols requesting the same configuration can be correctly executed in parallel.

Indeed, we found that parallelism can be achieved. The PCR[2] protocol in Figure 10 loads fluid onto an Opentrons Temperature Module and cycles through a sequence of temperature settings waiting for pre-determined times at each setting. When running multiple instances of this protocol virtualized, each instance follows the same thermocycling sequence, waiting for other instances to complete each thermocycling step before all may proceed. Our results in Figure 11 show that this concurrent waiting allows the protocol instances to execute in parallel with a near-linear speedup.

We also found that the maximum amount of parallelism scales linearly with available resources. When using 6-, 10-, 15-, and 24-tube racks for thermocycling PCR, a maximum of 6, 10, 15, and 24 PCRs could be run in parallel respectively. It is notable that each of these tube racks with varying numbers of tubes had a different layout. Despite this, no modifications to the original PCR protocol were needed to support the differences. This demonstrates another benefit of virtualization: portability of existing, unmodified protocols across different labware.

4.3 Composition of Protocols

Many real-world applications consist of smaller protocols that compose together. For example, a two-part PCR prep application created by the developers of Opentrons was originally developed as two separate protocols with separate Python implementations and separate documentation. Each protocol even has a separate configuration, one in CSV and the other in JSON.

In our implementation, each of these two protocols can be wrapped with a manifest that specifies the reagents and products of the original protocol. Since the manifest is specified through a Python function, the CSV and JSON configurations of the protocols are dynamically inspected to generate the correct manifest for the protocol.

[2]PCR = Polymerase Chain Reaction, a common protocol used to duplicate DNA. PCR consists mainly of thermocycling, or repeatedly heating a sample.

```python
# Original, unmodified PCR protocol implementation
def run(protocol):
    temp_mod = protocol.load_module('temperature module', 1)
    plates = protocol.loaded_labwares

    # Transfer to a plate on the temperature module
    pip.transfer( vol, plates[0]["A1"], plates[1]["A1"]) )

    # Thermocycle and deactivate
    for iter in range(N_ITER):
        for temp, time in TEMPS_AND_TIMES:
            temp_mod.await_temperature(temp)
            protocol.delay(time)
    temp_mod.deactivate()

    # Transfer to the result plate
    pip.transfer( vol, plates[1]["A1"], plates[2]["A1"]) )

# Manifest added to annotate the original protocol
# with information about locations of inputs and outputs
def manifest(p):
    return Manifest( reagents={ "I": p.loaded_labwares[0]["A1"] },
                     products={ "O": p.loaded_labwares[2]["A1"] })
```

annotated_pcr.py (a)

```python
# Import Python module containing annotated PCR protocol
import annotated_pcr.py

# Construct lazily executed DAG of concurrent PCR protocol executions
running = [ Run(VirtualProtocol(pcr), { "I": In(1500) })
            for _ in range(24) ]

# Flush to concurrently materialize the PCR results
Flush([ r.result()["O"]
        for r in wait(running).done ])
```

multi_pcr.py (b)

Fig. 10. The PCR protocol implemented in (a) is an existing protocol provided by the developers of the Opentrons OT-2 pipetting robot. The manifest in (a) specifies the location of input fluid and the location of the resulting thermocycled fluid. Together, (a) represents a wrapped protocol that can subsequently be executed concurrently in the calling code (c) which performs 24 concurrent PCRs.

# of PCRs	Serial	Concurrent	Speedup	% of ideal
1	3:51s	4:00s	0.96x	96%
6	23:06s	4:03s	5.70x	95%
10	38:30s	4:01s	9.60x	96%
15	57:45s	4:05s	14.10x	94%
24	1:32:24s	4:11s	22.09x	92%

Fig. 11. Results of running multiple instances of a single PCR thermal cycle of the PCR protocol shown in Figure 10. Expected serial execution time is computed using the execution time of the original non-virtualized PCR (3m 51s). The %s are of the speedup with respect to ideal linear speedup.

Protocol	Fig.	Lines of code		
		Original	Manifest	Calling
Mixing	1b	16	3	15
PCR thermocycling	10	21	3	18
PCR prep: part 1 + 2	13	68 + 88	36 + 38	36

Fig. 12. Number of lines of Python code for existing protocols, manifests they were wrapped with, and code that calls the wrapped protocol. Line counts refer to the implementations including boilerplate (which is excluded from figures).

Once manifests are provided for each protocol, the first protocol may be invoked, producing a fluid ID as output corresponding to a master mix solution. This fluid ID is then directly passed to the second protocol which accepts the master mix solution as well as DNA samples as inputs.

4.4 Ease of Use

To demonstrate that the effort required to develop manifests and to call wrapped protocols is minimal, we counted the lines of code for the simple mixing protocol in Figure 1b, the PCR protocol in Section 4.2, and the two-part PCR prep protocol in Section 4.3. These counts are recorded in Figure 12.

The larger manifests for PCR prep are due to the protocols depending on CSV and JSON configurations that must be inspected to dynamically generate the correct manifest. The code for inspecting configurations in the manifest code is lifted directly from the original protocol code with no changes.

5 LIMITATIONS AND FUTURE WORK

Resource reuse. The shims we describe in Section 3.2 cleanly map physical commands to virtualized operations. However, they can translate simple sequences of `aspirate-dispense` for in-place mixes into long `Mix-Take` sequences. Our naive, proof-of-concept implementation of `Mix` and `Take` allocates new space for every invocation. This can lead to resource exhaustion.

This issue could be mitigated through automatic resource reuse, requiring the dynamic, virtualized runtime to be both substance-aware and volume-aware. Substance-aware reuse might reallocate space as long as it is always used for the same substance. Volume-aware reuse may eliminate operations on empty fluid IDs.

Substance can be tracked by generating a unique substance ID for each input fluid and reusing the substance ID for subsequent `Take` operations. A new substance ID would only be generated for `Mix` or other such operations. An upper-bound on volume (volume cannot be known exactly because fluid may evaporate due to heating during protocol execution) could be tracked as well, initializing based on invocations of `In`.

Additional Requirements. Our implementation translates all `aspirate` and `dispense` commands into `Mix` and `Take` commands. However, different instances of aspiration and dispension commands in original, existing Opentrons protocols may have different additional requirements. For example, after dispensing, it may be desirable to blow out air from a pipette or touch the pipette against the tips of the well to ensure that all fluid is removed from the pipette. When aspirating, it may be desirable to aspirate air after aspirating the fluid to prevent the fluid from sliding out. These additional requirements only apply in some cases depending on the substance being handled and the whole procedure being performed.

```
from protocols import  mm_assembly, mm_dist_and_dna_transfer

# Call mm_assembly
results = Run(VirtualProtocol(mm_assembly), {
    "Buffer":   In(25),  "MgCl":     In(40),
    "dNTPs":    In(90),  "Water":    In(248),
    "primer 1": In(25),  "primer 2": In(25),
}).result()

# Pass both the intermediate result and DNA
# samples as input to mm_dist_and_dna_transfer
num_samples = 9
for i in range(num_samples):
    key, msg = f"dna_{i}", f"DNA for sample #{i}"
    results[key] = In(2, message=msg)

# Call mm_dist_and_dna_transfer
results = Run(
    VirtualProtocol(mm_dist_and_dna_transfer),
    results,
).result().values()

Flush(results)
```

Fig. 13. Dynamic code to run a functional composition of `mm_assembly` (master mix assembly) with `mm_dist_and_dna_transfer` (master mix distribution and combining with DNA)

Simply translating all aspiration and dispension into the same `Mix` and `Take` commands risks loss of specification for more complex protocols. It would be possible to allow users to specify these requirements for more complex fluid handling and use this to dynamically dispatch more complex commands that can intelligently satisfy the pipetting requirement.

Additional Hardware Support. We demonstrate the portability benefit of fluidic program virtualization by running a virtualized PCR protocol with tube racks of different layouts. However, these different tube rack layouts are being used with the same Opentrons OT-2 pipetting robot. We focused on Opentrons because of its wide adoption as a fluidic programming platform.

Our framework could be extended to support different pipetting robots such as the Hamilton STAR or even digital microfluidic devices such as the OpenDrop. The main challenge with virtualization for these devices is extracting a virtually-addressed representation of physically-addressed protocols. For pipetting robots, the techniques developed in this work readily apply. For digital microfluidic devices, a technique must be developed to convert a sequence of electrode actuations into a sequence of virtually-addressed mixing and splitting operations with reasonable accuracy. This technique may have to take into account the specifics of the device and the sensor data that was used to determine which electrodes to actuate.

Optimizations. The ability to virtualize existing physically-addressed protocols opens the door to generally optimizing existing protocols. Some pipetting robots have multi-channel pipettes and these present an opportunity for minimizing the run time of protocol execution by performing a "vectorization" of mixing and splitting. Another optimization is minimization of tip usage. These

optimizations are much more tractable for a virtually-addressed protocol, since it contains higher-level information about the fluidic operations being performed that a physically-addressed protocol.

6 CONCLUSION

In conclusion, we have demonstrated that existing fluidic protocols can be virtualized without modification. Virtualization of existing protocols allows for portability across different labware, pipettes, and hardware modules; concurrent execution that achieves near-linear speedup; and composition of separate protocols.

REFERENCES

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

[2] Mirela Alistar and Urs Gaudenz. 2017. OpenDrop: An Integrated Do-It-Yourself Platform for Personal Use of Biochips. See[1], 45.

[3] 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.

[4] Vaishnavi Ananthanarayanan and William Thies. 2010. Biocoder: A programming language for standardizing and automating biology protocols. Journal of biological engineering 4, 1 (2010), 1–13.

[5] Maxwell Bates, Aaron J Berliner, Joe Lachoff, Paul R Jaschke, and Eli S Groban. 2017. Wet lab accelerator: a web-based application democratizing laboratory automation for synthetic biology. ACS synthetic biology 6, 1 (2017), 167–171.

[6] Emma J Chory, Dana W Gretton, Erika A Debenedictis, and Kevin M Esvelt. 2020. Flexible open-source automation for robotic bioengineering. bioRxiv (2020).

[7] 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.

[8] Vishal Gupta, Jesús Irimia, Iván Pau, and Alfonso Rodríguez-Patón. 2017. BioBlocks: Programming protocols in biology made easier. ACS synthetic biology 6, 7 (2017), 1230–1232.

[9] Peter L Lee and Benjamin N Miles. 2018. Autoprotocol driven robotic cloud lab enables systematic machine learning approaches to designing, optimizing, and discovering novel biological synthesis pathways. In SIMB Annual Meeting 2018. SIMB.

[10] Alvin C Ma, Melissa S McNulty, Tanya L Poshusta, Jarryd M Campbell, Gabriel Martínez-Gálvez, David P Argue, Han B Lee, Mark D Urban, Cassandra E Bullard, Patrick R Blackburn, et al. 2016. FusX: a rapid one-step transcription activator-like effector assembly system for genome science. Human gene therapy 27, 6 (2016), 451–463.

[11] Andre Maia Chagas, Jennifer C Molloy, Lucia L Prieto-Godino, and Tom Baden. 2020. Leveraging open hardware to alleviate the burden of COVID-19 on global health systems. Plos Biology 18, 4 (2020), e3000730.

[12] Opentrons. 2018. OT-2 Pipetting Robot. https://opentrons.com/ot-2/

[13] 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.

[14] 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.

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

[16] William Thies, John Paul Urbanski, Todd Thorsen, and Saman Amarasinghe. 2008. Abstraction layers for scalable microfluidic biocomputing. Natural Computing 7, 2 (2008), 255–275.

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

[18] 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.

[19] Ellis Whitehead, Fabian Rudolf, Hans-Michael Kaltenbach, and Jörg Stelling. 2018. Automated planning enables complex protocols on liquid-handling robots. ACS synthetic biology 7, 3 (2018), 922–932.

[20] Max Willsey, Ashley P Stephenson, Chris Takahashi, Bichlien H Nguyen, Karin Strauss, and Luis Ceze. 2019. Scaling Microfluidics to Complex, Dynamic Protocols. In ICCAD. 1–6.

[21] Max Willsey, Ashley P. Stephenson, Chris Takahashi, Pranav Vaid, Bichlien H. Nguyen, Michal Piszczek, Christine Betts, Sharon Newman, Sarang Joshi, Karin Strauss, and Luis Ceze. 2019. 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 (Providence, RI, USA) (ASPLOS '19). ACM, New York, NY, USA. https://doi.org/10.1145/3297858.3304027