Programmable Microfluidics

Bill Thies

Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology

Stanford University – October 3, 2007
Acknowledgements

Prof. Saman Amarasinghe
Nada Amin
MIT Computer Science and Artificial Intelligence Laboratory

Prof. Todd Thorsen
J.P. Urbanski
David Craig
MIT Hatsopoulos Microfluids Laboratory

Prof. Jeremy Gunawardena
Natalie Andrew
Harvard Medical School

Prof. Mark Johnson
David Potter
Colorado Center for Reproductive Medicine
Microfluidic Chips

- **Idea:** a whole biology lab on a single chip
  - Input/output
  - **Sensors:** pH, glucose, temperature, etc.
  - **Actuators:** mixing, PCR, electrophoresis, cell lysis, etc.

- **Benefits:**
  - Small sample volumes
  - High throughput
  - Geometrical manipulation

- **Applications:**
  - Biochemistry
  - Cell biology
  - Biological computing
Moore’s Law of Microfluidics: Valve Density Doubles Every 4 Months

Source: Fluidigm Corporation (http://www.fluidigm.com/images/mlaw_lg.jpg)
Moore’s Law of Microfluidics: Valve Density Doubles Every 4 Months

Source: Fluidigm Corporation (http://www.fluidigm.com/didIFC.htm)
Current Practice: Expose Gate-Level Details to Users

- Manually map experiment to the valves of the device
  - Using Labview or custom C interface
  - Given a new device, start over and do mapping again
Our Approach: “Write Once, Run Anywhere”

• **Example: Gradient generation**

  ```
  Fluid yellow = input (0);
  Fluid blue = input(1);
  for (int i=0; i<=4; i++) {
    mix(yellow, 1-i/4, blue, i/4);
  }
  ```

• **Hidden from programmer:**
  – Location of fluids
  – Details of mixing, I/O
  – Logic of valve control
  – Timing of chip operations

450 Valve Operations
Our Approach: “Write Once, Run Anywhere”

- Example: Gradient generation

```
Fluid yellow = input(0);
Fluid blue = input(1);
for (int i=0; i<=4; i++) {
    mix(yellow, 1-i/4, blue, i/4);
}
```

- Hidden from programmer:
  - Location of fluids
  - Details of mixing, I/O
  - Logic of valve control
  - Timing of chip operations

```
setValve(0, HIGH);  setValve(1, HIGH);
setValve(2, LOW);   setValve(3, HIGH);
setValve(4, LOW);   setValve(5, LOW);
setValve(6, HIGH);  setValve(7, LOW);
setValve(8, LOW);   setValve(9, HIGH);
setValve(10, LOW);  setValve(11, HIGH);
setValve(12, LOW);  setValve(13, HIGH);
setValve(14, LOW);  setValve(15, HIGH);
setValve(16, LOW);  setValve(17, LOW);
setValve(18, LOW);  setValve(19, LOW);
wait(2000);
setValve(14, HIGH); setValve(2, LOW);
wait(1000);
setValve(4, HIGH);  setValve(12, LOW);
setValve(16, HIGH); setValve(18, HIGH);
setValve(19, LOW);
wait(2000);
```

450 Valve Operations
Our Approach:  
“Write Once, Run Anywhere”

- Example: Gradient generation

```java
Fluid yellow = input(0);
Fluid blue = input(1);
for (int i=0; i<=4; i++) {
    mix(yellow, 1-i/4, blue, i/4);
}
```

- Hidden from programmer:
  - Location of fluids
  - Details of mixing, I/O
  - Logic of valve control
  - Timing of chip operations

```
wait(2000);
setValve(14, HIGH); setValve(2, LOW);
wait(1000);
setValve(4, HIGH); setValve(12, LOW);
setValve(16, HIGH); setValve(18, HIGH);
setValve(19, LOW);
wait(2000);
setValve(0, LOW); setValve(1, LOW);
setValve(2, LOW); setValve(3, HIGH);
setValve(4, LOW); setValve(5, HIGH);
setValve(6, HIGH); setValve(7, LOW);
setValve(8, LOW); setValve(9, HIGH);
setValve(10, HIGH); setValve(11, LOW);
setValve(12, LOW); setValve(13, LOW);
setValve(14, LOW); setValve(15, LOW);
setValve(16, HIGH); setValve(17, LOW);
setValve(18, HIGH); setValve(19, LOW);
setValve(2, LOW); wait(1000);
setValve(4, HIGH); setValve(12, LOW);
setValve(16, HIGH); setValve(18, HIGH);
setValve(19, LOW);
setValve(6, HIGH); setValve(7, LOW);
setValve(10, HIGH); setValve(11, LOW);
setValve(12, LOW); setValve(13, LOW);
setValve(14, LOW); setValve(15, LOW);
setValve(16, HIGH); setValve(17, LOW);
setValve(18, HIGH); setValve(19, LOW);
```
Fluidic Abstraction Layers

Protocol Description Language
- readable code with high-level mixing ops

Fluidic Instruction Set Architecture (ISA)
- primitives for I/O, storage, transport, mixing

Fluidic Hardware Primitives
- valves, multiplexers, mixers, latches

Silicon Analog
C
x86
Pentium III, Pentium IV
transistors, registers, …
Fluidic Abstraction Layers

- **Protocol Description Language**
  - readable code with high-level mixing ops

- **Fluidic Instruction Set Architecture (ISA)**
  - primitives for I/O, storage, transport, mixing

  - **Benefits:**
    - Division of labor
    - Portability
    - Scalability
    - Expressivity

- **Fluidic Hardware Primitives**
  - valves, multiplexers, mixers, latches
Fluidic Abstraction Layers

Protocol Description Language
- readable code with high-level mixing ops

Fluidic Instruction Set Architecture (ISA)
- primitives for I/O, storage, transport, mixing

Benefits:
- Division of labor
- Portability
- Scalability
- Expressivity

Fluidic Hardware Primitives
- valves, multiplexers, mixers, latches
Primitive 1: A Valve (Quake et al.)
Primitive 1: A Valve (Quake et al.)

Control Layer

0. Start with mask of channels

Flow Layer
Primitive 1: A Valve (Quake et al.)

1. Deposit pattern on silicon wafer
Primitive 1: A Valve (Quake et al.)

2. Pour PDMS over mold
   - polydimethylsiloxane: “soft lithography”

Control Layer

Thick layer (poured)

Thin layer (spin-coated)

Flow Layer
Primitive 1: A Valve (Quake et al.)

3. Bake at 80° C (primary cure), then release PDMS from mold
Primitive 1: A Valve (Quake et al.)

Control Layer

4a. Punch hole in control channel
4b. Attach flow layer to glass slide

Flow Layer
Primitive 1: A Valve (Quake et al.)

5. Align flow layer over control layer
Primitive 1: A Valve (Quake et al.)

6. Bake at 80° C (secondary cure)
7. When pressure is high, control channel pinches flow channel to form a valve.

**Primitive 1: A Valve (Quake et al.)**

**Control Layer**

**Flow Layer**
Primitive 2: A Multiplexerer (Thorsen et al.)

![Diagram of a multiplexer with inputs and outputs]

- **Input**: Bit 2 0 1, Bit 1 0 1, Bit 0 0 1
- **Outputs**: Output 7, Output 6, Output 5, Output 4, Output 3, Output 2, Output 1, Output 0

Legend:
- Blue: flow layer
- Red: control layer
Primitive 2: A Multiplexer (Thorsen et al.)

Example: select 3 = 011
Primitive 2: A Multiplexer (Thorsen et al.)

Example: select 3 = 011

<table>
<thead>
<tr>
<th>Bit 2</th>
<th>Bit 1</th>
<th>Bit 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Output 0
Output 1
Output 2
Output 3
Output 4
Output 5
Output 6
Output 7

flow layer
control layer
Primitive 2: A Multiplexer (Thorsen et al.)

Example: select 3 = 011
Primitive 3: A Mixer (Quake et al.)

1. Load sample on top
Primitive 3: A Mixer (Quake et al.)

1. Load sample on top
2. Load sample on bottom
Primitive 3: A Mixer (Quake et al.)

1. Load sample on top
2. Load sample on bottom
3. Peristaltic pumping

*Rotary Mixing*
Primitive 4: A Latch (Our contribution)

• Purpose: align sample with specific location on device
  – Examples: end of storage cell, end of mixer, middle of sensor

• Latches are implemented as a partially closed valve
  – Background flow passes freely
  – Aqueous samples are caught
Primitive 5: Cell Trap

• Several methods for confining cells in microfluidic chips
  – U-shaped weirs
  – C-shaped rings / microseives
  – Holographic optical traps
  – Dialectrophoresis

• In our chips: U-Shaped Microseives in PDMS Chambers

Primitive 6: Imaging and Detection

- As PDMS chips are translucent, contents can be imaged directly
  - Fluorescence, color, opacity, etc.

- Feedback can be used to drive the experiment
Fluidic Abstraction Layers

Protocol Description Language
- readable code with high-level mixing ops

Fluidic Instruction Set Architecture (ISA)
- primitives for I/O, storage, transport, mixing

Fluidic Hardware Primitives
- valves, multiplexers, mixers, latches
Toward “General Purpose” Microfluidic Chips
Abstraction 1: Digital Architecture

- Recent techniques can control independent samples
  - Droplet-based samples [Fair et al.]
  - Continuous-flow samples [Our contribution]
  - Microfluidic latches [Our contribution]

- In abstract machine, all samples have unit volume
  - Input/output a sample
  - Store a sample
  - Operate on a sample

- Challenge for a digital architecture: fluid loss
  - No chip is perfect – will lose some volume over time
  - Causes: imprecise valves, adhesion to channels, evaporation, ...
  - How to maintain digital abstraction?
Maintaining a Digital Abstraction

Electronics
- Soft error handling?
- Randomized gates [Palem]
- Replenish charge (GAIN)
- Loss of charge

High-Level Language

Instruction Set Architecture (ISA)

Hardware

Microfluidics
- Expose loss in language
  - User deals with it
- Expose loss in ISA
  - Compiler deals with it
- Replenish fluids?
  - Maybe (e.g., with water)
  - But may affect chemistry
- Loss of fluids
Abstraction 2: Mix Instruction

• Microfluidic chips have various mixing technologies
  – Electrokinetic mixing [Levitan et al.]
  – Droplet mixing [Fair et al.]
  – Rotary mixing [Quake et al.]

• Common attributes:
  – Ability to mix two samples in equal proportions, store result

• Fluidic ISA: \texttt{mix} (int \texttt{src}_1, int \texttt{src}_2, int \texttt{dst})
  – Ex: \texttt{mix}(1, 2, 3)

<table>
<thead>
<tr>
<th>Storage Cells</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
</tbody>
</table>

![Mixer Diagram]

– To allow for lossy transport, only 1 unit of mixture retained
Gradient Generation in Fluidic ISA
Gradient Generation in Fluidic ISA

**Direct Control**
- 450 valve actuations
- only works on 1 chip

**Fluidic ISA**
- 15 instructions
- portable across chips

```
wait(2000);
setValve(14, HIGH); setValve(2, LOW);
wait(1000);
setValve(4, HIGH);  setValve(12, LOW);
setValve(16, HIGH); setValve(18, HIGH);
setValve(19, LOW);
wait(2000);
setValve(0, LOW);  setValve(1, LOW);
setValve(2, LOW);  setValve(3, HIGH);
setValve(4, LOW);  setValve(5, HIGH);
setValve(6, HIGH); setValve(7, LOW);
setValve(8, LOW);  setValve(9, HIGH);
setValve(10, HIGH); setValve(11, LOW);
setValve(12, LOW); setValve(13, LOW);
setValve(14, LOW); setValve(15, HIGH);
setValve(16, HIGH); setValve(17, LOW);
setValve(18, HIGH); setValve(19, LOW);
```
Implementation: Oil-Driven Chip

<table>
<thead>
<tr>
<th></th>
<th>Inputs</th>
<th>Storage Cells</th>
<th>Background Phase</th>
<th>Wash Phase</th>
<th>Mixing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chip 1</td>
<td>2</td>
<td>8</td>
<td>Oil</td>
<td>—</td>
<td>Rotary</td>
</tr>
</tbody>
</table>

Flow Layer  Control Layer
Implementation: Oil-Driven Chip

\texttt{mix (S_1, S_2, D) \{ 
1. Load S_1 
2. Load S_2 
3. Rotary mixing 
4. Store into D 
\}}

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Storage Cells</th>
<th>Background Phase</th>
<th>Wash Phase</th>
<th>Mixing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chip 1</td>
<td>2</td>
<td>8</td>
<td>Oil</td>
<td>—</td>
</tr>
</tbody>
</table>
Implementation 2: Air-Driven Chip

<table>
<thead>
<tr>
<th></th>
<th>Inputs</th>
<th>Storage Cells</th>
<th>Background Phase</th>
<th>Wash Phase</th>
<th>Mixing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chip 1</td>
<td>2</td>
<td>8</td>
<td>Oil</td>
<td>—</td>
<td>Rotary</td>
</tr>
<tr>
<td>Chip 2</td>
<td>4</td>
<td>32</td>
<td>Air</td>
<td>Water</td>
<td>In channels</td>
</tr>
</tbody>
</table>
Implementation 2: Air-Driven Chip

\[
\text{mix} \ (S_1, S_2, D) \{
\begin{align*}
1. & \text{ Load } S_1 \\
2. & \text{ Load } S_2 \\
3. & \text{ Mix / Store into } D \\
4. & \text{ Wash } S_1 \\
5. & \text{ Wash } S_2
\end{align*}
\}
\]

<table>
<thead>
<tr>
<th></th>
<th>Inputs</th>
<th>Storage Cells</th>
<th>Background Phase</th>
<th>Wash Phase</th>
<th>Mixing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chip 1</td>
<td>2</td>
<td>8</td>
<td>Oil</td>
<td>—</td>
<td>Rotary</td>
</tr>
<tr>
<td>Chip 2</td>
<td>4</td>
<td>32</td>
<td>Air</td>
<td>Water</td>
<td>In channels</td>
</tr>
</tbody>
</table>
Fluidic Abstraction Layers

Protocol Description Language
- readable code with high-level mixing ops

Fluidic Instruction Set Architecture (ISA)
- primitives for I/O, storage, transport, mixing

Fluidic Hardware Primitives
- valves, multiplexers, mixers, latches

chip 1

chip 2

chip 3
Abstraction 1: Managing Fluid Storage

- **Programmer uses location-independent Fluid variables**
  - Runtime system assigns & tracks location of each Fluid
  - Comparable to automatic memory management (e.g., Java)
Abstraction 2: Fluid Re-Generation

Programmer may use a Fluid variable multiple times
- Each time, a physical Fluid is consumed on-chip
- Runtime system re-generates Fluids from computation history
Custom Re-Generation

- Some species cannot be regenerated by repeating history
  - e.g., if selective mutagenesis has evolved unique sequence
- Users can extend Fluid class, specify how to regenerate
  - e.g., run PCR to amplify sequence of interest

```java
class DNASample extends Fluid {

    // Return array of fluids that are equivalent to this fluid
    Fluid[] regenerate() {
        Fluid amplified = performPCR(this, cycles, primer1, primer2, ...);
        Fluid[] diluted = dilute(amplified, Math.pow(2, cycles));
        return diluted;
    }

    // Return minimum quantity of this fluid needed to generate others
    int minQuantity() {
        return 1;
    }
}
```
Unique Fluids Prohibit Re-Generation

- Some Fluids may be unique, with no way to amplify
  - E.g., products of cell lysis

- Users can express this constraint using a UniqueFluid:

```java
class UniqueFluid extends Fluid {
    Fluid[] regenerate() {
        throw new EmptyFluidException();
    }
}

UniqueFluid f = lysisProduct();
UniqueFluid[] diluted = dilute(f);
for (int i=0; i<diluted.length; i++) {
    analyze(diluted[i]);
}
```

- Can compiler verify that unique fluids used only once?
  - Unique (linear) types is a rich research area in prog. languages
    [Wadler] [Hogg] [Baker] [Minsky] [Boyland] [Fahndrich & DeLine]
  - But solutions often require annotations & do not handle arrays
  - Practical approach: verify in simple cases, warn about others

→ Opportunity for programming language research
Abstraction 3: Arbitrary Mixing

- Allows mixing fluids in any proportion, not just 50/50
  - Fluid \texttt{mix} (Fluid \( F_1 \), float \( p_1 \), Fluid \( f_2 \), float \( F_2 \))
    - Returns Fluid that is \( p_1 \) parts \( F_1 \) and \( p_2 \) parts \( F_2 \)
  - Runtime system translates to 50/50 mixes in Fluidic ISA
  - Note: some mixtures only reachable within error tolerance \( \varepsilon \)

Fluid[] out = new Fluid[8];
Fluid yellow = input(0);
Fluid blue = input(1);
Fluid green = mix(yellow, blue);

out[0] = yellow;
out[1] = mix(yellow, green);
out[2] = green;
out[3] = mix(blue, green);
out[4] = blue;

Fluid[] out = new Fluid[8];
Fluid yellow = input(0);
Fluid blue = input(1);
Fluid green = mix(yellow, blue);

out[0] = yellow;
out[1] = mix(yellow, 3/4, blue, 1/4);
out[2] = mix(yellow, 1/2, blue, 1/2);
out[3] = mix(yellow, 1/4, blue, 3/4);
out[4] = blue;
Abstraction 3: Arbitrary Mixing

3. Arbitrary Mixing

```java
Fluid[] out = new Fluid[8];
Fluid yellow = input (0);
Fluid blue = input (1);

out[0] = yellow;
out[1] = mix(yellow, 3/4, blue, 1/4);
out[2] = mix(yellow, 1/2, blue, 1/2);
out[3] = mix(yellow, 1/4, blue, 3/4);
out[4] = blue;
```

4. Parameterized Mixing

```java
Fluid[] out = new Fluid[8];
Fluid yellow = input (0);
Fluid blue = input (1);

for (int i=0; i<=4; i++) {
    out[i] = mix(yellow, 1-i/4, blue, i/4);
}
```

- Allows mixing fluids in any proportion, not just 50/50
  - Fluid `mix` (Fluid $F_1$, float $p_1$, Fluid $F_2$, float $F_2$)
    - Returns Fluid that is $p_1$ parts $F_1$ and $p_2$ parts $F_2$
  - Runtime system translates to 50/50 mixes in Fluidic ISA
  - Note: some mixtures only reachable within error tolerance $\varepsilon$
Abstraction 4: Cell Traps

• Unlike fluids, cells adhere to a specific location on chip
  – To interact with cells, need to move Fluids to their location

• CellTrap abstraction establishes a fixed chamber on chip
  – Fundamental capability: fill with a given fluid (incl. cell culture)

```java
class CellTrap {

  // establish a new, empty location on chip
  CellTrap();

  // replace contents of cell trap with new fluid; return old contents
  UniqueFluid drainAndRefill(Fluid newContents);

  // regenerate contents of cell trap; return drained fluid as needed
  Fluid drainAndRegenerate();
}
```
Abstraction 4: Cell Traps

```java
CellTrap celltrap = new CellTrap(); // setup cell culture
for (int i=0; i<N; i++)
    celltrap.drainAndRefill(cellCulture);

celltrap.drainAndRefill(distilledWater); // analyze cell metabolites
Fluid metabolites = drainAndRegenerate();
analyzeWithIndicators(metabolites);

celltrap.drainAndRefill(antibodyStain); // stain cells for imaging
```

→ Must schedule all uses of metabolites before staining
  – Otherwise, runtime error
  – Like unique variables, difficult to verify safety in general case
  – But thanks to language, compiler can give useful warnings
Abstraction 5: Timing Constraints

- Precise timing is critical for many biology protocols
  - Minimum delay: cell growth, enzyme digest, denaturing, etc.
  - Maximum delay: avoid precipitation, photobleaching, etc.
  - Exact delay: regular measurements, synchronized steps, etc.

- Simple API for indicating timing constraints:
  - fluid.useBetween(N, M) — celltrap.useBetween(N, M)
  → Schedule next use of a Fluid (or drain of a CellTrap) between N and M seconds from time of the call
  → Also becomes part of Fluid’s regeneration history

- Note: may require parallel execution
  - Fluid f1 = mix(…); f1.useBetween(10, 10);
  - Fluid f2 = mix(…); f2.useBetween(10, 10);
  - Fluid f3 = mix(f1, f2);
Scheduling the Execution

• Scheduling problem has two parts:
  1. Given dependence graph, find a good schedule
  2. Extract dependence graph from the program
1. Finding a Schedule

Abstract scheduling problem:
- Given task graph $G = (V, E)$ with $[\text{min}, \text{max}]$ latency per edge
- Find shortest schedule $(V \rightarrow Z)$ respecting latency on each edge

→ Case 1: Unbounded parallelism
  - Can express as system of linear difference constraints
  - Solve optimally in polynomial time

→ Case 2: Limited parallelism
  - Adds constraint: only $k$ vertices can be scheduled at once
  - Can be shown to be NP-hard (reduce from PARTITION)
  - Rely on greedy heuristics for now
2. Extracting Dependence Graph

• Static analysis difficult due to aliasing, etc.
  – Requires extracting precise producer-consumer relationships

• Opportunity:
  Perform scheduling at runtime, using lazy evaluation
  – Microfluidic operations are slow → computer can run ahead
  – Build dependence graph of all operations up to decision point

• Hazard: constraints that span decision points
  – Dynamic analysis cannot look into upcoming control flow
  – We currently prohibit such constraints – leave as open problem
BioStream Protocol Language

- Implements the abstractions
  - Full support for storage management, fluid re-generation, arbitrary mixing
  - Partial support for cells, timing

- Implemented as a Java library
  - Allows flexible integration with general-purpose Java code

- Targets microfluidic chips or auto-generated simulator

```
Fluid yellow = input (0);
Fluid blue = input (1);
Fluid[] out = new Fluid[8];
for (int i=0; i<=4; i++)
  out[i] = mix(yellow, 1-i/4, blue, i/4);
```
Applications in Progress

1. What are the best indicators for oocyte viability?
   - With Mark Johnson’s and Todd Thorsen’s groups
   - During in-vitro fertilization, monitor cell metabolites and select healthiest embryo for implantation

2. How do mammalian signal transduction pathways respond to complex inputs?
   - With Jeremy Gunawardena’s and Todd Thorsen’s groups
   - Isolate cells and stimulate with square wave, sine wave, etc.
Generating Complex Signals

CellTrap cells = new CellTrap();
… // setup cell culture
while (true) {
    float target = targetSignal(getTime());
    Fluid f = mix(EGF, target,
                  WATER, 1-target);
    cells.drainAndFill(f);
    cells.useAfter(10*SEC);
}
Additional Applications

• Killer apps: react to feedback, redirect the experiment
  – Recursive-descent search
  – Fixed-pH reaction
  – Directed evolution
  – Long, complex protocols

• Application to biological computation
  – Many emerging technologies:
    DNA computing, cellular signaling, biomolecular automata, …
  – But not yet able to assemble, sustain, and adapt themselves
  – Microfluidics provides a scaffold to explore underlying biology
Compiler Optimizations
Algorithms for Efficient Mixing

• Mixing is fundamental operation of microfluidics
  – Prepare samples for analysis
  – Dilute concentrated substances
  – Control reagent volumes

  Analogous to ALU operations on microprocessors

• How to synthesize complex mixture using simple steps?
  – Many systems support only 50/50 mixers
  – Should minimize number of mixes, reagent usage
  – Note: some mixtures only reachable within error tolerance $\epsilon$

  Interesting scheduling and optimization problem
Why Not Binary Search?

5 inputs, 4 mixes
Why Not Binary Search?

4 inputs, 3 mixes

5 inputs, 4 mixes
Min-Mix Algorithm

• Simple algorithm yields minimal number of mixes
  – For any number of reagents, to any reachable concentration
  – Also minimizes reagent usage on certain chips
Min-Mix Algorithm: Key Insights

1. The mixing process can be represented by a tree.

```
      A
     / \  \
    /   \ \
   B     B
```

5/8 A, 3/8 B
Min-Mix Algorithm: Key Insights

1. The mixing process can be represented by a tree.

<table>
<thead>
<tr>
<th>d</th>
<th>$2^{-d}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1/8</td>
</tr>
<tr>
<td>2</td>
<td>1/4</td>
</tr>
<tr>
<td>1</td>
<td>1/2</td>
</tr>
</tbody>
</table>

2. The contribution of an input sample to the overall mixture is $2^{-d}$, where $d$ is the depth of the sample in the tree.

$5/8$ A, $3/8$ B
Min-Mix Algorithm: Key Insights

1. The mixing process can be represented by a tree.

<table>
<thead>
<tr>
<th>d</th>
<th>$2^{-d}$</th>
<th>1/2</th>
<th>1/4</th>
<th>1/8</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

![Diagram showing mixing process with nodes A and B and binary representation of 5 and 3]

2. The contribution of an input sample to the overall mixture is $2^{-d}$, where $d$ is the depth of the sample in the tree.

3. In the optimal mixing tree, a reagent appears at depths corresponding to the binary representation of its overall concentration.

5/8 A, 3/8 B
Min-Mix Algorithm

- **Example:** mix 5/16 A, 7/16 B, 4/16 C
  
<table>
<thead>
<tr>
<th>d</th>
<th>2^{-d}</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>1/16</td>
</tr>
<tr>
<td>3</td>
<td>1/8</td>
</tr>
<tr>
<td>2</td>
<td>1/4</td>
</tr>
<tr>
<td>1</td>
<td>1/2</td>
</tr>
</tbody>
</table>

![Diagram of A, B, C nodes connected in a tree structure]

- **In paper:** pseudocode, proof of correctness / optimality
Work In Progress
CAD Tools for Microfluidic Chips

- Microfluidic design tools are in their infancy
  - Most groups use Adobe Illustrator or AutoCAD
  - Limited automation; every line drawn by hand

- Due to fast fabrication, redesign is very frequent
  - Student can do multiple design cycles per week
First Step: Automatic Routing

- **First target:** automate the routing of control channels
  - Connecting valves to pneumatic ports is very tedious
  - Simple constraints govern the channel placement

- **AutoCAD plugin automates this task**
  - Developed with Nada Amin
Related Work

• Aquacore – builds on our work, ISA + architecture [Amin et al.]

• Automatic generation / scheduling of biology protocols
  – Robot scientist: generates-tests genetic hypotheses [King et al.]
  – EDNAC computer for automatically solving 3-SAT [Johnson]
  – Compile SAT to microfluidic chips [Landweber et al.] [van Noort]
  – Mapping sequence graphs to grid-based chips [Su/Chakrabarty]

• Custom microfluidic chips for biological computation
  – DNA computing [Grover & Mathies] [van Noort et al.] [McCaskill] [Livstone, Weiss, & Landweber] [Gehani & Reif] [Farfel & Stefanovic]
  – Self-assembly [Somei, Kaneda, Fujii, & Murata] [Whitesides et al.]

• General-purpose microfluidic chips
  – Using electrowetting, with flexible mixing [Fair et al.]
  – Using dialectrophoresis, with retargettable GUI [Gascoyne et al.]
  – Using Braille displays as programmable actuators [Gu et al.]
Conclusions

• Abstraction layers for programmable microfluidics
  – General-purpose chips
  – Fluidic ISA
  – BioStream language
  – Mixing algorithms

• Vision for microfluidics: everyone uses standard chip

• Vision for software: a defacto language for experimental science
  – Download a colleague’s code, run it on your chip
  – Compose modules and libraries to enable complex experiments that are impossible to perform today

http://cag.csail.mit.edu/biostream
Extra Slides
How Can Computer Scientists Contribute?

• Applying the ideas from our field to a new domain
  – Sometimes requires deep adaptations (e.g., digital gain)

• Our contributions:
  – First soft-lithography digital architecture with sample alignment
  – First demonstration of portability: same code, multiple chips
  – New high-level programming abstractions for microfluidics
  – First $O(\lg n)$ mixing algorithm for lossy unit volumes (vs $O(n)$)

• Open problems:
  – Adapt unique (linear) types to microfluidics
  – Sound scheduling under timing constraints
  – Dynamic optimization of slow co-processors (lazy vectorization?)
  – Mixing algorithms for different ISA’s (e.g., lossless mixing)
  – Generate a CAD layout from a problem description