Computer Science and Artificial Intelligence Laboratory

Technical Report

MIT-CSAIL-TR-2006-034 May 5, 2006

Abstraction Layers for Scalable Microfluidic Biocomputers (Extended Version)

William Thies, John Paul Urbanski, Todd Thorsen, and Saman Amarasinghe

massachusetts institute of technology, cambridge, ma 02139 usa — www.csail.mit.edu

Abstraction Layers for Scalable Microfluidic Biocomputers (Extended Version*)

William Thies†, John Paul Urbanski§, Todd Thorsen§, and Saman Amarasinghe†

†MIT Computer Science and Artificial Intelligence Laboratory §MIT Hatsopoulos Microfluids Laboratory

{thies, urbanski, thorsen, saman}@mit.edu

Abstract

Microfluidic devices are emerging as an attractive technology for automatically orchestrating the reactions needed in a biological computer. Thousands of microfluidic primitives have already been integrated on a single chip, and recent trends indicate that the hardware complexity is increasing at rates comparable to Moore's Law. As in the case of silicon, it will be critical to develop abstraction layers—such as programming languages and Instruction Set Architectures (ISAs)—that decouple software development from changes in the underlying device technology.

Towards this end, this paper presents BioStream, a portable language for describing biology protocols, and the Fluidic ISA, a stable interface for microfluidic chip designers. A novel algorithm translates microfluidic mixing operations from the BioStream layer to the Fluidic ISA. To demonstrate the benefits of these abstraction layers, we build two microfluidic chips that can both execute BioStream code despite significant differences at the device level. We consider this to be an important step towards building scalable biocomputers.

1 Introduction

One of the challenges in biological computing is that the laboratory protocols needed to carry out a computation can be very time consuming. For example, a 20-variable 3-SAT problem required 96 hours to complete [1], not counting the considerable time needed for setup and evaluation. To automate and optimize this process, researchers have turned to microfluidic devices [2–10]. Microfluidics offers the promise of a "lab on a chip" system that can individually control picoliter-scale quantities of fluids, with integrated support for operations such as mixing, storage, PCR, heating/cooling, cell lysis, electrophoresis, and others [11–13]. Apart from being amenable to computer control, microfluidics drastically reduces the volumes of samples, thereby reducing costs and improving capture kinetics. Using microfluidics, DNA hybridization times can be reduced from 24 hours to 4 minutes [10] and the number of bases needed to encode information can be decreased from 15 bases per bit to 1 base per bit [1, 8].

Thus has emerged a vision for creating a hybrid DNA computer: one that uses microfluidics for the plumbing (the control paths) and biological primitives for the computations (the ALUs). On the hardware side, this vision is becoming scalable: microfluidic chips have integrated up to 3,574 valves with 1,000 individually-addressable storage chambers [14]. Moreover, recent trends indicate that microfluidics is following a path similar to Moore's law, with the number of soft-lithography valves per unit area doubling every 4.5 months [15].

On the software side, however, the microfluidic realm is lagging far behind its silicon counterpart. For silicon computers, the complexity and scale of the underlying hardware is masked by a set of well-defined abstraction layers. For example, transistors are organized into gates, which combine to form functional units, which together can implement an Instruction Set Architecture (ISA). The user operates at an even higher level of abstraction (e.g., C++), which is automatically translated into the ISA. These abstraction layers have proven critical for managing complexity. Without them, the computing field would have stagnated as every researcher tried to gain a transistor-level understanding of his machine.

Unfortunately, the current practice in experimental microfluidics is to expose all of the hardware resources directly to the experimentalist. Using a graphical system such as Labview, the user orchestrates the

*This is an extended version of a paper [16] that appeared in the 12th International Meeting on DNA Computing, June, 2006.

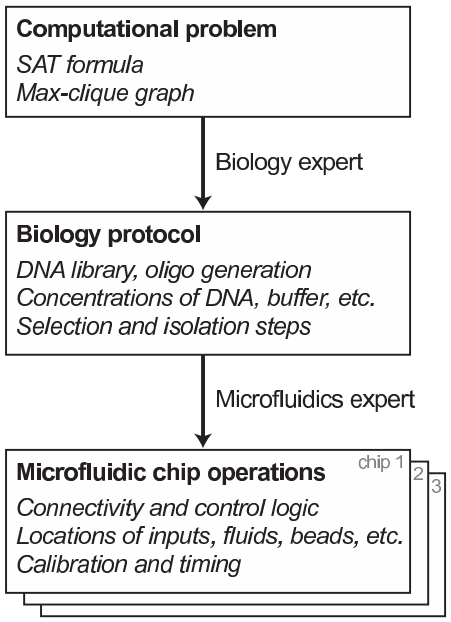

Figure 1: Abstraction layers for DNA computing.

individual behavior of each valve in the microfluidic device. While this practice is merely tedious for today's devices, it will soon become completely intractable—akin to programming a modern microprocessor by directly toggling each of a million gates.

In this paper, we present a system and methodology that uses new abstraction layers for scalable biological computing. As illustrated in Figure 1, our system consists of three layers. At the highest level, the programmer indicates the abstract computation to be performed—for example, in the form of a SAT formula. With some expertise in DNA computing and experimental biology, the computation can be transformed to the next layer: a portable biological protocol for performing the computation. The protocol is portable in that it does not depend on the physical implementation of the protocol; for example, it specifies fluid concentrations but not fluid volumes. Finally, the bottom layer specifies the operations needed to execute the protocol on a specific microfluidic chip. Each microfluidic chip designer provides a library that translates an abstract protocol into the specific sequence of valve actuations needed to execute that protocol on a specific chip.

These abstraction layers provide many benefits. Primarily, by using an architecture-independent description of the biological protocol (the middle layer), the application development can be decoupled from advances in the underlying device technology. Thus, as microfluidic devices come to support additional inputs, mixers, storage cells, etc., the existing suite of protocols can run without modification (much as C programs run without modification on successive generations of microprocessors). In addition, the protocol layer serves as a division of labor. Rather than requiring a heroic and brittle translation from a SAT formula directly to a microfluidic chip, a biologist provides a mapping to the abstract protocol while a microfluidics expert maps the protocol to the underlying device. The abstract protocol is also perfectly suited to simulation, thereby allowing the logical operations to be verified without relying on any physical implementation. Further, a portable protocol description could serve the role of pseudocode in technical publications, providing a precise account of the experimental methods used. Third-party protocols could be downloaded and executed (or called as sub-routines) on one's own microfluidic device.

In the long term, the protocol description language will support all of the operations needed for biological computing. However, as there does not yet exist a single microfluidic device that can encompass all the functionality (preparation of DNA libraries, selection, readout, etc.), this paper focuses on three fun-

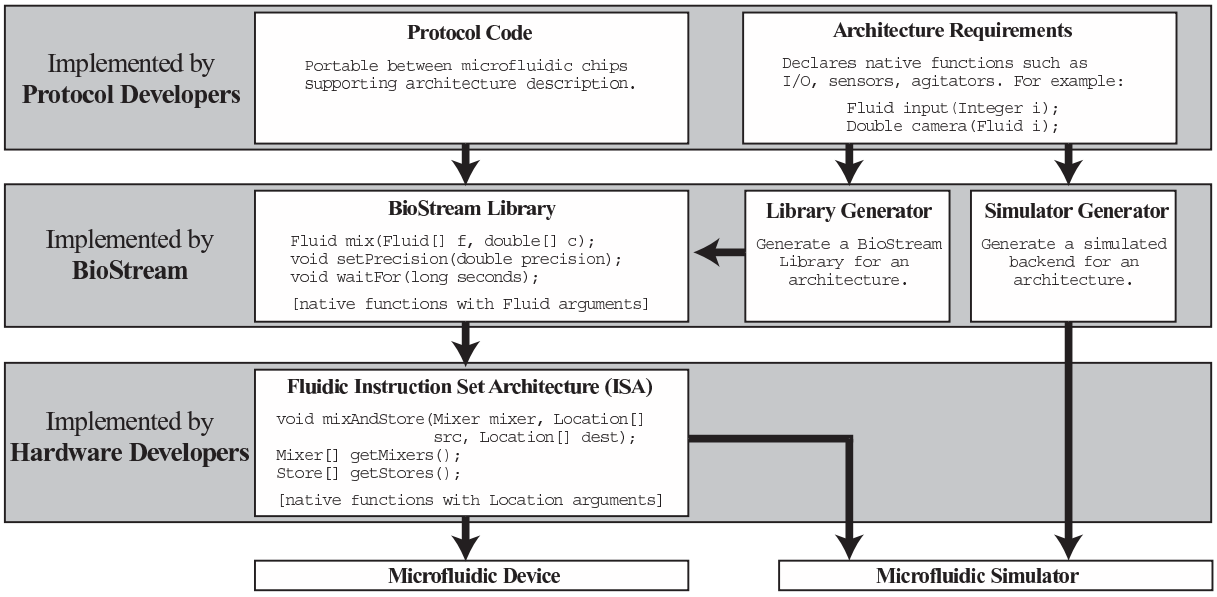

Figure 2: Abstraction layers in the BioStream system.

damental primitives: fluid mixing, fluid transport, and fluid storage. We describe a programming system called BioStream (Section 2) that provides an architecture-independent interface for these operations. To show that BioStream is portable, we execute BioStream code on two fundamentally different microfluidic architectures (Section 3). We also present a novel algorithm for mixing fluids to a given concentration using the minimal number of simple on-chip mixing steps (Section 4). Our system represents a fully-functional, end-to-end demonstration of portable software on microfluidic hardware.

2 BioStream Protocol Language

We have developed a software system called BioStream for portable microfluidics protocols. BioStream is a Java library that virtualizes many aspects of the underlying hardware resources. While BioStream can be targeted by a compiler (for example, a DNA computing compiler that converts a mathematical problem into a biological protocol), it is also suitable for direct programming and experimentation by biologists. As such, the language provides several high-level abstractions to improve readability and programmer productivity.

2.1 Providing Portability

As shown in Figure 2, BioStream offers two levels of abstraction underneath the protocol developer. The first abstraction layer is the BioStream library, which provides first-class Fluid objects that are not associated with a given hardware location. The library also supports a general mix operation for combining Fluids in arbitrary proportions and with adjustable precision. The second abstraction layer, the Fluidic Instruction Set Architecture (ISA), interfaces with the underlying architecture. The fundamental operation is `mixAndStore`, which mixes the contents of several locations on chip and stores the result in a destination location. The Fluidic ISA exports the properties of the mixers and stores available on chip so that the library can properly interface with it.

In addition to the abstractions for mixing, there are some architecture-specific features that need to be made available to the programmer. These "native functions" include I/O devices, sensors, and agitators that might not be supported by every chip, but are needed to execute the program; for example, special input lines, cameras, or heaters. As shown in Figure 2, BioStream supports this functionality by having the programmer declare a set of architecture requirements. BioStream uses the requirements to generate a library which contains the same functionality; it also checks that the architecture target supports all of the required

functions. Finally, BioStream includes a generic simulator that inputs a set of architecture requirements and outputs a virtual machine that emulates the architecture. This allows full protocol development and validation even without hardware resources.

The BioStream system is fully implemented. The reflection capabilities of Java are utilized to automatically generate the library and the simulator from the architecture requirements. As described in Section 3, we also execute the Fluidic ISA on two real microfluidic chips.

2.2 Example Protocol

An example of a BioStream protocol appears in Figure 3. This is a general program that seeks to find the ratio of two reagents that leads to the highest activity in the presence of a given indicator. Experiments of this sort are common in biology. For example, the program could be applied to investigate the roles of cytochrome-c and caspase 8 in activating apoptosis (cell death); cell lysate would serve as the indicator in this experiment [17,18]. The protocol uses feedback from a luminescence detector to guide the search for the highest activity. After sampling some concentrations in the given range, it descends recursively and narrows the range for the next round of sampling. Using self-directed mixing, a high precision can be obtained after only a few rounds.

The recursive descent program declares a `SimpleLibrary` interface (see bottom of Figure 3) describing the functionality required on the target architecture. In this case, a camera is needed to detect luminescence. While we have not mounted a camera on our current device, it would be straightforward to do so.

2.3 Improving Programmer Productivity

A key abstraction provided by BioStream is the use of Fluids as first-class objects in the programming language. The challenge in implementing this functionality is that physical fluids can be used only once, as they are consumed in mixtures and reactions. However, the programmer might reference a Fluid variable multiple times (e.g., variables A and B in the recursive descent example). BioStream supports this behavior by keeping track of how each Fluid was generated and regenerating Fluids that are reused.

The regeneration mechanism works by associating each Fluid object with the name and arguments of the function that created it. The creating function must be a mix operation or a native function, both of which are visible to BioStream (the Fluid constructor is not exposed). BioStream maintains a valid bit for each Fluid, which indicates whether or not the Fluid is stored in a storage chamber on the chip. By default, the bit is true when the Fluid is first created, and it is invalidated when the Fluid is used as an argument to a BioStream function. If a BioStream function is called with an invalid Fluid, that Fluid is regenerated using its history. Note that this regeneration mechanism is fully dynamic (no analysis of the source code is needed) and is accurate even in the presence of pointers and aliasing.

The computation history created for Fluids can be viewed as a dependence tree with several interesting applications. For example, the library can execute a program in a demand-driven fashion by initializing each Fluid to an invalid state and only generating it when it is used by a native function. This lazy evaluation affords the library more flexibility in scheduling the mixing operations when the Fluids are needed. For example, operations could be reordered to minimize storage requirements or to issue parallel operations with vector control. Just-in-time optimizations such as these are especially promising for microfluidic chips, as silicon computers operate much faster than their microfluidic counterparts and have cycles to spare at runtime.

A final abstraction offered by the BioStream interface is the mix operation, which combines a set of fluids in arbitrary proportions. This offers significant gains for programmer productivity, as otherwise mixtures need to be synthesized one step at a time using low-level primitives. Section 4 describes how this abstraction can be supported efficiently by the runtime system.

```
import biostream.library.*;
// Requires following external setup in laboratory:
//
//  - input(0) -- fluid A
//  - input(1) -- fluid B
//  - input(2) -- luminescent activity indicator
//
// This program will recursively zoom in on the mixture
// of A and B that has the highest activity.
public class RecursiveDescent {
   public static void main(String[] args) {
      // args[0]:  the backend to use
      SimpleLibrary library = (SimpleLibrary)
          LibraryFactory.buildLibrary("SimpleLibrary", args[0]);
      run(library);
   }
   private static void run(SimpleLibrary library) {
      // number of rounds and samples per round
      int ROUNDS = 10; int SAMPLES = 5;
      // input fluids
      Fluid A = library.input(new Integer(0));
      Fluid B = library.input(new Integer(1));
      Fluid indicator = library.input(new Integer(2));
      // trial samples and best activity
      Fluid [] samp = new Fluid[SAMPLES];
      double bestActivity = -1;
      // center and radius of current concentration range
      double center = 0.5, radius = 0.5;
      for (int i=0; i<ROUNDS; i++) {
         // set absolute mixing precision to 10X more
         // than the difference in the samples
         library.setPrecision(0.1*(2*radius)/SAMPLES);
         // mix new array of A, B, indicator
         for (int j=1; j<SAMPLES; j++) {
            double target = center+radius*(1-2*(double)j/SAMPLES);
            Fluid mixture = library.mix(A, target, B, 1-target);
            samp[j] = library.mix(indicator, 0.9, mixture, 0.1);
         }
         // wait 30 minutes for indicator to show activity
         library.waitFor(30*60);
         // find maximum activity
         bestActivity = -1;
         int bestIndex = -1;
         for (int j=1; j<SAMPLES; j++) {
            double act = library.luminescence(samp[j]).doubleValue();
            if (act > bestActivity)
               bestActivity = act; bestIndex = j;
         }
         // zoom window by factor of 2 around best activity
         center = center+radius*(1-2*(double)bestIndex/SAMPLES);
         radius = radius / 2;
         // move center away from edges
         if (center < radius) center = radius;
         if (center > 1-radius) center = 1-radius;
      }
      // print value and location of highest activity
      System.out.println("Found activity " + bestActivity +
                         " at " + center + " +/- " + radius);
   }
}
// Declares devices needed by RecursiveDescent.
interface SimpleLibrary extends FluidLibrary {
   Fluid input(Integer i);              // require array of fluid inputs
   Double luminescence(Fluid f); // require luminescence camera
}
```

Figure 3: Recursive descent search in BioStream.

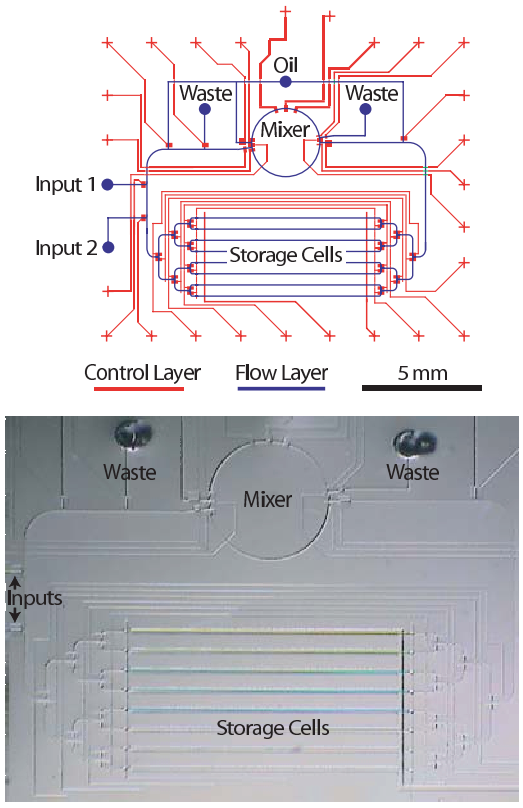

Figure 4: Layout and photo of Chip 1 (driven by oil).

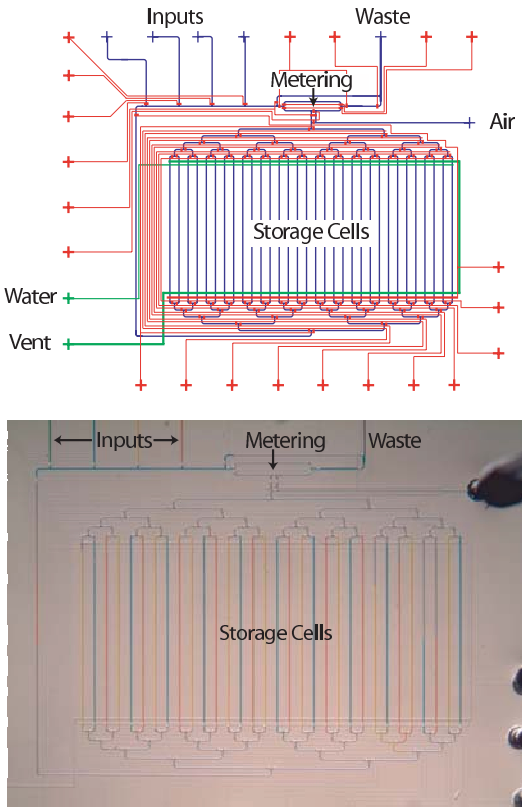

Figure 5: Layout and photo of Chip 2 (driven by air).

	Driving fluid	Wash fluid	Mixing	Sample size	Inputs	Storage cells	Valves	Control lines	Advantages
Chip 1	oil	N/A	rotary mixer	half of mixer	2	8	46	26	better sample isolation and retention
Chip 2	air	water	during transport	full mixer	4	32	140	21	faster and simpler chip operation

Table 1: Comparison of the key differences between the microfluidic chips developed.

3 Microfluidic Implementation

To demonstrate an end-to-end system, we have designed and fabricated two microfluidic chips using a standard multi-layer soft lithography process [13]. While there are fundamental differences between the chips (see Table 1), both provide support for programmable mixing, storage, and transport of fluid samples. More specifically, both chips implement the `mixAndStore` operation in the Fluidic ISA: they can load two samples from storage, mix them together, and store the result. Thus, despite their differences, code written in BioStream will be portable between the chips.

The first chip (see Figure 4) isolates fluid samples by suspending them in oil [19]. To implement `mixAndStore`, each input sample is transported from a storage bin to one side of the mixer. The mixer uses rotary flow, driven by peristaltic pumps, to mix the samples to uniformity [20]. Following mixing, one half of the mixer is drained and stored in the target location. While the second half could also be stored, it is currently discarded, as the basic `mixAndStore` abstraction produces only one unit of output.

The second chip (see Figure 5) isolates fluid samples using air instead of oil. Because fluid transport is very rapid in the absence of oil, a dedicated mixing element is not needed. Instead, the input samples are loaded from storage and aligned in a metering element; when the element is drained, the samples are mixed during transport to storage. Because the samples are in direct contact with the walls of the flow channels, a small fraction of the sample is lost during transport. This introduces the need for a wash phase, to clean the channel walls between operations. Also, to maintain sample volumes, the entire result of mixing is stored. Any excess volume is discarded in future mixing operations, as the metering element has fixed capacity.

To demonstrate BioStream's portability between these two chips, consider the following code, which generates a gradient of concentrations:

```
Fluid blue = input(1);
Fluid yellow = input(2);
Fluid[] gradient = new Fluid[5];
for (int i=0; i<=4; i++) {
  gradient[i] = mix(blue, yellow, i/4.0, 1-i/4.0);
}
```

This code was used to generate the gradient pictured in Figure 4 and produces an identical result on both microfluidic devices. (The gradient shown in Figure 5 is different and was generated by a different program.)

4 Mixing Algorithms

The mixing and dilution of fluids plays a fundamental role in almost all bioanalytical procedures. Mixing is used to prepare input samples for analysis, to dilute concentrated substances, and to control reagent volumes. In DNA computing, mixing is needed for reagent preparation (e.g., DNA libraries, PCR buffers, detection assays) and, in some techniques, for restriction digests [21, 22] or fine-grained concentration control [23]. It is critical to provide integrated support for mixing on microfluidic devices, as otherwise the samples would have to leave the system every time a mixture is needed.

As described in the previous sections, a microfluidic chip supports the `mixAndStore` instruction from the Fluidic ISA. This operation simply mixes two fluids in equal proportions. However, the mix command in BioStream allows the programmer to specify complex mixtures involving multiple fluids in various concentrations. To bridge the gap between these abstractions, this section describes how to obtain a complex mixture using a series of simple steps. We describe an abstract model for mixing, an algorithm for minimizing the number of steps required, how to deal with error tolerances, and directions for future work.

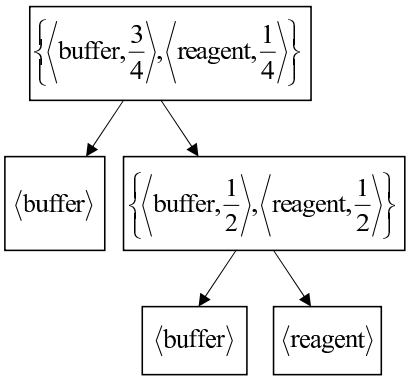

Figure 6: Mixing tree yielding 3/4 buffer and 1/4 reagent.

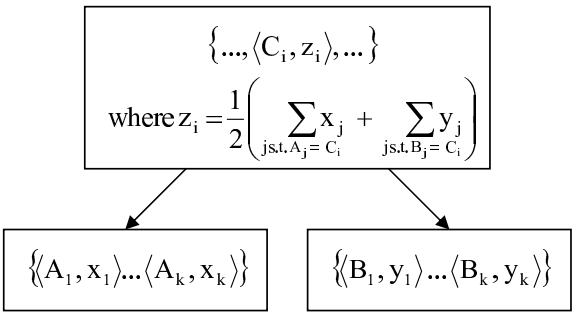

Figure 7: Calculation of a parent mixture from child mixtures using a 1-to-1 mixer. For each substance, the resulting concentration is the average of the concentrations in the children.

4.1 A Model of Mixing

The following definition gives our notation for mixtures.

Definition 1. *A mixture $\mathcal{M}$ is a set of substances S_i at given concentrations c_i:*

$$\mathcal{M} = \{\langle S_1, c_1\rangle \ldots \langle S_k, c_k\rangle\}$$
$$\textstyle\sum_{i=1}^{k} c_i = 1$$

For example, a mixture of $3/4$ buffer and $1/4$ reagent is denoted as $\{\langle \text{buffer}, 3/4\rangle, \langle \text{reagent}, 1/4\rangle\}$. We further define a ***sample*** to be a mixture with only one substance ($|\mathcal{M}| = 1$). For example, a sample of buffer is denoted $\{\langle \text{buffer}, 1\rangle\}$, or just $\langle \text{buffer}\rangle$.

To obtain a given mixture on a microfluidic chip, one performs a series of mixes using an on-chip mixing primitive. While the capabilities of this mixer might vary from one chip to another, a simple ***1-to-1 mixing model*** can be implemented on both continuous flow and droplet-based architectures [20, 24]. In this model, all fluids are stored in uniform chambers of unit volume. The mix operation combines two fluids in equal proportions, producing two units of the mixture. However, since there may be some amount of fluid loss with every operation, the result of the mixture might not be able to completely fill the contents of two storage cells. Thus, the result is stored in only one storage cell, and the extra mixture is discarded.

The 1-to-1 mixing process can be visualized using a "mixing tree". As depicted in Figure 6, each leaf node of a mixing tree represents a sample, while each internal node represents the mixture resulting from the combination of its children. Figure 7 illustrates that the mixture at an internal node can be calculated as the arithmetic mean of the components in child mixtures. In the 1-to-1 model, mixing trees are binary trees because each mix operation has two inputs. Evaluation of the tree proceeds from the leaf nodes upwards; the mixture for a given node can be produced once the child mixtures are available. The overall result of the operation is the mixture specified at the root node.

The following theorem is useful for reasoning about mixing trees. It describes the concentration of a substance in the overall mixture based on the depths of leaf nodes containing samples of the substance. The *depth* of a node n in a binary tree is the length of the path from the root node to n.

Theorem 1. *Consider a mixing tree and a substance S. Let m_d denote the number of leaf nodes with sample $\langle S\rangle$ appearing at depth d of the tree. Then the concentration of S contained in the root mixture is given by $\sum_d m_d * 2^{-d}$.*

Proof. A sample at depth d is diluted d times in the mixing process, each time by a factor of two. Thus it contributes 2^{-d} to the root mixture. Since each mix operation sums the concentrations from child nodes, the overall contribution is the sum across the leaf nodes at all depths $\sum_d m_d * 2^{-d}$. □

The following theorem describes the set of mixtures that can be obtained using a 1-to-1 mixer. Informally, it states that a mixture is reachable if and only if the concentration of each substance can be written as an integral fraction $k/2^d$.

Theorem 2. *(1-to-1 Mixing Reachability) Consider a finite set of substances $\{S_1 \ldots S_k\}$ with an unlimited supply of samples $\langle S_i \rangle$. Let $\mathcal{R}$ denote the set of mixtures that can be obtained via any sequence of 1-to-1 mixes. Then:*

$$\mathcal{R} = \left\{ \begin{array}{c} \{\langle S_1, c_1 \rangle \ldots \langle S_k, c_k \rangle\} \; s.t. \; \exists \, p_i, q_i, d \in \mathcal{Z} : \\ LCM(q_1 \ldots q_k) = 2^d \; \wedge \; \forall i \in [1,k] : c_i = \dfrac{p_i}{q_i} \end{array} \right\}$$

Proof. The equality in the theorem can be shown via bi-directional inclusion of $\mathcal{R}$ and the right hand side (RHS).

$\mathcal{R} \subseteq$ RHS: Given a mixing tree for the mixture, construct p_i, q_i, and d as follows to satisfy the RHS. Select d as the maximum depth of the tree (i.e., the maximum path length from the root node to a leaf node) and set all $q_i = 2^d$, thereby satisfying the LCM condition. Then, for leaf nodes at a depth less than d, replace the node with an internal node whose children are leaves with the same sample as the original. This preserves the identity of the mixture but increases the depth of some nodes. Iterate until all leaf nodes are at depth d. By Theorem 1, if a substance has concentration c_i in the mixture then it must have $c_i * 2^d$ leaf nodes in this tree. Thus, setting p_i to the number of leaf nodes with sample $\langle S_i \rangle$, we have that $p_i/q_i = c_i * 2^d/2^d = c_i$ as required.

$\mathcal{R} \supseteq$ RHS: Given a mixture satisfying the RHS and values of p_i, q_i, and d satisfying the conjuncts, construct a mixing tree that can be used to obtain the given mixture. The tree has d levels and 2^d leaves. Assign sample $\langle S_i \rangle$ to any $p_i * 2^d/q_i$ of the leaves (this is an integral quantity because 2^d is a common multiple of the q_i). By the definition of mixture, $\sum_i (p_i/q_i) = \sum_i c_i = 1$ and there is a one-to-one mapping between leaves and samples. By Theorem 1, the resulting mixture has a concentration of $k/2^d$ for a substance with k samples at the leaves. Thus the concentration for S_i in the assignment is $(p_i * 2^d/q_i)/2^d = p_i/q_i = c_i$ as desired. □

It is natural to suggest a number of optimization problems for mixing. Of particular interest are the number of mixes and the number of samples consumed, as these directly impact the running time and resource requirements of a laboratory experiment. The following theorem shows that (under the 1-to-1 model) these two optimization problems are equivalent.

Theorem 3. *In any 1-to-1 mixing sequence, the number of samples consumed is exactly one greater than the number of mixes.*

Proof. By induction on the number of nodes, there is always exactly one more leaf node than internal node in a binary tree. The mixing tree is a binary tree in which each internal node represents a mix and each leaf node represents a sample. Thus there is always exactly one more sample consumed than there are mixes. □

Note that this theorem only holds under the 1-to-1 mixing model, in which two units of volume are mixed but only one unit of the mixture is retained. For microfluidic chips that attempt to retain both units of mixture (such as droplet-based architectures or our oil-driven chip), it might be possible to decrease the number of samples consumed by increasing the number of mix operations.

4.2 Algorithm for Optimal Mixing

In this section, we give an efficient algorithm for finding a mixing tree that requires the minimal number of mixes to obtain a given concentration. For clarity, we frame the problem as follows:

```
node buildMixingTree
        (mixture { ⟨S1, p1 / n⟩, ..., ⟨Sk, pk / n⟩ } ) {
   depth = lg(n)
   bins = new stack[depth+1]
   // pre-processing: build a stack of the
   // bitwise components of each concentration
   for i = 1 to k
      mask = 1
      for j = 0 to depth-1
         if (mask & pi ≠ 0) then
            bins[j].push(Si)
         endif
         mask = mask << 1
      endfor
   endfor
   return buildMixingHelper(bins, depth)
}

node buildMixingHelper(stack[] bins, int pow) {
   if bins[pow].empty() then
      node child1 = buildMixingHelper(bins, pow-1)
      node child2 = buildMixingHelper(bins, pow-1)
      return ⟨child1, child2⟩ as internal node
   else
      return bins[pow].pop() as leaf node
   endif
}
```

Figure 8: MIN-MIX algorithm.

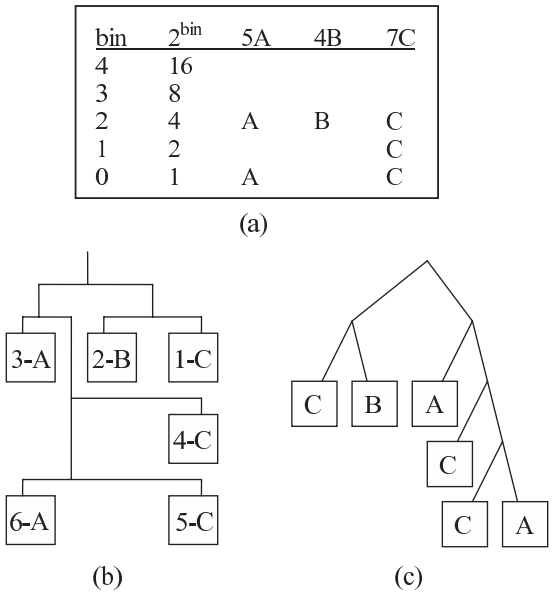

Figure 9: Example operation of MIN-MIX for the mixture $\{\langle A, 5/16\rangle, \langle B, 4/16\rangle, \langle C, 7/16\rangle\}$. Part (a) illustrates the algorithm's allocation of substances to bins. The bin layout directly translates to a valid mixing tree, which appears in (b) with numbers indicating the order in which nodes are added to the tree. The mixing tree is redrawn in (c) for clarity.

Problem 1. *(Minimal Mixing) Consider a finite set of substances $\{S_1 \ldots S_k\}$ with an unlimited supply of samples $\langle S_i\rangle$. Given a reachable mixture $\{\langle S_1, p_1/n\rangle \ldots \langle S_k, p_k/n\rangle\}$, what is the mixing tree with the minimal number of leaves?*

Our algorithm runs in $O(k \lg n)$ time[1] and produces an optimal mixing tree (with respect to this metric). The tree produced has no more than $k \lg n$ internal nodes.

The idea behind the algorithm, which we refer to as MIN-MIX, is to place a leaf node with sample $\langle S\rangle$ at depth d in the mixing tree if and only if the target concentration for S has a 1 in bit $\lg n - d$ of its binary representation. Theorem 1 then ensures that all substances have the desired concentrations, while fewer than $\lg n$ samples are used for each one.

Psuedocode for MIN-MIX appears in Figure 8. We illustrate its operation for the example mixture of $\{\langle A, 5/16\rangle, \langle B, 4/16\rangle, \langle C, 7/16\rangle\}$. As shown in Figure 9, the algorithm begins with a pre-processing stage that allocates substances to bins according to the binary representation of the target concentrations. It then builds the mixing tree via calls to `buildMixingHelper`, which descends through the bins. When a bin is empty, an internal node is created in the graph and the procedure recurses into the next bin. When a bin has a substance identifier in it, the substance is removed from the bin and a corresponding sample is added as a leaf node to the graph. Figure 9 labels the order in which the nodes in the final mixing tree are created by the algorithm.

The following lemma is key to proving the correctness of MIN-MIX. We denote the nth least significant bit of x by $\text{LSB}(x, n)$. That is, $\text{LSB}(x, n) \equiv (x \ll n)\ \&\ 1$.

[1] $\lg n$ denotes $\log_2 n$.

Lemma 1. *Consider the mixing tree t produced by* MIN-MIX($\{\langle S_1, p_1/n\rangle \ldots \langle S_k, p_k/n\rangle\}$). *A substance S_i appears at a depth d in t if and only if LSB$(p_i, \lg n - d) = 1$.*

Proof. **If:** It suffices to show that there is a substance added to the mixing tree for each LSB of 1 drawn from the p_i (that the substance appears at depth d is given by the only if direction.) Further, since bins[j] is constructed to contain all substances i for which LSB$(p_i, j) = 1$, it suffices to show that a) all bins are empty at the end of the procedure, and b) the procedure does not try to pop from an empty bin. To show (a), use the invariant that each call to `buildMixingHelper` adds a total of 2^{-d} to the mixing tree, where d is the current depth; either a leaf node is added (which contributes 2^{-d} by Theorem 1) or two child nodes are added, contributing $2 * 2^{-(d+1)} = 2^{-d}$. But since the initial depth is 0, the external call results in $2^0 = 1$ unit of mixture being generated. Since the bins represent exactly one unit of mixture (i.e.,$\sum_j bins[j] * 2^{-j} = 1$), all bins will be used. To show (b), observe that `buildMixingTree` references the bins in order, testing if each is empty before proceeding. Thus no empty bin will ever be dereferenced.

Only if: When a substance is added to the tree from bins[j], it appears at depth $\lg n - j$ in the tree. This is evident from the recursive call in `buildMixingHelper`: it initially draws from bins[$\lg n$] and then works down when the upper bins are empty. By construction, bins[j] contains only substances S_i with LSB$(p_i, j) = 1$. Thus, if S_i appears at depth d in the mixing tree, it was added from bins[$\lg n - d$] which has LSB$(p_i, \lg n - d) = 1$. □

The following theorem asserts the correctness of MIN-MIX.

Theorem 4. *The mixing tree given by* MIN-MIX *gives the correct concentration for each substance in the target mixture.*

Proof. Consider a component $\langle S, p/n\rangle$ of the mixture passed to MIN-MIX. Let m_d denote the number of leaf nodes with sample S at depth d of the resulting mixing tree. By Lemma 1, $m_d =$ LSB$(p, lg(n) - d)$. Using Theorem 1, this implies that the concentration for S in the root mixture is given by:

$$\begin{aligned} c &= \sum_d \text{LSB}(p, lg(n) - d) * 2^{-d} \\ &= \sum_x \text{LSB}(p, x) * 2^{-(lg(n)-x)} \\ &= \sum_x \text{LSB}(p, x) * 2^x/n \\ &= p/n \end{aligned}$$

Thus the concentration in the root node of the mixing tree is the same as that passed to MIN-MIX. □

The following theorem asserts the optimality of the mixing trees produced by MIN-MIX.

Theorem 5. *Consider the mixing tree t produced by* MIN-MIX($\{\langle S_1, p_1/n\rangle \ldots \langle S_k, p_k/n\rangle\}$). *The number of leaf nodes $\mathcal{L}(t)$ is given by:*

$$\mathcal{L}(t) = \sum_{i=1}^{k} \sum_{j=0}^{\lg n} LSB(p_i, j)$$

There does not exist a mixing tree that yields the given mixture with fewer leaf nodes than $\mathcal{L}(t)$.

Proof. That MIN-MIX produces a tree t with $\mathcal{L}(t)$ leaf nodes follows directly from Lemma 1, as there is a one-to-one correspondence between leaf nodes and input samples. To prove optimality, Theorem 1 gives that $p_i/n = \sum_d m_d * 2^{-d}$. Thus $p_i = \sum_d m_d * 2^{\lg n - d} = \sum_d \sum_{i=1}^{m_d} 2^{\lg n - d}$. That is, p_i is a sum of powers of two, and the number of leaf nodes determines the number of summands. The minimal number of summands is the number of non-zero bits in the binary representation for p_i; this quantity is $\sum_{j=0}^{\lg n} LSB(p_i, j)$. Thus it is impossible to obtain a concentration of p_i for all k substances in the tree with fewer than $\sum_{i=1}^{k} \sum_{j=0}^{\lg n} LSB(p_i, j)$ leaf nodes. □

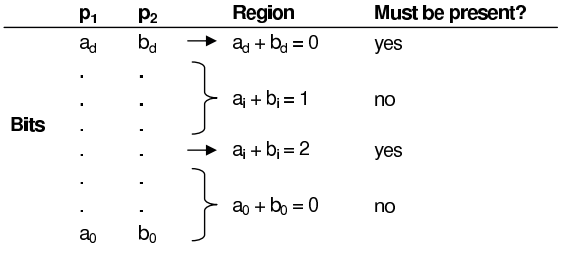

Figure 10: Arrangement of bits for any $p_1 + p_2 = 2^d$.

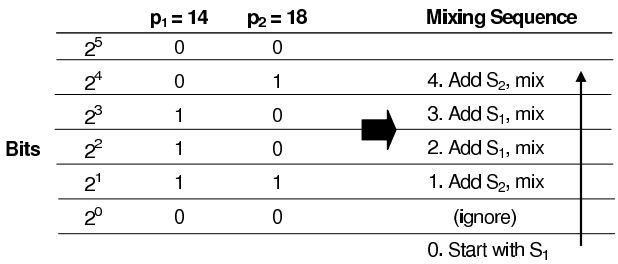

Figure 11: Example of mixing $14/32$ and $18/32$ using `twoWayMix`.

```
twoWayMix(mixture { ⟨S1, p1/n⟩, ⟨S2, p2/n⟩ } ) {
  // start with S2
  fluid = S2

  // ignore where both are zero
  int start = 0
  while (LSB(p1, start) = 0)
    start = start + 1
  endwhile

  // keep running mixture, based on bits of p1:
  //   bit is 0 - mix with S2
  //   bit is 1 - mix with S1
  for i = start to lg(n)-1
    fluid = mix (fluid, S2-LSB(p1, i) )
  endfor

  return fluid
}
```

Figure 12: Algorithm for mixing two substances.

The following theorem describes the running time of MIN-MIX.

Theorem 6. MIN-MIX($\{\langle S_1, p_1/n\rangle \ldots \langle S_k, p_k/n\rangle\}$) *runs in* $O(k \lg n)$ *time.*

Proof. The pre-processing stage in `buildMixingTree` executes $k \lg n$ iterations with constant cost per iteration. By Theorem 5, the recursive procedure returns a tree with $\sum_{i=1}^{k} \sum_{j=0}^{lg(n)} LSB(p_i, j) = O(k \lg n)$ leaf nodes, and by Theorem 3 this implies that there are $O(k \lg n)$ total nodes in the tree. Since there is constant cost at each node, the overall complexity is $O(k \lg n)$. □

4.3 Special Case: Mixing Two Substances

The minimal mixing tree admits a particularly compact representation when only two substances $\langle s_1, p_1/n\rangle$ and $\langle s_2, p_2/n\rangle$ are being mixed. Because the two target concentrations must sum to a power of two (in order to be reachable with a 1-to-1 mixer), there is a special pattern in the bitwise representation of p_1 and p_2 (see Figure 10). The least significant bits might be zero in both concentrations, but then some bit must be one in each of them. The higher-order bits must be one in exactly one of the concentrations (to carry a value upwards) and the most significant bit is zero (as we assume $p_1, p_2 < n$).

Algorithm `twoWayMix`, shown in Figure 12, exploits this pattern to directly execute the mix sequence without building a mixing tree. The sequence of mixes is completely encoded in the binary representation of either concentration. As illustrated by the example in Figure 11, the algorithm starts with a unit of S_2 and then skips over all the low-order zero bits (these result from a fraction p_1/n that is not in lowest terms). When it gets to a high bit, it maintains a running mixture–requiring no temporary storage–in which either S_1 or S_2 is added to the mix depending on the next most significant bit of p_1. It can be shown that this procedure is equivalent to building a mixing tree. However, it is attractive from hardware design standpoint due to its simplicity and the fact that it directly performs a mixture based on the binary representation of the desired concentration.

4.4 Supporting Error Tolerances

Thus far the presentation has been in terms of mixtures that can be obtained exactly with a 1-to-1 mixer, i.e., those with target concentrations in the form of $k/2^d$. However, the programmer should not be concerned with the reachability of a given mixture. In the BioStream system, the programmer specifies a concentration range $[c_{min}, c_{max}]$ and the system ensures that the mixture produced will fall within the given range[2]. Such error tolerances are already a natural aspect of scientific experiments, as all measuring equipment has a finite precision that is carefully noted as part of the procedure.

Given a concentration range, the system increases the internal precision d until some concentration $k/2^d$ (which can be obtained exactly) falls within the range. When performing a mixture with concentration ranges $\{\langle S_1, [c_{1,min}, c_{1,max}]\rangle \ldots \langle S_k, [c_{k,min}, c_{k,max}]\rangle\}$ the system needs to choose concrete concentrations c_i and a precision d that satisfies the following conditions:

1. $\forall i : \exists k_i \; s.t. \; c_i = k_i/2^d$
2. $\forall i : c_{i,min} \leq c_i \leq c_{i,max}$
3. $\sum_i c_i = 1$

The first condition guarantees that the mixture can be obtained using a 1-to-1 mixer. The second condition states that the concrete concentrations c_i are within the range specified by the programmer. The third condition ensures that the c_i form a valid mixture, i.e., that they sum to one.

The BioStream system uses a simple greedy algorithm to choose c_i and d satisfying these conditions. It increases d until there exists a c_i satisfying (1) and (2) for all i. If multiple candidates for a given c_i exist, it selects the smallest possible. Then it checks condition (3). If the sum exceeds one, it increases d and starts over. If the sum is less than one, it increases by $1/2^d$ some c_i for which $c_i \leq c_{i,max} - 1/2^d$. If no such c_i exists, it increases d and starts over. Otherwise the algorithm continues until the conditions are satisfied.

One can imagine other selection schemes that select c_i and d to optimize some criterion, such as the number of mixes required by the resulting mixture. This would be straightforward to implement via an exhaustive search at a given precision level, but it could be costly depending on the size of the error margins. It will be a fruitful area of future research to optimize the selection of target concentrations while respecting the error bounds.

4.5 Open Problems

We suggest three avenues for future research in mixing algorithms.

N-to-M mixing. It is simple to build a rotary mixer that combines fluids in a ratio other than 1-to-1; for example, 1-to-2, 1-to-3, or even a ternary mixer such as 1-to-2-to-3. Judging by exhaustive experiments, it appears that a 1-to-2 mixer can obtain any concentration $k/3^n$. However, we are unaware of a closed form for the mixtures that can be obtained with a general N-to-M mixer. Likewise, we consider it to be an open problem to formulate an efficient algorithm for determining the minimal mix sequence using an N-to-M mixer (i.e., one that does not resort to an exhaustive lookup table.) A solution to this problem could reduce mixing time and reagent consumption while increasing precision.

Minimizing storage requirements. Given a mixing tree, it is straightforward to find an evaluation order that minimizes the number of temporaries; one can apply the classical node labeling algorithm that minimizes register usage for trees [25, p. 561]. However, we are unaware of an efficient algorithm for finding the mixing tree that minimizes the number of temporaries needed to obtain a given mixture. This could be an important optimization, as experiments often demand as many parallel samples as can be supported by the architecture. Also, storage chambers on microfluidic chips are relatively limited and expensive compared to storage on today's computers.

[2] Alternately, BioStream supports a global error tolerance ϵ that applies to all concentrations.

Heterogeneous inputs. Our presentation treats each input sample as a black box. However, in practice, the user is able to prepare large quantities of reagents as inputs to the chip. For an application that produces an array of concentrations, what inputs should the user prepare to minimize the number of mixes required? And if some inputs are related (e.g., a sample of 10% acid and 20% acid) how can that be incorporated into the mixing algorithm? Like the previous items, these are interesting algorithmic questions that can have a practical impact.

5 Related Work

Several researchers have pursued the goal of automating the control systems for microfluidic chips. Gascoyne et al. describe a graphical user interface for controlling chips that manipulate droplets over a two-dimensional grid [26]. By varying parameters in the interface, the software can target grids with varying dimensions, speeds, etc. However, portability is limited to grid-based droplet processors. While the BioStream protocol language could target their chips, their software is not suitable for targeting ours.

Su et al. represent protocols as acyclic sequence graphs and map them to droplet-based processors using automatic scheduling [27] and module placement [28]. While the sequence graph is portable, it lacks the expressiveness of a programming language and cannot represent feedback loops (as in our recursive descent example).

Livstone et al. compile an abstract SAT problem into a sequence of DNA-computing steps [5]. The output of their system would be a good match for BioStream and the abstraction layers proposed in this paper. King et al. demonstrate a "robot scientist" that directs laboratory experiments using a high-level programming language [29], but lacks the abstraction layers needed to target other devices. Gu et al. have controlled microfluidic chips using a programmable Braille display [30], though this approach also lacks portability.

There are other microfluidic chips that support flexible gradient generation [31–33] and programmable mixing on a droplet array [34]. To the best of our knowledge, our chips are the only ones that provide arbitrary mixing of discrete samples in a soft lithography medium. A more detailed comparison of the devices is published elsewhere [19].

Fair et al. also suggest a mixing algorithm for diluting a single reagent by a given factor [35]. It seems that their algorithm performs a binary search for the target concentration, progressively approximating the target by a factor of two. However, since intermediate reagents must be regenerated in the search, this algorithm requires $O(n)$ mixes to obtain a concentration k/n. In contrast, our algorithm needs $O(\lg n)$ to mix two fluids.

6 Conclusions

Microfluidic devices are an exciting substrate for biological computing because they allow precise and automatic control of the underlying biological protocols. However, as the complexity of microfluidic hardware comes to rival that of silicon-based computers, it will be critical to develop effective abstraction layers that decouple application development from low-level hardware details.

This paper presents two new abstraction layers for microfluidic biocomputers: the BioStream protocol language and the Fluidic ISA. Protocols expressed in BioStream are portable across all devices implementing a given Fluidic ISA. We demonstrate this portability by building two fundamentally different microfluidic devices that support execution of the same BioStream code. We also present a new and optimal algorithm for obtaining a given concentration of fluids using a simple on-chip mixing device. This algorithm is essential for efficiently supporting the mix abstraction in the BioStream language.

It remains an interesting area of future work to leverage DNA computing technology to target the BioStream language from a high-level description of the computation. This will create an end-to-end platform for biological computing that is seamlessly portable across future generations of microfluidic chips.

7 Acknowledgements

We are grateful to David Wentzlaff and Mats Cooper for early contributions to this research. We also thank John Albeck for helpful discussions about experimental protocols. This work was partially supported by National Science Foundation grant #CNS-0305453. J.P.U. was funded in part by the National Science and Engineering Research Council of Canada (PGSM Scholarship).

References

[1] R. S. Braich, N. Chelyapov, C. Johnson, P. W. K. Rothemund, and L. Adleman. Solution of a 20-variable 3-SAT problem on a DNA computer. *Science*, 296, 2002.

[2] J. Farfel and D. Stefanovic. Towards practical biomolecular computers using microfluidic deoxyribozyme logic gate networks. In *DNA 11*, 2005.

[3] A. Gehani and J. Reif. Micro flow bio-molecular computation. *Biosystems*, 52, 1999.

[4] W. H. Grover and R. A. Mathies. An integrated microfluidic processor for single nucleotide polymorphism-based DNA computing. *Lab on a Chip*, 5, 2005.

[5] M. S. Livstone, R. Weiss, and L. F. Landweber. Automated design and programming of a microfluidic DNA computer. *Natural Computing*, 2006.

[6] J. S. McCaskill. Optically programming DNA computing in microflow reactors. *BioSystems*, 59, 2001.

[7] K. Somei, S. Kaneda, T. Fujii, and S. Murata. A microfluidic device for DNA tile self-assembly. In *DNA 11*, 2005.

[8] D. van Noort. A programmable molecular computer in microreactors. In *DNA 11*, 2005.

[9] D. van Noort, F.-U. Gast, and J. S. McCaskill. DNA computing in microreactors. In *DNA 8*, 2002.

[10] D. van Noort and B.-T. Zhang. PDMS valves in DNA computers. In *SPIE International Symposium on Smart Materials, Nano-, and Micro-Smart Systems*, 2004.

[11] D. N. Breslauer, P. J. Lee, and L. P. Lee. Microfluidics-based systems biology. *Molecular BioSystems*, 2, 2006.

[12] D. Erickson and D. Li. Integrated microfluidic devices. *Analytica Chimica Acta*, 507, 2004.

[13] S. K. Sia and G. M. Whitesides. Microfluidic devices fabricated in poly(dimethylsiloxane) for biological studies. *Electrophoresis*, 24, 2003.

[14] T. Thorsen, S. Maerkl, and S. Quake. Microfluidic large scale integration. *Science*, 298, 2002.

[15] J. W. Hong and S. R. Quake. Integrated nanoliter systems. *Nature BioTechnology*, 21(10), 2003.

[16] W. Thies, J. P. Urbanski, T. Thorsen, and S. Amarasinghe. Abstraction layers for scalable microfluidic biocomputers. In *DNA 12*, 2006.

[17] H. Ellerby, S. Martin, L. Ellerby, S. Naiem, S. Rabizadeh, G. Salvesen, C. Casiano, N. Cashman, D. Green, and D. Bredesen. Establishment of a cell-free system of neuronal apoptosis: Comparison of premitochondrial, mitochondrial, and postmitochondrial phases. *Neuroscience*, 17, 1997.

[18] L. Allan, N. Morrice, S. Brady, G. Magee, S. Pathak, and P. Clarke. Inhibition of caspase-9 through phosphorylation at Thr 125 by ERK MAPK. *Nature Cell Biology*, 5, 2003.

[19] J. P. Urbanski, W. Thies, C. Rhodes, S. Amarasinghe, and T. Thorsen. Digital microfluidics using soft lithography. *Lab on a Chip*, 6, 2006.

[20] H. Chou, M. Unger, and S. Quake. A microfabricated rotary pump. *Biomedical Microdevices*, 3, 2001.

[21] D. Faulhammer, A. R. Cukras, R. J. Lipton, and L. F. Landweber. Molecular computation: RNA solutions to chess problems. *PNAS*, 97(4), 2000.

[22] Q. Ouyang, P. D. Kaplan, S. Liu, and A. Libchaber. DNA solution of the maximal clique problem. *Science*, 278, 1997.

[23] M. Yamamoto, N. Matsuura, T. Shiba, Y. Kawazoe, and A. Ohuchi. Solutions of shortest path problems by concentration control. In *DNA 7*, 2002.

[24] P. Paik, V. Pamula, and R. Fair. Rapid droplet mixers for digitial microfluidic systems. *Lab on a Chip*, 3, 2003.

[25] R. S. Alfred V. Aho and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, second edition, 1988.

[26] P. R. C. Gascoyne, J. V. Vykoukal, J. A. Schwartz, T. J. Anderson, D. M. Vykoukal, K. W. Current, C. McConaghy, F. F. Becker, and C. Andrews. Dielectrophoresis-based programmable fluidic processors. *Lab on a Chip*, 4, 2004.

[27] F. Su and K. Chakrabarty. Architectural-level synthesis of digital microfluidics-based biochips. In *ICCAD*, 2004.

[28] F. Su and K. Chakrabarty. Unified high-level synthesis and module placement for defect-tolerant microfluidic biochips. In *DAC*, 2005.

[29] R. D. King, K. E. Whelan, F. M. Jones, P. G. K. Reiser, C. H. Bryant, S. H. Muggleton, D. B. Kell, and S. G. Oliver. Functional genomic hypothesis generation and experimentation by a robot scientist. *Nature*, 427, 2004.

[30] W. Gu, X. Zhu, N. Futai, B. S. Cho, and S. Takayama. Computerized microfluidic cell culture using elastomeric channels and Braille displays. *PNAS*, 101(45), 2004.

[31] S. K. W. Dertinger, D. T. Chiu, N. L. Jeon, and G. M. Whitesides. Generation of gradients having complex shapes using microfluidic networks. *Anal. Chem.*, 73, 2001.

[32] C. Neils, Z. Tyree, B. Finlayson, and A. Folch. Combinatorial mixing of microfluidic streams. *Lab on a Chip*, 4, 2004.

[33] F. Lin, W. Saadi, S. W. Rhee, S.-J. Wang, S. Mittalb, and N. L. Jeon. Generation of dynamic temporal and spatial concentration gradients using microfluidic devices. *Lab on a Chip*, 4, 2004.

[34] M. Pollack, R. Fair, and A. Shenderov. Electrowetting-based actuation of liquid droplets for microfluidic applications. *Applied Physics Letters*, 77(11), 2000.

[35] H. Ren, V. Srinivasan, and R. Fair. Design and testing of an interpolating mixing architecture for electrowetting-based droplet-on-chip chemical dilution. *Transducers*, 2003.