Compiling computations to constraints for verified computation

Benjamin Braun
Thesis supervised by Dr. Michael Walfish
The University of Texas at Austin, Austin, TX

Abstract. We present a compiler that automates the task of converting high-level code to constraint sets of the form accepted by the Ginger and Zaatar protocols for verified computation. Performing the conversion from high-level code to constraints by hand is prone to human error and therefore not practical for large computations. This paper evaluates the performance of the compiler and the effectiveness of its optimizations on reducing the size of the constraint set. We show that the compiler can produce constraint sets for a number of interesting computations, including DNA sequence alignment of 200 nucleotide sequences and partition-about-medoids clustering of 100-dimensional data into two clusters.

1 Introduction

Resources such as the cloud or peer-to-peer networks offer powerful computing resources to clients at low prices. When clients outsource work to a remote computer, they risk that incorrect results may be returned. Verified computation is a cryptographic primitive for checking whether the remote computer carried out the computation correctly.

There are many ways a client could check that a remote computer has returned correct results. A client could simply outsource the same computation multiple times to different remote computers, and check that the results match [2, 5, 10, 14]. However, in this approach the remote computers could collude and agree to return a particular, incorrect, output. Another strategy the client could employ is to leverage trusted hardware [6, 16] and remote attestation [1, 15] to ensure that the remote machine is running software that the client knows is correct. These approaches place trust in the manufacturer of the trusted hardware.

On the other hand, new protocols for verified computation are emerging that can detect when a remote computer has returned incorrect results with a bounded probability of error and that make no assumptions about the remote computer. Some protocols of this kind have been built and evaluated, including those of Thaler et. al. [8, 22] and the Pepper, Ginger, and Zaatar protocols of Setty et. al. [18–21]. These protocols were developed to turn results from complexity theory, such as probabilistically checkable proofs [3, 4], into nearly-practical systems.

In order to use the Ginger and Zaatar protocols to outsource a computation, the computation must be represented as a set of constraints. This paper presents a multi-target compiler that takes high-level source code for a computation and produces a representation of the same computation as a set of constrains in the format required by the Ginger and Zaatar protocols.

The first part of this paper gives an overview of the Ginger and Zaatar protocols and introduces the problem of converting a computation to a set of constraints. Next, we presents the Broader Function Description Language (BFDL), a high-level language for specifying computations that have equivalent constraint sets. Next, we develops a systematic approach to compiling a computation expressed in BFDL to constraints. Finally, we describe an optimizing compiler that implements this approach. We use the compiler to produce constraint sets for a number of interesting benchmark computations and investigate the effectiveness of various compiler optimizations on reducing the size of generated constaint sets. We close with known limitations to the compiler, and a discussion of future work.

We include BFDL code for all benchmark computations in the appendices.

2 Background

We give an overview of the Ginger and Zaatar protocols, focusing on the format of contraint sets these protocols accept.

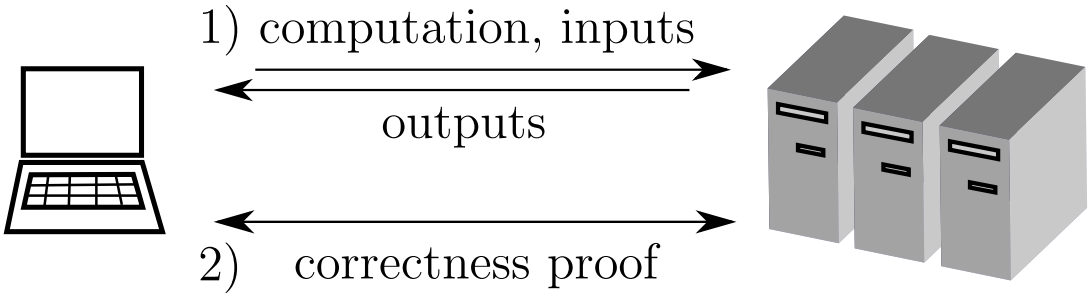

Figure 1—Diagram of verified computation using the Ginger or Zaatar protocols. A client, left, first sends a specification of a computation and a batch of input datasets to an untrusted remote computer, right, which may be part of the cloud or a peer-to-peer network. The remote computer runs the specified computation on the inputs, and returns the output of the computation. Finally, the client and remote computer enter an interactive protocol which allows the client to determine whether the remote computer performed the computation correctly.

2.1 Protocols for verified computation

Ginger. The Ginger protocol [21] provides an efficient, general-purpose way to perform verified computation, illustrated in Figure 1. The protocol makes use of a linear commitment primitive which is a refined version of the primitive of Ishai et al. [11]. During the Ginger protocol, both the client and the remote computer perform operations which make use of a representation of the computation being outsourced as a set of constraints.

More precisely, in order to run the Ginger protocol, the computation being outsourced must be represented as a set of quadratic constraints over the field of integers modulo p, $\mathbb{Z}_p$, where p is some large prime. The terminology used in the preceding statement requires some explanation. A *quadratic constraint* is an equation of total degree 2 that uses additions and multiplications. A polynomial (or equation) has total degree n whenever each of the terms of the polynomial (or equation) can be written as a product or at most n variables and a constant factor. For example, $Z_1 + Z_2 - 2Z_1 \cdot Z_2 = 0$ is a quadratic constraint. A set $\mathcal{C}$ of quadratic constraints over variables Z_i is satisfiable if there is some assignment to the Z_i such that all of the constraints in $\mathcal{C}$ hold simultaneously. The variables are partitioned into three subsets, X, Y, and M, where the elements of X are designated "input variables," the elements of Y are designated "output variables," and the elements of M are said to be "intermediate variables." The names of these designations will make more sense once the notion of constraints-computation equivalence is defined, below.

Let $\mathcal{C}(X = x, Y = y)$ denote the constraint set where each variable X_i is fixed to have value x_i and each variable Y_i is fixed to have value y_i. Thus, the M_i are the only free variables in $\mathcal{C}(X = x, Y = y)$.

Constraints-Computation Equivalence

A set of constraints $\mathcal{C}$ is *equivalent* to a computation $\Psi : \mathbb{Z}_p^{|X|} \to \mathbb{Z}_p^{|Y|}$ if, for any $x \in \mathbb{Z}_p^{|X|}$ and $y \in \mathbb{Z}_p^{|Y|}$, $\mathcal{C}(X = x, Y = y)$ is satisfiable iff. $y = \Psi(x)$.

For example, the following set of quadratic constraints $\mathcal{C}$ is equivalent to the computation Ψ which takes an integer in $\mathbb{Z}_p$ and returns $x + 1$.

$$\mathcal{C} = \left\{ \begin{array}{rcl} M_1 - X_1 & = & 0 \\ M_1 + 1 - Y_1 & = & 0 \end{array} \right\}$$

Proof. Consider two, possibly equal, values x, y in $\mathbb{Z}_p$. If $y = x + 1$, then assigning $M_1 = x$ satisfies $\mathcal{C}(X = x, Y = y)$. Otherwise, if $y \neq x + 1$, then there is no value of M_1 which satisfies $\mathcal{C}(X = x, Y = y)$. Since these two cases describe all possible values of x and y, $\mathcal{C}(X = x, Y = y)$ is satisfiable if and only if $y = \Psi(x)$. □

Once a constraint set $\mathcal{C}$ equivalent to a computation Ψ has been found, the client and remote computer can begin the Ginger protocol. The client first sends its input datasets to the remote computer. After the remote computer returns output for each of the input datasets, it proves to the client that it performed the computation correctly by allowing the client to query its assignments to the variables of $\mathcal{C}$. The client's queries have the form of vectors in $\mathbb{Z}_p^n$, where $n = |M| + |M|^2$ and $|M|$ is the number of intermediate variables in the constraint set $\mathcal{C}$. The remote computer responds to these queries by evaluating the dot product of each query vector with a specially formed proof vector $w \in \mathbb{Z}_p^n$. For a complete description of these queries and what they mean, as well as a proof that this protocol detects incorrect answers with high probability (under standard cryptographic assumptions), please see the original Ginger publication [21].

Zaatar. The Zaatar protocol results from modifications to the Ginger protocol inspired by Quadratic Arithmetic Programs (QAPs) [9]. The Zaatar protocol specifies that the constraints for the computation being outsourced must have the form $p_A(Z) \cdot p_B(Z) = p_C(Z)$, where p_A, p_B, and p_C are polynomials with total degree 1. Any such constraint is clearly a quadratic constraint. The compiler presented in this document can compile to quadratic constraints as well as to constraints in the format required by Zaatar.

The Zaatar protocol differs from Ginger in how the remote computer derives its proof vector w, and how the client queries this vector. As in Ginger, the remote computer proves to the client that it performed a computation correctly on a batch of datasets by allowing the client to perform certain queries. These queries take the form of vectors in $\mathbb{Z}_p^{n'}$, where $n' = |Z| + |\mathcal{C}| + 1$ and $|Z|$ is the number of variables and $|\mathcal{C}|$ is the number of constraints in a constraint set $\mathcal{C}$ for the computation being outsourced. The prover responds to a query q for each input dataset by evaluating the dot product between q and a specially constructed proof vector $w \in \mathbb{Z}_p^{n'}$. For a complete description of how the queries and the proof vector are constructed, please see [19].

3 BFDL

We desire a high level language for representing the class of computations which have equivalent constraint sets. Such a language should be designed so that a programmer can use the language without any knowledge of constraint sets, their formats, or the Ginger or Zaatar protocols. On the other hand, the language should not deceive the programmer about the expressiveness of constraints. We will show in section 6.1 that any computation which can be represented as a list of simple assignment statements can be written as a set of constraints. Hence, we seek a language which limits the programmer to using code structures which can be expanded into lists of assignment statements.

The problem of compiling computations to lists of assignment statements is also encountered when compiling high level code to circuits. The Secure Function Description Language (SFDL) was developed by researchers studying secure computation to be a language which compiles to "theoretician's Boolean circuits" [13]. SFDL limits the programmer to writing programs which can be expanded to a list of assignment statements. The only looping structures available in SFDL are for-loops whose loop bounds are compile-time constants. Function call recursion is not supported in SFDL.

The original SFDL compiler, the Fairplay compiler, unrolls for loops, separates multiple-arity assignments, and expands if-else statements to produce a list of assignment statements equivalent to the computation being compiled.

We derive our compiler from the Fairplay compiler. Our modified compiler outputs constraint sets of either the form accepted by Ginger or the form accepted by Zaatar. Because array access at an input-dependent index does not have an efficient expansion in terms of constraint sets, our modified compiler does not accept code where array accesses are not resolvable to constants. We augment the existing type system of the Fairplay compiler with type inference functionality so that fewer variables have to have declared types. Because of these modifications, our compiler accepts a different language than the original Fairplay compiler does. We term this new language BFDL, for Broadened Function Description Language.

Snippet 1 BFDL code to count occurrences of an input integer in a list of $m = 10$ input integers

```
 1. program test {
 2.   const m = 10;
 3.   type Input = struct{int<32> key, int<32>[m] list};
 4.   type Output = struct{int count};
 5.   function Output output (Input X){
 6.     var int count;
 7.     var int i;
 8.     count = 0;
 9.     for (i = 0 to m-1) {
10.       if (X.list[i] == X.key) {
11.         count = count + 1;
12.       }
13.     }
14.     output.count = count;
15.   }
16. }
```

Syntax of BFDL. BFDL uses C-style syntax, is statically typed, and supports type-inference. The language supports three primitive types: `boolean`, `int`, and `float`. The `boolean` type can have two values, 0, which is identified with the `false` keyword, and 1, which is identified with the `true` keyword. The `int` type by default represents an arbitrary precision integer. The `float` type represents a rational number whose integer is an arbitrary precision integer and whose denominator is an arbitrary precision integer, but which must be a power of two.

As an example, BFDL code for a computation that counts the number of occurrences of an input integer in an array of $m = 10$ input integers is given in Snippet 1. The structure of this code should be familiar to those familiar with the C programming language. The keywords `program`, `const`, `type`, `var`, `for`, and `function` begin a program, constant, type, variable, for-loop, or function definition, respectively. The main function of a program must have the name `output`. Struct types are defined with the `struct` keyword, in C fashion. Of interest is the `int<32>` type used in line 3, which is defined to be the type of a 32-bit signed integer, i.e. an integer in the range $[-2^{31}, 2^{31} - 1]$. Array indirection is supported, but the index into the array must be resolvable at compile time, i.e. not a function of the inputs. It is required that variables which are inputs to the main function (`output`) have declared types which are bounded, however other variables can be declared with the unqualified `int` and `float` types. Helper functions to the `output` function can be defined (see the appendix for examples.) However, function calls cannot be recursive. That is, a function can appear at most once in a function call stack of any BFDL program.

We will present a systematic approach for compiling the code in Snippet 1 to constraints. This approach will involve combining constraint sets for simple computations to produce a constraint set for a more complex computations. At first glance, the notion of compiling the code in Snippet 1 to constraints is not well defined, because while this example computation operates over 32-bit integers, we only defined constraint equivalence for computations over $\mathbb{Z}_p$. We will show how to form constraint sets for simple computations, and show how to resolve the difference between 32-bit integer computations and computations over Z_p, in the next sections.

4 Constraint sets for simple computations

This section presents constraint sets equivalent to simple computations. All of the presented constraint sets are in the form accepted by the Zaatar protocol, and are hence of the form accepted by Ginger as well.

Boolean function	Constraint
False	$0 - Y_1 = 0$
NOR	$(1 - X_1) \cdot (1 - X_2) - Y_1 = 0$
X_2, NOT X_1	$(1 - X_1) \cdot X_2 - Y_1 = 0$
NOT X_1	$1 - X_1 - Y_1 = 0$
X_1, NOT X_2	$X_1 \cdot (1 - X_2) - Y_1 = 0$
NOT X_2	$1 - X_2 - Y_1 = 0$
XOR	$(-2X_1) \cdot X_2 + X_1 + X_2 - Y_1 = 0$
NAND	$(-X_1) \cdot X_2 + 1 - Y_1 = 0$
AND	$X_1 \cdot X_2 - Y_1 = 0$
EQUAL	$(2X_1) \cdot X_2 - X_1 - X_2 + 1 - Y_1 = 0$
X_2	$X_2 - Y_1 = 0$
$X_1 \implies X_2$	$(-X_1) \cdot (1 - X_2) + 1 - Y_1 = 0$
X_1	$X_1 - Y_1 = 0$
$X_2 \implies X_1$	$(1 - X_1) \cdot (-X_2) + 1 - Y_1 = 0$
OR	$(X_1 - 1) \cdot (1 - X_2) + 1 - Y_1 = 0$
True	$1 - Y_1 = 0$

Figure 2—Each of the 16 Boolean functions of two inputs X_1 and X_2 can be represented as a single constraint.

4.1 Evaluating simple polynomials

Let Ψ be a computation which evaluates a polynomial over variables X taking values in $\mathbb{Z}_p$ of the form $p_A(X) \cdot p_B(X) + p_C(X)$, where p_A, p_B, and p_C are polynomials with total degree 1. Then the constraint $p_A(X) \cdot p_B(X) - p_C(X) - Y_1 = 0$ is clearly equivalent to Ψ.

One special case of the above result is that any Boolean expression of two inputs has an equivalent constraint set. For example, the NAND (not-and) of two Boolean (0 or 1) inputs X_1 and X_2 can be written as the polynomial $(-X_1) \cdot X_2 + 1$. Hence, an equivalent constraint for NAND is $(-X_1) \cdot X_2 + 1 - Y_1 = 0$. This process can be used to represent any two-input Boolean functions as a single constraint, as shown in Figure 2.

Another special case is the construction of equivalent constraints for a simple multiplexer, also called a mux. Let Ψ_{MUX} be a computation over $\mathbb{Z}_p$ which takes one Boolean input X_1 and two inputs X_2 and X_3, and returns X_2 if X_1 is 1 and X_3 if X_1 is 0. Then Ψ_{MUX} can be written as the polynomial $X_1 \cdot (X_2 - X_3) + X_3$, so a constraint equivalent to Ψ_{MUX} is $X_1 \cdot (X_2 - X_3) + X_3 - Y_1 = 0$.

4.2 Equality testing

Define the computation $\Psi_{!=}(X) : \mathbb{Z}_p^2 \to \mathbb{Z}_p$ which takes two values X_1 and X_2 in $\mathbb{Z}_p$ and returns 1 if X_1 and X_2 are different values of $\mathbb{Z}_p$ and 0 if they are the same. The constraint set shown below is equivalent to $\Psi_{!=}$.

$$\mathcal{C}_{!=} = \left\{ \begin{array}{rcl} M_1 \cdot (X_1 - X_2) - Y_1 & = & 0 \\ (1 - Y_1) \cdot (X_1 - X_2) & = & 0 \end{array} \right\}$$

Proof. Consider some $x \in \mathbb{Z}_p^2$ and some $y \in \mathbb{Z}_p$. The proof proceeds by cases.

Say $x_1 = x_2$, hence $\Psi_{!=}(x) = 0$. If $y = \Psi_{!=}(x) = 0$, then setting $M_1 = 0$ satisfies $\mathcal{C}_{!=}(X = x, Y = y)$. Otherwise, if $y \neq 0$, then the first constraint of $\mathcal{C}_{!=}(X = x, Y = y)$ is not satisfiable.

Take instead $x_1 \neq x_2$, so $\Psi_{!=}(x) = 1$. If $y = \Psi_{!=}(x) = 1$, then the constraint set $\mathcal{C}_{!=}(X = x, Y = y)$ is satisfied by setting M_1 to be the multiplicative inverse of $X_1 - X_2$ in $\mathbb{Z}_p$. Otherwise, if $y \neq 1$, then the second constraint of $\mathcal{C}_{!=}(X = x, Y = y)$ is not satisfiable.

Thus, $\mathcal{C}_{!=}(X = x, Y = y)$ is satisfiable if and only if $\Psi_{!=}(x) = y$. □

4.3 Order comparisons

The field $\mathbb{Z}_p$ is not usually defined with a notion of an ordering of its elements. However, the hope for outsourcing computations using Ginger and Zaatar would be bleak if we could only outsource computations that do not use the $<$ operation. We can define the $<$ binary operation on a subset U of $\mathbb{Z}_p$ by defining a mapping from $\mathbb{Z}$ to $\mathbb{Z}_p$; this allows elements of U to make use of the standard ordering inherited from $\mathbb{Z}$. Define the mapping θ:

$$\begin{aligned}\theta: \quad & \mathbb{Z} \to \mathbb{Z}_p \text{ where } p \text{ is a large, odd prime} \\ & \theta(x) = x \mod p\end{aligned}$$

We can define a computation $\Psi_{<,U}(X) : \mathbb{Z}_p^2 \to \mathbb{Z}_p$ which takes two values X_1 and X_2 in a subset $U \subset \mathbb{Z}_p$ and returns 1 if $\theta^{-1}(X_1) < \theta^{-1}(X_2)$ and 0 otherwise. Strictly speaking, θ is not uniquely invertible. However, we define its inverse θ^{-1} to be in the range $[-(p+1)/2, (p+1)/2-1]$, so that θ^{-1} maps an element of $\mathbb{Z}_p$ in the sequence $0, 1, 2, ...(p+1)/2-1$ to the corresponding element of the sequence of nonnegative integers $0, 1, 2, ...(p+1)/2-1$ and an element of $\mathbb{Z}_p$ in the sequence $(p+1)/2, ...p-2, p-1$ to the corresponding element of the sequence of negative integers $-(p+1)/2, -(p+1)/2+1... -2, -1$.

If U is defined such that $\theta^{-1}(U)$ is the set of all integers $\{x \text{ s.t. } |x| < 2^{N-1}\}$, then $\Psi_{<,U}$ has an equivalent constraint set over Z_p, shown below, so long as $p > 2^N$.

$$\mathcal{C}_{<,U} = \left\{ \begin{array}{ll} A_1. & M_0(1-M_0) = 0, \\ A_2. & M_1(2-M_1) = 0, \\ & \vdots \\ A_{N-1}. & M_{N-2}(2^{N-2} - M_{N-2}) = 0, \\ \\ B_1. & M_< \cdot (1-M_<) = 0, \\ B_2. & M_= \cdot (1-M_=) = 0, \\ B_3. & M_> \cdot (1-M_>) = 0, \\ B_4. & M_< + M_= + M_> - 1 = 0, \\ B_5. & M_< \cdot (X_1 - X_2 + 2^{N-1} - \sum_{i=0}^{N-2} B_i) = 0 \\ B_6. & M_= \cdot (X_1 - X_2) = 0 \\ B_7. & M_> \cdot (X_2 - X_1 + 2^{N-1} - \sum_{i=0}^{N-2} B_i) = 0 \\ B_8. & Y_1 - M_< = 0 \end{array} \right\}$$

Proof. We assume that X_1 and X_2 will always only be assigned values in U. Constraints B_1 through B_4 of $\mathcal{C}_{<,U}$ are satisfied precisely when one of $M_<$, $M_=$, and $M_>$ takes on the value 1 and the remaining two take on the value 0. Hence, say that $M_<$ is assigned 1. In this case, $\mathcal{C}_{<,U}$ can be seen as containing the constraint set shown in section C.1 of [21], which is satisfiable iff. $\theta^{-1}(X_1) < \theta^{-1}(X_2)$. Hence, $\mathcal{C}_{<,U}$ is satisfiable in this case iff. $\theta^{-1}(X_1) < \theta^{-1}(X_2)$ and $Y_1 = 1$. Similarly, when $M_>$ is assigned 1, $\mathcal{C}_{<,U}$ can be seen as containing the constraint set shown in section C.1 of [21] except where the roles of the input variables have been reversed, hence $\mathcal{C}_{<,U}$ is satisfiable in this case iff. $\theta^{-1}(X_2) < \theta^{-1}(X_1)$ and $Y_1 = 0$.

Finally, if $M_=$ is assigned 1, then the constraint set $\mathcal{C}_{<,U}$ boils down to the constraints

$$\left\{ \begin{array}{l} X_1 - X_2 = 0 \\ Y_1 - 0 = 0 \end{array} \right\}$$

, which are satisfiable iff. $X_1 = X_2$ and $Y_1 = 0$.

Hence, for some $x_1, x_2 \in U$ and some $y \in Z_p$, $\mathcal{C}_{<,U}(X = x, Y = y)$ is satisfiable only when $y = 0$ and $x_1 = x_2$ or $\theta^{-1}(x_1) > \theta^{-1}(x_2)$ or when $y = 1$ and $\theta^{-1}(x_1) < \theta^{-1}(x_2)$. But then $\mathcal{C}_{<,U}$ is equivalent to $\Psi_{<,U}$. □

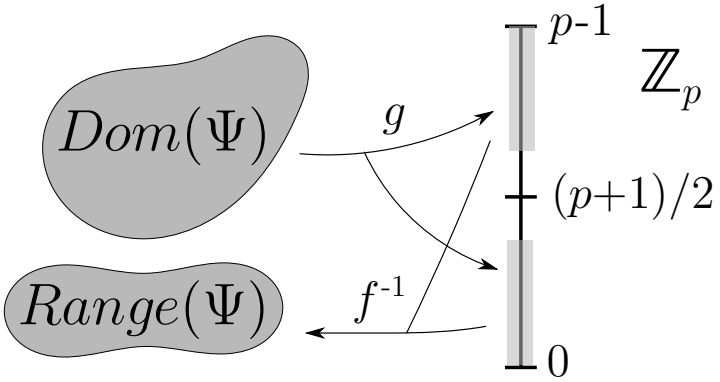

Figure 3—Diagram of mapping the domain and range of a computation Ψ to $\mathbb{Z}_p$, so that the tools of Ginger and Zaatar may be applied. Functions f and g are bijections and return output vectors in $\mathbb{Z}_p^n$, for some n. The case where $n = 1$ is shown, in general $\mathbb{Z}_p$ appears n times in the right-hand side of the above diagram.

5 Outsourcing computations not over $\mathbb{Z}_p$

For a computation Ψ which is not over $\mathbb{Z}_p$, there is (sometimes) a substitute computation for Ψ over $\mathbb{Z}_p$. Define $\tilde{\Psi}$ be a computation over $\mathbb{Z}_p$ with the property that $\Psi(X) = f^{-1}(\tilde{\Psi}(g(X)))$, where g is a bijection from the domain of Ψ to vectors whose components lie in a subset of $\mathbb{Z}_p$ and f is a bijection from the range of Ψ to vectors whose components lie in a subset of $\mathbb{Z}_p$. Rather than outsourcing Ψ, a client can outsource $\tilde{\Psi}$ on an input X mapped under the function g. Then, after applying a proof protocol to guarantee that the remote computer's output Y' is in fact $\tilde{\Psi}(g(X))$, the client can retrieve the value of $\Psi(X)$ by computing $f^{-1}(Y')$. This strategy is illustrated in Figure 3.

One example of this strategy is that the computation $\Psi_{<,U}$ of section 4.3 is a substitute for the computation that takes two integers x_1, x_2 in the set $U = [-2^{N-1}, 2^N]$ for some integer N and returns 1 if $x_1 < x_2$ and 0 otherwise. Here, we have set $f(x_1, x_2) = (\theta(x_1), \theta(x_2))$ and $g = \theta$, where θ is defined in Section 4.3. Thus f and g are bijections as long as $p > 2^N$.

Another use of this strategy is to produce substitutes for computations over rational numbers. Here we make use the mapping function θ_Q:

$$\begin{aligned} \theta_Q : \quad & \mathbb{Q} \to Z_p \\ & \theta_Q\left(\frac{a}{b}\right) = ab^{-1} \mod p \end{aligned}$$

Define the set of rational numbers $U = \{a/b : |a| < 2^{N_a}, b \in \{1, 2, 2^2, 2^3, ...2^{N_b}\}\}$. Assuming that p is a prime with at least $2(1 + \max\{N_a, N_b\})$ bits, θ_Q is a bijection from U to $\mathbb{Z}_p$ [21].

5.1 Constraints for arithmetic over rational numbers

Let x_1 and x_2 be two rational numbers in the set $U = \{a/b : |a| < 2^{N_a}, b \in \{1, 2, 2^2, 2^3, ...2^{N_b}\}\}$ whose sum and product is also in U, and let p be a prime with at least $2(1 + \max\{N_a, N_b\})$ bits. Then it turns out that $\theta_Q(x_1 + x_2) = \theta_Q(x_1) + \theta_Q(x_2)$ and that $\theta_Q(x_1 \cdot x_2) = \theta_Q(x_1) \cdot \theta_Q(x_2)$ [21]. Hence, the operations of addition and multiplication over such numbers x_1 and x_2 can be substituted with the operations of addition and multiplication over $\mathbb{Z}_p$, where $f(x_1, x_2) = (\theta_Q(x_1), \theta_Q(x_2))$ and $g = \theta_Q$.

5.2 Constraints for order comparison over rational numbers

We can produce a substitute computation for the computation Φ that takes two rational numbers x_1, x_2 in the set $U = \{a/b : |a| < 2^{N_a}, b \in \{1, 2, 2^2, 2^3, ...2^{N_b}\}\}$ for two integers N_a and N_b and returns 1 if $x_1 < x_2$ and 0 otherwise. Under these assumptions, it is the case that $x_1 - x_2 \in S$, where $S = \{a/b : |a| < 2^{N'_a}, b \in \{1, 2, 2^2, 2^3, ...2^{N_b}\}\}$ where $N'_a = N_a + N_b + 1$.

We can define $\Psi_{<Q,S}$ to be the computation over $\mathbb{Z}_p$ which takes two field elements x_1, x_2 and returns 1 if $\theta_Q^{-1}(x_1) < \theta_Q^{-1}(x_2)$ and 0 otherwise. Clearly, $\Psi_{<Q,S}$ is a substitute computation for Φ where we have defined $f(x_1, x_2) = (\theta_Q(x_1), \theta_Q(x_2))$ and $g = \theta_Q$. Constraints equivalent to $\Psi_{<Q,S}$ are shown below.

$$\mathcal{C}_{<Q,S} = \left\{ \begin{array}{ll} D_1. & A_0(1-A_0) = 0, \\ D_2. & A_1(2-A_1) = 0, \\ \vdots & \vdots \\ D_{N'_a}. & A_{N'_a-1}(2^{N'_a-1} - A_{N'_a-1}) = 0, \\ D_{N'_a+1}. & A - (p - 2^{N'_a}) - \sum_{i=0}^{N'_a-1} A_i = 0, \\ E_1. & B_0(1-B_0) = 0, \\ E_2. & B_1(1-B_1) = 0, \\ \vdots & \vdots \\ E_{N_b+1}. & B_{N_b}(1-B_{N_b}) = 0, \\ F_1. & \sum_{i=0}^{N_b} B_i - 1 = 0, \\ F_2. & B - \sum_{i=0}^{N_b} B_i \cdot \theta_Q^{-1}(1/2^i) = 0, \\ F_3. & C - A \cdot B = 0 \\ G_1. & M_< \cdot (1 - M_<) = 0, \\ G_2. & M_= \cdot (1 - M_=) = 0, \\ G_3. & M_> \cdot (1 - M_>) = 0, \\ G_4. & M_< + M_= + M_> - 1 = 0, \\ G_5. & M_< \cdot (X_1 - X_2 - C) = 0 \\ G_6. & M_= \cdot (X_1 - X_2) = 0 \\ G_7. & M_> \cdot (X_2 - X_1 - C) = 0 \\ G_8. & Y_1 - M_< = 0 \end{array} \right\}$$

Proof. Assume that p is a prime with at least $2(1 + \max\{N'_a, N_b\})$ bits.

Constraints G_1 through G_4 of $\mathcal{C}_{<Q,S}$ are satisfied whenever exactly one of $M_<$, $M_=$ and $M_>$ are assigned 1 and the remaining two are assigned 0. If $M_<$ is assigned 1, then the constraint set can be seen as containing the constraint set in section C.2 of [21], which is satisfiable iff $\theta_Q^{-1}(X_1) < \theta_Q^{-1}(X_2)$. Hence, in this case, $\mathcal{C}_{<Q,S}$ is satisfiable iff. $\theta_Q^{-1}(X_1) < \theta_Q^{-1}(X_2)$ and $Y_1 = 1$. If instead $M_>$ is assigned 1, then the constraint set can be seen as containing the contraint set in section C.2 of [21], except that the roles of the input variables have been reversed. Hence, in this case, $\mathcal{C}_{<Q,S}$ is satisfiable iff. $\theta_Q^{-1}(X_2) < \theta_Q^{-1}(X_1)$.

Finally, if $M_=$ is assigned 1, then the constraint set boils down to the following constraints

$$\left\{ \begin{array}{l} X_1 - X_2 = 0 \\ Y_1 - 0 = 0 \end{array} \right\}$$

, which are satisfiable iff. $X_1 = X_2$ and $Y_1 = 0$.

Hence, for some $x_1, x_2 \in \theta_Q(U)$ and some $y \in Z_p$, $\mathcal{C}_{<,U}(X = x, Y = y)$ is satisfiable only when $y = 0$ and $x_1 = x_2$ or $\theta_Q^{-1}(x_1) > \theta_Q^{-1}(x_2)$ or when $y = 1$ and $\theta_Q^{-1}(x_1) < \theta_Q^{-1}(x_2)$. But then $\mathcal{C}_{<,U}$ is equivalent to $\Psi_{<,U}$.

The assumption that p is a prime with at least $2(1 + \max\{N'_a, N_b\})$ bits was required to make use of results from Section C.2 of [21]. □

6 Compiling BFDL to constraints

6.1 Systematic construction of constraint sets

We now return to the problem of compiling the example BFDL code in Snippet 1 to constraints. Compilation begins by producing from BFDL code an intermediate representation of the computation in single assignment form. In *single assignment form*, a computation is represented as a sequence of assignment statements such that no variable is assigned no more than once.

The BFDL code in Snippet 1 is converted to single assignment form in Snippet 2. Note that the loop in line 9 of Snippet 1 has been unrolled, and that variables have been added to represent the state of variable `count` at each point in the computation. Furthermore, note that the if statement has been converted to single assignment form by creating a multiplexing statement which chooses between two values for `count` based on the value of the condition of the if statement.

Snippet 2 Assignments to count occurrences of an input integer in a list of $m = 10$ input integers

$cond_1 \leftarrow$ `X.list[0] != X.key`
$count_1 \leftarrow \Psi_{MUX}(cond_1, 0, 1)$
$cond_2 \leftarrow$ `X.list[1] != X.key`
$count_2 \leftarrow \Psi_{MUX}(cond_2, count_1, count_1 + 1)$
...
$cond_{10} \leftarrow$ `X.list[9] != X.key`
$count_{10} \leftarrow \Psi_{MUX}(cond_{10}, count_9, count_9 + 1)$

Each of the right hand sides in the assignments in Snippet 2 perform computations which we know how to convert to constraints, by Section 4. These constraint sets can be combined to form a constraint set for the entire computation, shown in Snippet 3. The only wrinkle when combining constraint sets in this way is that intermediate variables will need to be renamed to avoid naming collisions between the constraint sets.

Snippet 3 Constraint set which counts occurrences of an input integer in a list of $m = 10$ input integers

$$\mathcal{C} = \left\{ \begin{array}{rcl} M_1 \cdot (X_{\texttt{list[0]}} - X_{\texttt{key}}) - M_{cond_1} & = & 0 \\ (1 - M_{cond_1}) \cdot (X_{\texttt{list[0]}} - X_{\texttt{key}}) & = & 0 \\ M_{cond_1} \cdot (0 - 1) + 1 - M_{count_1} & = & 0 \\ M_2 \cdot (X_{\texttt{list[1]}} - X_{\texttt{key}}) - M_{cond_2} & = & 0 \\ (1 - M_{cond_2}) \cdot (X_{\texttt{list[1]}} - X_{\texttt{key}}) & = & 0 \\ M_{cond_2} \cdot (M_{count_1} - M_{count_1} + 1) + M_{count_1} - M_{count_2} & = & 0 \\ \ldots & & \\ M_{10} \cdot (X_{\texttt{list[9]}} - X_{\texttt{key}}) - M_{cond_{10}} & = & 0 \\ (1 - M_{cond_{10}}) \cdot (X_{\texttt{list[9]}} - X_{\texttt{key}}) & = & 0 \\ M_{cond_{10}} \cdot (M_{count_9} - M_{count_9} + 1) + M_{count_9} - Y_{count_{10}} & = & 0 \end{array} \right\}$$

The constraint set in Snippet 3 has 11 designated input variables X_i, subscripted according to their name in the single assignment form and 1 designated output variable Y_{count_1}. The remaining 29 intermediate variables M_i are subscripted according to either their name in the single assignment form or integers starting from 1 if they were created by expanding an operation into constraint sets.

A detail glossed over in the above description of compilation is that the original computation was over $\mathbb{Z}$, but the constraint sets are over $\mathbb{Z}_p$. In fact, the point at which operations over $\mathbb{Z}$ were converted to operations over $\mathbb{Z}_p$ is in the conversion from BFDL to single assignment form. Any constants written in a BFDL

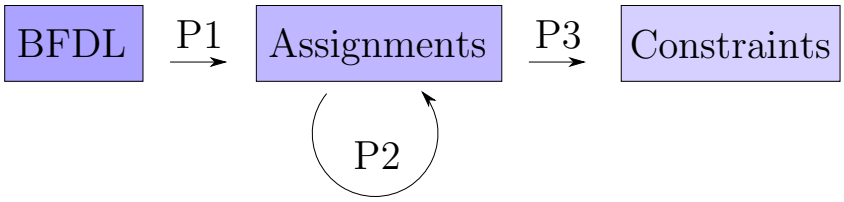

Figure 4—Diagram of the compiler pipeline used to compile a computation specified in BFDL to a constraint set for either the Ginger or Zaatar framework. The target framework for the compiler is determined before compiling begins. Task P1 is performed using the BFDL compiler described in Section 6.2 to compile a piece of BFDL code to a list of assignments. Task P2 uses various optimizations in the modified SFDL compiler to reduce the number of assignment statements in this list of assignments. The behavior of these optimizations changes depending on whether the target framework is Ginger or Zaatar. Task P3 is performed by the compiler backend and converts a list of assignments to a constraint set of the format accepted by the target framework.

program, say a constant -3, are replaced with their image under the function θ_Q (defined in Section 5), which for this constant would be $p - 3$. Any instances of floating point addition, multiplication, and order comparison in a BFDL program are treated as operations over rational numbers, which can be represented as constraints using the techniques of Section 5. Thus, the value of p must be chosen to be large enough so that θ_Q bijectively maps all possible intermediate values of any variable in the computation to $\mathbb{Z}_p$, when the computation is run on any valid input. The compiler produces alongside the output constraint set for a computation the smallest value of p for which the computation can be run safely in the Ginger and Zaatar protocols.

6.2 Compiler implementation

We implement a compiler, illustrated in Figure 4, which automates the translation described in Section 6.1. The task of converting BFDL code to assignments, P1, is performed by a modified version of the Fairplay compiler [13]. The task of optimizing the list of assignments, P2, refers to existing optimizations in the Fairplay compiler as well as some domain-specific optimizations we add, see below. The BFDL compiler contains 5294 lines of Java, as opposed to the Fairplay compiler's 3886. The task of converting assignments to constraints, P3, is performed by a new Python program of 1105 lines.

6.3 Robust compilation for large computations

When the size of the computation being performed becomes very large, the list of assignments being produced may be too large to hold in memory. One modification of the BFDL compiler is that it streams the list of assignments to disk as it is generated. This dramatically reduces the memory use required to run the compiler on large computations. An additional measure could be to use a structured database to hold the compiled representation as in [12], and this would allow the compiler to scale up to even larger computations.

6.4 Optimizations

The work performed by the client and the remote computer during the Ginger and Zaatar protocols scales with the number of variables, and constraints in a constraint set. We present a number of optimizations which the compiler performs in an attempt to reduce the number of variables or constraints in the compiled constraint set.

The Fairplay compiler removes dead code and redundant statements from the output list of assignments, and it evaluates constant expressions where possible. These optimizations, which are global in the Fairplay compiler, are replaced with locally optimizing equivalents in the BFDL compiler. Implementing these optimizations in the streaming output model used by the compiler requires some care.

First, the BFDL compiler sequentially expands program structures in the BFDL code to produce a list

$$
\begin{array}{l}
M_4 \leftarrow M_1 + M_3 \\
M_5 \leftarrow M_2 + M_3 \\
M_6 \leftarrow M_4 \cdot M_5 \\
\qquad\downarrow \\
M_6 \leftarrow (M_1+M_3)\cdot(M_2+M_3)
\end{array}
\quad\Bigg|\quad
\begin{array}{l}
M_4 \leftarrow M_1 \cdot M_3 \\
M_5 \leftarrow M_2 + M_3 \\
M_6 \leftarrow M_4 + M_5 \\
\qquad\downarrow \\
M_6 \leftarrow (M_1)\cdot(M_3)+M_2+M_3
\end{array}
\quad\Bigg|\quad
\begin{array}{l}
M_3 \leftarrow M_2 - M_1 \\
M_4 \leftarrow M_3 \cdot M_3 \\
\qquad\downarrow \\
M_4 \leftarrow (M_2-M_1)\cdot(M_2-M_1)
\end{array}
$$

Figure 5—Combining simple assignment statements to produce a single assignment statement which can be represented in a single constraint. The extent to which the compiler can consolidate statements in this way is limited by the constraint format accepted by the target protocol for compilation, which is either Zaatar or Ginger. Left, Center: the arguments to an addition or multiplication operation are inlined, resulting in a computation with a single assignment which can be expanded to a single constraint. Right: The variable M_3 is inlined *twice* into an assignment for M_4. By default, the compiler will inline an assignment only if the assignment becomes dead code as a result. When an assignment is referenced multiple times, such as in the right-most example above, the compiler will inline an expression up to a maximum number of times indicated by a compiler parameter.

of assignment statements. Specifically, the compiler visits each node on the parse tree of the BFDL code in-order, and for each node emits a sequence of assignment statements equivalent to that node.

Next, the compiler gathers profiling information on the output list of assignment statements. The compiler produces a reversed list of the assignments (using the efficient `tac` program available on many Linux distributions). In the reversed ordering, the first assignment which references a given variable ends the scope of, or "kills," the referenced variable. By iterating over the reversed list of assignment statements, the compiler can efficiently produce a list (in reverse order) of tuples (i, j, k) where i is some variable, j is the assignment which kills variable i (if i is killed), and k is the number of times variable i is referenced. This output list of tuples is reversed (again with the `tac` program) to produce a list of profiling data (in the correct order) which is used during optimization.

Dead, redundant, and constant assignments. The compiler performs dead code elimination in a streaming fashion by iterating over the list of assignments and removing those which, according to the profiling data, are never referenced.

The compiler also retains a buffer of the past N statements written to the disk in this way. Only assignment statements in this window of statements are visible to the optimization routines. The compiler also retains a table of values for each live variable which can be referenced at the current node of the parse tree being visited, if that value is a constant, or a simple reference to another variable. Hence, the compiler can make use of the buffer of the past N statements and the values of some of the live variables when applying optimizations. Redundant assignments can be removed by efficiently checking whether the statement under investigation is in the past N statements, using a hash table lookup. Constant expressions are simplified by attempting to replace variable references in every assignment with their corresponding constant values (if available).

Combining gates By default, the BFDL compiler outputs one assignment statement for every primitive operation in the BFDL code, i.e. one for each $+$, $\cdot$, $!=$, $<$, and Ψ_{MUX} operation. However, we can represent an assignment statement of the form $Z_i \leftarrow p_A(Z) \cdot p_B(Z) + p_C(Z)$, where p_A, p_B, and p_C are polynomials of total degree 1, in a single constraint in the Zaatar format. Additionally, any assignment statment of the form $Z_i \leftarrow P_2(Z)$ where P_2 is any polynomial of total degree 2 can be represented using a single constraint in the Ginger format.

We introduce an optimization to the BFDL compiler which consolidates arithmetic operations, so that the resulting assignment statement can be expanded in a single constraint. This optimization is shown in Figure 5. Only statements which all simultaneously exist in the buffer of the past N statements can be combined in this way.

6.5 Improvements to the type system

The BFDL compiler associates with each variable its declared type, which appears in the BFDL code, and its actual type, which is determined by the expressions assigning a variable its value. The input variables to the main function, `output`, have actual type equal to their declared type. The compiler can propagate types through any of the primitive operations ($+$, $\cdot$, $!=$, $<$) in a computation, to produce the actual type of the remaining variables. If at any time the declared type of a variable is not a subtype of the actual type of that variables value, a type error is thrown. The type system uses conservative rules to determine the type of an expression. In the case that the programmer can prove that a particular expression has a particular type, the programmer can make use of the special meta-function `reinterpret_cast` to re-type an expression.

7 Evaluation

Benchmark computations We implemented 6 benchmark computations in BFDL, which we use to evaluate the performance of the compiler. We implement (a) Floyd-Warshall all pairs shortest paths [7], (b) DNA sequence global alignment [17], (c) the fannkuch benchmark [1], (d) insertion sort, (e) matrix multiplication, and (f) Partitioning Around Medoids (PAM) clustering [23].

Details of testbed We run All experiments on a computer equipped with an Intel Xeon processor E31270 3.40GHz with 15.6 GB of RAM. All running time data was collected by repeating the trial at least three times; the standard deviation of the measurement was always less than 5% of the mean.

Analysis of compiler improvements. Figure 6 shows the effectiveness of the gate combining optimization. In general, this computation offers at least a 2x reduction in the size of the list of assignment statements.

Figure 7 shows the effect of increasing the size of the compiler window on the size of the list of assignment statements. A lower buffer size uses less memory during compilation. For buffer sizes above 1024, the results are within 1% of the unlimited size case.

End-to-end running time Figure 8 shows the running times of the compiler to produce constraint sets for the benchmark computations, and the size of the produced constraint sets.

8 Conclusion

The compiler presented in this paper successfully produces constraint sets equivalent to arguably useful computations at realistic input sizes. The compiler allows a user to easily apply the Ginger and Zaatar protocols to outsource some computation Ψ in a simple way. The user writes the computation Ψ in BFDL, runs the compiler on the BFDL code to produce a constraint set equivalent to Ψ using either the Ginger or Zaatar format specification, and then outsources the constraint set produced using the Ginger or Zaatar protocols.

9 Acknowledgements

I would like to thank Dr. Mike Walfish and doctoral candidate Srinath Setty for their help on this project.

[1] See http://www.haskell.org/haskellwiki/Shootout/Fannkuch.

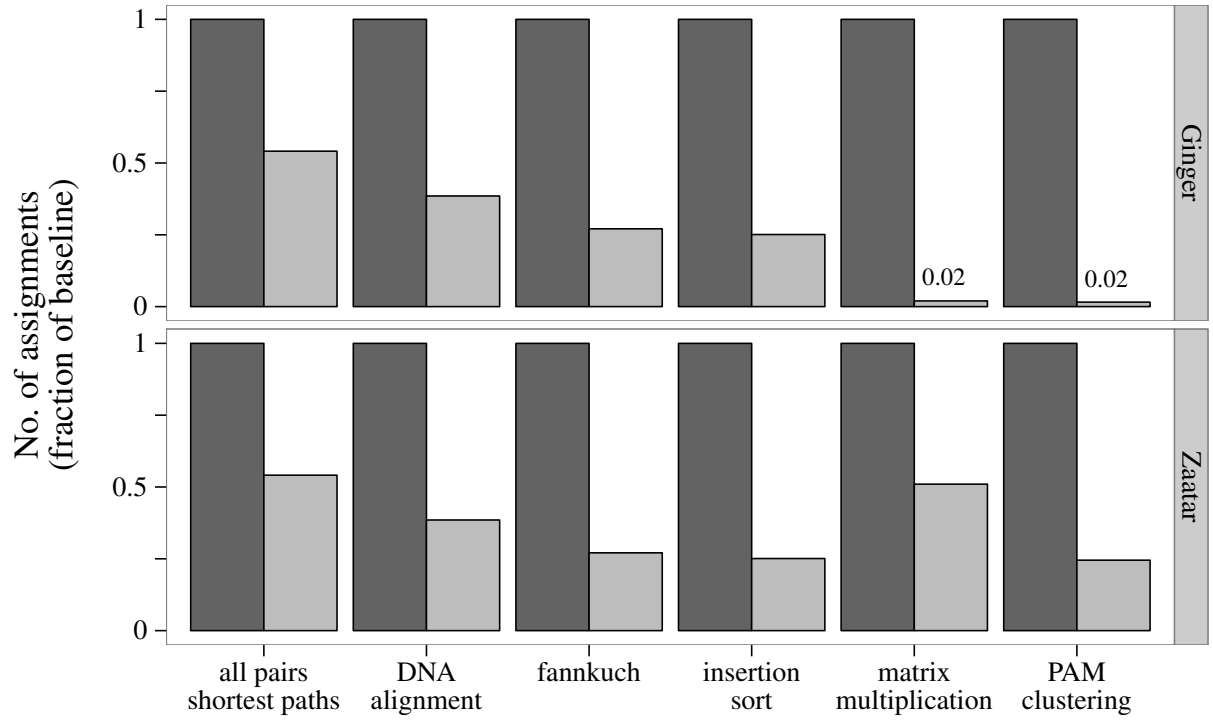

Figure 6—Number of assignment statements produced by the compiler when gate combining is disabled (dark bar) or enabled (light bar) for the benchmark computations, and when the compiler is outputting Ginger (top graph) or Zaatar (bottom graph) format constraints. The computations are run at the following input sizes: Floyd-Warshall's all pairs shortest paths algorithm on a graph with $m = 25$ nodes, DNA alignment on two $m = 200$ nucleotide sequences, the Fannkuch benchmark on $m = 100$ permutations of the numbers $\{1, 2, ...13\}$, insertion sort on an array of $m = 256$ integers, matrix multiplication on $m \times m$ matrices where $m = 100$, and PAM clustering on $m = 20$ data points of dimension 100.

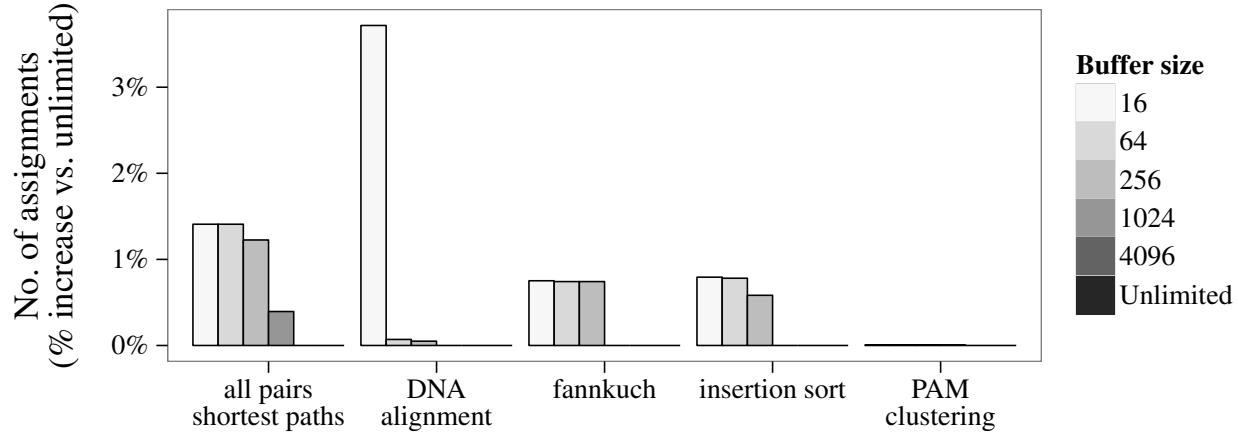

Figure 7—Increase in the size of the list of assignments produced by the compiler (Zaatar output mode) for various choices of compiler window size. The computations are run at the following input sizes: Floyd-Warshall's all pairs shortest paths algorithm on a graph with $m = 10$ nodes, DNA alignment on two $m = 50$ nucleotide sequences, the Fannkuch benchmark on 10 permutations of the numbers $\{1, 2, ...13\}$, insertion sort on an array of $m = 64$ elements, and PAM clustering on $m = 5$ data points of dimension 100. Results for Ginger are similar.

Ginger						
Computation Ψ	Compile (s)	Optimize (s)	Constraints (s)	Total (s)	$\|w\|$	K
DNA alignment	221	518	213	953	1.8e+12	4.7e+06
fannkuch	80	441	198	721	1.4e+12	4.6e+06
all pairs shortest paths	17	57	132	206	4.7e+11	2.7e+06
insertion sort	24	82	253	360	1.6e+12	5.2e+06
matrix multiplication	45	208	46	301	2.5e+09	1.1e+06
PAM clustering	89	525	206	822	3e+11	3.9e+06
Zaatar						
Computation Ψ	Compile (s)	Optimize (s)	Constraints (s)	Total (s)	$\|w\|$	$Nz(A,B,C)$
DNA alignment	218	559	201	979	3.5e+06	5.7e+06
fannkuch	80	487	202	770	3.3e+06	5.8e+06
all pairs shortest paths	18	62	106	187	1.7e+06	3.4e+06
insertion sort	25	86	201	312	3.4e+06	6.6e+06
matrix multiplication	46	319	150	516	2.1e+06	4e+06
PAM clustering	95	497	225	819	3.4e+06	6.7e+06

K: # of non-zero terms in Ginger constraint set for Ψ

$Nz(A,B,C)$: # of non-zero terms in Zaatar constraint set for Ψ

$|w|$: # of entries in the vector w encoding a proof (§2.1)

Figure 8—Compile time and constraint set sizes for the benchmark applications, when the compiler is run in either Ginger or Zaatar mode, with Ginger results on top and Zaatar results on bottom. Input sizes to all computations are the same as those in Figure 6. Compile: time to produce the initial compiled circuit. Optimize: time to optimize the circuit (dead code elimination, concise gate introduction). Constraints: time to convert the circuit description of the computation to a constraint set equivalent to the computation. Total: sum of the preceding three columns. For an analysis of how K, $Nz(A,B,C)$ and $|w|$ affect the runtime of the Ginger and Zaatar protocols, see [19].

References

[1] S. Alsouri, Ö. Dagdelen, and S. Katzenbeisser. Group-based attestation: Enhancing privacy and management in remote attestation. In *TRUST*, 2010.

[2] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. SETI@home: An experiment in public-resource computing. *CACM*, 45(11):56–61, Nov. 2002.

[3] S. Arora, C. Lund, R. Motwani, M. Sudan, and M. Szegedy. Proof verification and the hardness of approximation problems. *J. of the ACM*, 45(3):501–555, May 1998.

[4] S. Arora and S. Safra. Probabilistic checking of proofs: a new characterization of NP. *J. of the ACM*, 45(1):70–122, Jan. 1998.

[5] M. Castro and B. Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Trans. on Computer Sys.*, 20(4):398–461, 2002.

[6] A. Chiesa and E. Tromer. Proof-carrying data and hearsay arguments from signature cards. In *ICS*, 2010.

[7] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.

[8] G. Cormode, M. Mitzenmacher, and J. Thaler. Practical verified computation with streaming interactive proofs. In *ITCS*, 2012. Earlier version: http://arxiv.org/abs/1105.2003, May 2011.

[9] R. Gennaro, C. Gentry, B. Parno, and M. Raykova. Quadratic span programs and succinct NIZKs without PCPs. Apr. 2012. Cryptology eprint 215.

[10] A. Haeberlen, P. Kouznetsov, and P. Druschel. PeerReview: Practical accountability for distributed systems. In *SOSP*, 2007.

[11] Y. Ishai, E. Kushilevitz, and R. Ostrovsky. Efficient arguments without short PCPs. In *Conference on Computational Complexity (CCC)*, 2007.

[12] B. Kreuter, abhi shelat, and C. hao Shen. Billion-gate secure computation with malicious adversaries. In *USENIX Security*, Aug. 2012.

[13] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay—a secure two-party computation system. In *USENIX Security*, 2004.

[14] D. Malkhi and M. Reiter. Byzantine quorum systems. *Distributed Computing*, 11(4):203–213, 1998.

[15] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An execution infrastructure for TCB minimization. In *EuroSys*, 2008.

[16] A.-R. Sadeghi, T. Schneider, and M. Winandy. Token-based cloud computing: secure outsourcing of data and arbitrary computations with lower latency. In *TRUST*, June 2010.

[17] P. H. Sellers. On the theory and computation of evolutionary distances. *SIAM J. of Appl. Math.*, 26(4):787–793, 1974.

[18] S. Setty, A. J. Blumberg, and M. Walfish. Toward practical and unconditional verification of remote computations. In *HotOS*, 2011.

[19] S. Setty, B. Braun, V. Vu, A. J. Blumberg, B. Parno, and M. Walfish. Resolving the conflict between generality and plausibility in verified computation, Oct. 2012.

[20] S. Setty, R. McPherson, A. J. Blumberg, and M. Walfish. Making argument systems for outsourced computation practical (sometimes). In *NDSS*, Feb. 2012.

[21] S. Setty, V. Vu, N. Panpalia, B. Braun, A. J. Blumberg, and M. Walfish. Taking proof-based verified computation a few steps closer to practicality (extended version). In *USENIX Security*, Aug. 2012.

[22] J. Thaler, M. Roberts, M. Mitzenmacher, and H. Pfister. Verifiable computation with massively parallel interactive proofs. In *USENIX HotCloud Workshop*, June 2012.

[23] S. Theodoridis and K. Koutroumbas. *Pattern Recognition, Third Edition.* Academic Press, Inc., 2006.

10 Appendix: BFDL code for benchmark computations

10.1 BFDL code for Floyd-Warshall's all pairs shortest paths

```
program f_w_apsp{
  //Constants
  const m = 50;
  const Infinity = 0x7FFFFFFF; //Maximum value of a 32bit signed int

  //types
  type dist_t = int<32>; //The type for weights and shortest path lengths

  //Input - a square matrix holding the weights of the directed edges of
  //the graph. W[i][j] is the weight of edge (i,j), or Infinity if
  //the edge does not exist. The graph must not have any negative weight cycles.
  type Input = struct{dist_t[m][m] W};

  //Output - a square matrix holding the shortest path lengths between
  //every pair of nodes in the graph.
  //We assume that the shortest path between any two nodes will have a
  //length which fits in a 32 bit signed integer.
  type Output = struct {dist_t[m][m] D};

  //functions
  function dist_t min(dist_t a, dist_t b){
    if (a < b){
      min = a;
    } else {
      min = b;
    }
  }

  function Output output (Input X){
    var int i;
    var int j;
    var int k;
    var dist_t[m][m] D;
    var dist_t[m][m] Dnext;

    //Initialize the matrix D to be the weights of the graph
    D = X.W;

    //These for loops follow the structure of the description of the Floyd
    //Warshall algorithm in the CLR Intro to Algorithms textbook.
    for(k = 0 to m-1) {
      for(i = 0 to m-1) {
        for(j = 0 to m-1) {
          Dnext[i][j] = D[i][j];

          //If D[i][k] and D[k][j] are both finite, then the path that goes
          //from i -> k -> j is an option.

          //Strictly speaking, it is possible for the shortest path between
          //two nodes to exceed the bounds of 32 bit signed integers.
          //However, we explicitly assumed that this does not occur in the
          //problem statement - hence a reinterpret_cast here is valid.
          if (D[i][k] != Infinity & D[k][j] != Infinity){
            Dnext[i][j] = min(
                  Dnext[i][j],
                  reinterpret_cast(dist_t, D[i][k] + D[k][j]));
```

```
          }
        }
      }
      D = Dnext;
    }

    output.D = D;
  }
}
```

10.2 BFDL code for DNA alignment

```
program DNA_alignment{
  //Constants
  const m = 50;
  const n = 50; //n >= m

  //types
  type char = int<16>;
  type Input = struct{char[m] A, char[n] B};

  //If the LCS is shorter than n characters, the result will be terminated
  //with the zero character, i.e. a C-style string is returned.
  type Output = struct {char[n] LCS};

  //functions
  //Returns true iff. 0 <= i < a and 0 <= j < b
  function boolean checkIndex2(int i, int j, int a, int b){
    checkIndex2 = (0 <= i) & (i < a) & (0 <= j) & (j < b);
  }

  //main method, called "output"
  function Output output (Input X){
    var int i;
    var int ir;
    var int j;
    var int jr;
    var int[m] A;
    var int[n] B;
    //Dynamic programming memo
    var int[m][n] LL;
    //Hold choices made at each step, for use when backtracking
    var int[m][n] choices;
    //Used when backtracking
    var boolean inserted;
    var int iPtr;
    var int jPtr;
    var int diag;
    var int down;
    var int right;
    var char[m] LCS; //min(m,n) = m

    A = X.A;
    B = X.B;
    //Go backwards from i = m-1 downto 0
    for(ir = 0 to m-1){
      i = m-1-ir;
      for(jr = 0 to n-1){
        j = n-1-jr;
```

```
    if (A[i] == B[j]){
      if (checkIndex2(i+1,j+1,m,n)){
        diag = LL[i+1][j+1];
      } else {
        diag = 0;
      }
      //Diagonal jump
      LL[i][j] = 1 + diag;
      choices[i][j] = 0;
    } else {
      if (checkIndex2(i+1,j,m,n)){
        down = LL[i+1][j];
      } else {
        down = 0;
      }
      if (checkIndex2(i,j+1,m,n)){
        right = LL[i][j+1];
      } else {
        right = 0;
      }
      //Assertion: down and right differ by at most 1
      if (down == right + 1){
        //Jump down
        LL[i][j] = down;
        choices[i][j] = 1;
      } else {
        //Jump right if down == right or right == down + 1.
        LL[i][j] = right;
        choices[i][j] = 2;
      }
    }
  }
}

//Construct LCS, allowing it to have intermittent zero characters
iPtr = 0;
jPtr = 0; //Pointers to where in LL we are with respect to backtracking
for(i = 0 to m-1){
  LCS[i] = 0; //If A[i] is not in the LCS, this remains 0.
  for(j = 0 to n-1){
    if ((i == iPtr) & (j == jPtr)){ //Loop until we meet up with the iPtr and jPtr
      if (choices[i][j] == 0){ //we made a diagonal jump here
        LCS[i] = A[i];
        iPtr = iPtr + 1;
        jPtr = jPtr + 1;
      } else {
        if (choices[i][j] == 1){//jump down
          iPtr = iPtr + 1;
        } else { //jump right
          jPtr = jPtr + 1;
        }
      }
    }
  }
}

//Now move any string terminator (\0) characters in LCS to the end ala
//insertion sort
for(i = 1 to m-1){
```

```
      inserted = false;
      for(j = 0 to i-1){
        if ((LCS[j] == 0) & !inserted){
          //Swap LCS[j] and LCS[i].
          LCS[j] = LCS[i];
          LCS[i] = 0;
          inserted = true;
        }
      }
    }

    output.LCS = LCS;
  }
}
```

10.3 BFDL code for fannkuch benchmark

```
program fannkuch{
  //Constants
  const m = 13; //Fn only supplied for m <= 13.
  const Fn = {0, 0, 1, 2, 4, 7, 10, 16, 22, 30, 38, 51, 65, 80}; //the n'th Fannkuch numbers
  const log2_m = 5;
  const L = 5; //How many permutations to run.

  //types
  type Input = struct{int<log2_m>[m] a};
  type Output = struct {int flips};

  //main method, called "output"
  function Output output (Input X){
    var int i;
    var int j;
    var int k;
    var int u;
    var int iter;
    var int flipNum;
    var int[m] a;
    var int[m] b;
    var int flips;
    var int winningK;
    var int aWinningK;
    var int winningU;
    var int aWinningU;

    output.flips = 0;

    for(j = 0 to L-1){
      a = X.a;
      b = a;
      flips = 0;
      for(i = 0 to Fn[m]-1) {
        if (a[0] != 1){
          flips = flips + 1;
          for(flipNum = 2 to m) {
            if (a[0] != flipNum){
            } else {
              for(j = 0 to flipNum-1){
          b[j] = a[flipNum - 1 - j];
              }
            }
          }
          a = b;
        }
      }
      if (flips > output.flips){
        output.flips = flips;
      }

      //Advance X.a to the next lexographic permutation.
      //Find the largest index k such that a[k] < a[k+1].
      winningK = -1;
      aWinningK = -1;
      for(k = 0 to m-2){
        if (X.a[k] < X.a[k+1]){
```

```
          winningK = k;
          aWinningK = X.a[k];
        }
      }
      if (winningK != -1){
        //Find the largest index u such that a[winningK] < a[u].
        //Since k >= 0 and u > k, u >= 1.
        for(u = 1 to m-1){
          if (aWinningK < X.a[u]){
            winningU = u;
            aWinningU = X.a[u];
          }
        }

        //Swap a[k] and a[u].
        for(i = 0 to m-1){
          if (i == winningK){
            X.a[i] = aWinningU;
          }
          if (i == winningU){
            X.a[i] = aWinningK;
          }
        }

        //Reverse the sequence a[k+1] up to a[m-1]
        b = X.a;
        for(i = 0 to m-1){
          if (i == winningK){
            for(u = i+1 to m-1){
              X.a[u] = b[i+m-u]; //m-1-(u-(i+1)) = m-1-(u-i-1)
            }
          }
        }
      }
    }
  }
}
```

10.4 BFDL code for insertion sort

```
program insertion_sort{
  //Constants
  const m = 512;

  //types
  type Input = struct{int<32>[m] a};
  type Output = struct {int<32>[m] Ya};

  //functions

  //main method, called "output"
  function Output output (Input X){
    var int i;
    var int j;
    var int[m] a;
    var int[m] b;
    var boolean inserted;

    a = X.a;
    b = a;

    for(i = 0 to m-1) {
      inserted = false;
      //Try to insert a[i] into a[0...i-1]
      for(j = 0 to i-1) {
        if (b[i] < b[j]) {
          if (inserted) {
            b[j] = a[j-1];
          } else {
            b[j] = a[i];
            inserted = true;
          }
        } else {
          b[j] = a[j];
        }
      }
      if (inserted) {
        b[i] = a[i-1];
      } else {
        b[i] = a[i];
      }
      //update list
      a = b;
    }

    output.Ya = a;
  }
}
```

10.5 BFDL code for matrix multiplication

```
program matrixmult{
  //Constants
  const m = 10;

  //Types
  type Output = struct {int[m][m] Y};

  //Functions
  function Output output (int<32>[m][m] Xa, int<32>[m][m] Xb){
    var int i;
    var int j;
    var int k;
    for (i = 0 to m-1) {
      for (j = 0 to m-1) {
        output.Y[i][j] = 0;
        for(k = 0 to m-1) {
          output.Y[i][j] = output.Y[i][j] + Xa[i][k] * Xb[k][j];
        }
      }
    }
  }
}
```

10.6 BFDL code for PAM clustering

```
program pam_clustering{
  //Constants
  const k = 2; //Number of clusters
  const d = 100; //Dimensionality of datapoints
  const m = 5; //Number of datapoints
  const L = 5; //Number of times to run the swap phase (Each iteration tests (m-1)*k configurations)

  //types

  //Input - An m by d matrix A, such that A[i][j] is the j-th coordinate
  //of the i-th datapoint
  type Input = struct{float<32, 5>[m][d] A};

  //Output - A vector c of m elements, such that c[i] indicates which
  //cluster,
  //numbered 1 through k, datapoint i is in.
  type Output = struct {int[m] c};

  function float getDistance(float[d] a, float[d] b){
    var float diff;
    var int i;
    //Squared euclidean distance
    getDistance = 0;
    for(i = 0 to d-1){
      diff = a[i] - b[i];
      getDistance = getDistance + diff * diff;
    }
  }

  function float getPointCost(float[d] a, float[k][d] medoids){
    var float cndPointCost;
    var int i;

    getPointCost = getDistance(a, medoids[0]);
    for(i = 1 to k-1){
      cndPointCost = getDistance(a, medoids[i]);
      if (cndPointCost < getPointCost){
        getPointCost = cndPointCost;
      }
    }
  }

  function float getCost(float[m][d] A, float[k][d] medoids){
    var float cost;
    var int i;
    cost = 0;
    for(i = 0 to m-1){
      cost = cost + getPointCost(A[i], medoids);
    }
    getCost = cost;
  }

  //main method, called "output"
  function Output output (Input X){
    var int i;
    var int j;
    var int u;
```

```
var int iter;
var float[m][d] A;
var int[m] c;
var float[k][d] medoids; //The current medoids
var boolean[m] isMedoid; //isMedoid[i] is true if the i-th dp is a medoid
var float optDist;
var float newDist;
var float newCost;
var float currentCost;
var float testDistance;
var float[d] backup;
var float[m] pointCosts;
var float[m] newPointCosts;

A = X.A;

//Randomly choose medoids
for(i = 0 to m-1){
  isMedoid[i] = false;
}
//Currently, just set the medoids to be the first k datapoints
for(i = 0 to k-1){
  isMedoid[i] = true;
  medoids[i] = A[i];
}

//Initial cost
currentCost = getCost(A, medoids);

//Swap phase
for(iter = 0 to L-1) {
  for(i = 0 to k-1){
    //Swap medoid i with a nonmedoid point in the best way,
    //or leave it as is if no better point found
    for(j = 0 to m-1){
      if (j != i){
        if (!isMedoid[j]){
          //This is a possible medoid - nonmedoid swap.
          //Compute the change in cost as a function of the swap
          backup = medoids[i];
          medoids[i] = A[j];
          newCost = getCost(A, medoids);

          //Is it an improvement?
          if (newCost < currentCost){
            isMedoid[j] = true;
            isMedoid[i] = false;
            currentCost = newCost;
          } else {
            medoids[i] = backup;
          }
        }
      }
    }
  }
}

//Classify points (Note - a medoid may end up in a different
//cluster than its own, if two medoids have the same coordinates)
```

```
    for(i = 0 to m-1){
      c[i] = 0;
      optDist = getDistance(medoids[0], A[i]);
      for(j = 1 to k-1){
        newDist = getDistance(medoids[j], A[i]);
        if (newDist < optDist){
          optDist = newDist;
          c[i] = j;
        }
      }
    }

    output.c = c;
  }
}
```