Exploring Type Transformations On Vector Types For Generating Code For FPGAs

Ross Meikleham
School of Computing Science
University of Glasgow
1107023m@student.gla.ac.uk

I. Abstract

In the past scientific Fortran code was mainly written sequentially; programmers could rely on faster processors being produced in the future which would in turn result in a speedup of these programs. However, in the past decade processor speeds have started to stagnate. These scientific programs typically have a large amount of data parallelism that is able to be exploited. This can result in very large performance gains when converted to run on heterogeneous platforms such as FPGAs or GPUs. However this typically requires heavy modification of older code, and in-depth knowledge of parallel programming. This project focuses on exploring type transformations with dependent types to automatically generate correct by construction program variants, and selecting one of these variants which results in the "best" results with respect to a given cost model.

II. Problem Description

We want to exploit the parallelism of sequential scientific Fortran code without having to heavily modify the code itself. There are a multitude of ways to do this but we focus on using FPGAs (Field Programmable Gate Arrays).

A FPGA is an integrated circuit that contains an array of programmable logic blocks and interconnects which wire the blocks together. These logic blocks operate in parallel and are programmed using a Hardware Description Language(HDL) such as Verilog or VHDL. Currently designing programs in a HDL is a tedious and error prone process, especially as programs start to become more complex [1]. However, code that is highly parallelizable can result in significant performance gains when synthesized to run on a FPGA.

The main focus in research on synthesizing sequential code to HDL has been on creating compilers which generate HDL code from C, C++, and MATLAB rather than Fortran. The goals of many of these compilers are focused on reducing programming time and software maintenance issues rather than optimising for performance. Often, optimizing for space and architecture optimizations requires modifying of the original source code, and are not provided by the compilers themselves [2].

There can be a multitude of ways to parallelize certain sequential code, for example consider the Fortran95 program in listing 1 which adds 1 to each element of a 12 element array.

```
program pd_example

integer, dimension (1 : 12) :: A, C

A = (/ (I, I = 1, 12) /)
C = A + 1

print *, C

end program pd_example
```

Listing 1. F95 Loop

Considering that each array element is independent of one another during the adding operation it's entirely possible to perform the addition of each element in parallel. We can tune how much we parallelize the array addition using factors of the size of the array. In this case the factors of 12 are 1, 2, 3, 4, 6, and 12.

For each factor **f** of 12 there exists a factor **f'** such that **f** * **f'** = 12. Let **f** be the number of circuits and **f'** be the number of additions performed by each component, increasing **f** increases the degree of parallelization but also increases the total area used on a FPGA board.

Given a cost model and a program we want to optimise a program with respect to the given cost model, this involves generating equivalent variants of the given program, and then selecting the program which gives the best result from the given model to synthesize.

III. Project Aims

The main aim of this project is to design a framework which acts as an intermediate stage in HDL synthesis from Fortran code onto an FPGA. We restrict the framework to using a pure functional approach; representing a program as vector structures[3] operated on with a combination of element processing functions with map, fold, and zip operations.

This approach allows us to perform type transformations on the computations to tune the amount of parallelism whilst still preserving the results of the computations.

We aim to model the framework as an EDSL (Embedded Domain Specific Language); this saves the tedious work of having to implement a custom DSL (Domain Specific Language) for this specific purpose, allowing us to use the language constructs of the host language used to implement the framework. This also has the added benefit of being able to interpret these programs on a typical desktop computing platform which can be useful for testing purposes before transforming into a HDL.

We also aim to implement a rudimentary cost model which should be able to be replaced with more complex cost models in the future. Each given function has a given performance and space cost, represented as integers. The lower the integer the lower the performance cost, and vice versa. The same goes for the space cost. For example given a function with performance cost **p**, and space cost **s** which is being mapped on over a 1 dimensional vector with **n** elements; performing the map entirely in parallel would incur a performance cost of **p**, and space cost of **n * s**. Performing the map sequentially would incur a performance cost of **n * p**, and a space cost of **s**. A higher amount of parallelization increases the space cost but reduces the performance cost, a smaller amount of parallelization reduces the space cost but increases the performance cost.

Another aim is to also be able to create an interface for converting the optimal program for the given cost model into an Intermediate Representation(IR) format which is then itself processed and converted to a HDL.

The focus of this project isn't on translating Fortran code into the EDSL, or directly translating the optimal program into a HDL.

IV. Background Reading

A. High Level Synthesis Tools

1) Handel-C: Handel-C[4] is a subset of C with added non-standard extensions designed to target FPGAs. Some features of Handel-C include co-routines, concurrency and communication. The aim of Handel-C is to be an interface to hardware that's similar to a traditional imperative programming language, while allowing the programmer to exploit the natural concurrency provided in a hardware environment [5].

Synchronous communication between concurrent processes can be performed via channels. Communication can also be performed between concurrent processes using a mechanism called signals, a process assigns a value to a signal which can be read by another signal in the same clock cycle.

Parallelism in Handel-C has to be explicitly defined. There is also no support for limiting shared access to variables; it's possible for two concurrent processes to write to the same register at the same time, which is undefined. It's also the programmers responsibility to ensure that shared access to variables is handled correctly.

2) SystemC: SystemC[6] was originally designed as a modelling language for modelling hardware in C++. It provides a set of modelling constructs that are similar to Verilog or VHDL. SystemC is made up a set of C++ classes and macros which allow designers to model systems at several abstraction levels and connect components together from different abstraction levels [7]. SystemC is compatible with the C++, this allows for simulations to be created by compiling the program with a C++ compiler.

SystemC allows programmers to take advantage of object orientated programming for constructing components. This has many benefits; for example being able to separate independent components, composing different components, and reusing them [8].

Like Handel-C, SystemC uses channels and signals to communicate between concurrent processes.

3) Xilinx AccelDSP: AccelDSP is a high level synthesis design tool which generates a hardware module that can be run on a Xilinx FPGA from a MATLAB specification [9]. It has the ability to optimize floating point operations by dynamically estimating the needed bit width and automatically converting floating point to fixed point.

AccelDSP partially allows for modifying space requirements by including features such as (partial) loop and matrix multiplication unrolling, pipelining and memory mapping. However, AccelDSP can only be used for streaming data applications and the MATLAB code needs to be rewritten manually into a streaming loop [2]. Also AccelDSP is locked into only being able to work with Xilinx FPGAs, and cannot be used in FPGAs from other vendors.

4) Synphony C Compiler: The Symphony C Compiler[10] generates RTL implementations for ASICs and FPGAs from a subset of C/C++. Its focus is on parallelizing single-threaded sequential code for synthesis. Most optimization options are set in the source code through the use of pragmas [2]. Other options such as the target platform and clock rate are set through the GUI or the compilation script. The compiler also attempts to perform optimizations by predicting area and timing results.

5) Catapult C: Catapult C[11] accepts a subset of C, C++ and SystemC code. Its setup tool allows for specifying a number of tuning options such as the target technology, clock rate, design constraints, as well as a number of optimizations. Libraries are included to support a number of useful structures such as arbitrary width data types. Like SystemC, Catapult C simulations can be created by compiling the program with a C++ compiler. It also supports keeping track of performance figures for each revision, allowing for manual tuning [2].

B. Dependently Typed Languages

Dependently typed languages can give extremely powerful compile time guarantees, and many have facilities for verifying properties of programs through interactive theorem checkers or proof assistants. Types are a first class language construct in that values can appear in types. The type systems in these languages are usually an extension of type systems found in purely functional languages such as ML or Haskell. However, a complication of a dependent type system is that type inference is no longer possible in many cases that would be possible in ML or Haskell like type systems [12]. Listing 2 implements a list with size in Idris.

```
Data Vect : Nat -> Type -> Type
  Nil : Vect 0 a
  (::) : a -> Vect n a -> Vect (n + 1) a
```

Listing 2. Vector definition in Idris

Notice that the type itself contains a natural number to represent the size, two Vects are not of the same type if their sizes differ. Appending a single element to the Vect increases its size by 1.

To illustrate some of the benefits of having the size in the type listing 3 defines an append function which takes two Vects and appends the second one to the first one, creating a Vect which size is equivalent to the sum of the sizes of the two Vects.

```
append : Vect n a -> Vect m a ->
         Vect (n + m) a

append [] ys = ys
append (x :: xs) ys = x :: append xs ys
```

Listing 3. Append definition of Vects in Idris

The Idris compiler enforces that the function returns a Vect which size is exactly the size of the first Vect added to the size of the second Vect.

Implementing the EDSL in a dependently typed language allows us to verify at compile time that the vector transformations still preserve the results of the computations.

We explore the three main dependently typed languages to select which one is most appropriate to use for this project.

1) Idris: Idris[13] is a dependently typed programming language which features Haskell-like syntax and tactic based theorem proving, it was also designed to be used for practical general purpose programming, and puts high level programming ahead of theorem proving. As a result of this it has high level features such as type classes and list comprehensions.

Like Haskell it also includes 'do' notation for chaining together monadic computations, making them more readable. The compiler is designed to perform many performance improving optimizations, most notably applying partial evaluation whenever possible. This can be useful for efficiently implementing EDSLs[14]. Idris also has support for implementing EDSLs[15], such as allowing to extend the syntax of the language.

A primary goal of Idris is to generate efficient code, and it's main back end compiles to c. During compilation Idris code can be transformed to a number of intermediate languages that can be used to create custom back ends. Currently there exists a number of back end implementations, most notably LLVM, Javascript, and Java.

Idris currently at the time of writing does not have a stable release (the current release version being 0.9.20) which may cause some issues in development, mainly that code written for the current release may not be compatible with future releases of the language.

2) Agda: Agda[16] is a dependently typed programming language with similar syntax to Haskell, and a similar type system to Idris. Unlike both Idris and Coq, Agda does not have support for tactics, so proofs have to be written in a functional style. Agda is a total functional language[17] (i.e. every function must terminate, and every function must cover all possible values for its input type(s)). As a result of this Agda is not Turing complete, and recursive functions must match a schema which can be proven to terminate.

Agda was mainly designed with theorem proving in mind, and does not contain a lot of the "high level" features present in Idris.

Agda's main back end compiles down to Haskell.

3) Coq: Coq[18] is a proof assistant which implements a dependently typed programming language called Gallina, thus the main focus of Coq is for theorem proving rather than for general purpose programming. It has ML-like syntax. Coq Programs can be extracted to a more conventional programming language such as Haskell, this allows for verifying properties of programs written in mainstream languages. Coq, like Agda is a total functional language and as a result is not Turing complete.

Coq, unlike both Idris and Agda, doesn't support induction-recursion[19] and as a result it's not possible to define mutually recursive types and functions.

C. Type Transformations on Vector Types

Type transformations on vector types are described in more detail by Vanderbauwhede [3]. A type transformation is essentially a function applied to a value of a given type, which generates a value of another type. For a function to be a type transformation it must also obey a set of rules on how the types are modified.

We use the following definition for a vector type:

- A vector type is a type containing a natural number **k**, and an arbitrary type **a**.
- If **k** is greater than 0, the vector type represents a list of size **k** of type **a**.
- If **k** is equal to 0, the vector type is equivalent to the type **a**.

This definition is similar to the Vect type as shown in figure 2. With the key difference being that this definition cannot have an empty vector, and that the type of an empty vector of a given type is equivalent to the type itself.

There are three fundamental type transformations on vector types (as well as their respective inverse transformations) which are reshaping, mapping, and converting a type to a singleton vector. These transformations modify the vector whilst still retaining some of the structural properties, mainly the ordering of elements, and the total amount of base elements contained in the vector.

A program transformed into a combination of map, fold, and zip operations on a vector can be transformed using a series of these type transformations whilst still retaining the same computational results.

We can transform the Fortran code from listing 1 into this form by representing the array **A** as a 1 dimensional vector, and the addition on **A** as a map transformation:

```
C = map (+ 1) A
```

which is equivalent to:

```
C = map (+ 1) [1,2,3,4,5,6,7,8,9,10,11,12]
```

Using the singleton transformation, we can increase the dimension of **A** to 2, since the dimensionality of the vector is being increased, a repeated application of map is needed:

```
C = map (map (+ 1))
    [[1,2,3,4,5,6,7,8,9,10,11,12]]
```

We can then perform a reshape operation using the value 6, which is a factor of the size of the first dimension of A. This multiplies the size of the second dimension by 6, and divides the size of the first dimension by 6. The previous program is transformed into the following:

```
C = map (map (+ 1))
    [[1,2], [3,4], [5,6],
     [7,8], [9,10], [11,12]]
```

By performing these transformations and then flattening the result back into a 1 dimensional vector we get the exact same result as the original program.

What we gain from this is that the inner maps can each be performed sequentially, whilst the outer map can be performed in parallel, by transforming the program we can tune the amount of parallelization.

V. Feasibility Study

Below we show how it's possible to directly implement the three main vector type transformations (along with a couple of other particularly useful ones) in Idris, which are guaranteed to be correct at compile time.

The *Shape* of a vector is defined as an Idris *Vect* of a given number dimensions, each represented as a natural number. This shape is used to define the vector type.

Defining the vector requires a recursive type definition, having the base case of a 0 dimensional vector equivalent to the base vector type. This follows from the result that if any dimension in the vector has 0 elements, then the vector itself must have no elements.

```
Shape : Nat -> Type
Shape n = Vect n Nat

Vector : Shape n -> Type -> Type
Vector Nil t = t
Vector (v::vs) t = Vect v (Array vs t)
```

A. Implementing Vector Type Transformation Functions

From the *Vector* definition it's trivial to define the *singleton* function as a function taking an n dimensional *Vector* and mapping it to a 1+n dimensional *Vector*, as well as its inverse function:

```
singleton : Vector xs t ->
            Vector (1::xs) t
singleton t = [t]

invSingleton : Vector (1::xs) t ->
               Vector xs t
invSingleton [t] = t
```

Defining the *map* instance uses pattern matching on the dependent types to check whether the number of dimensions the *Vector* has is 0, in which case the given function is applied to the single element. Otherwise the map function is applied recursively to all elements in the *Vector* until all the base elements have been transformed.

```
mapV : (a -> b) -> Vector xs a ->
                   Vector xs b
mapV f v {xs=[]}  = f v
mapV f v {xs = (y::ys)} = map (mapV f) v
```

Taking a given *Vector* which has 2 or more dimensions, we can decrease the dimensionality by 1 by concatenating all the Sub-Vectors as follows

```
redDim : Vector (x1::x2::xs) t ->
         Vector (x1*x2::xs) t
redDim [] = []
redDim (v::vs) = v ++ redDim vs
```

This results in a *Vector* with a dimensionality of one less than the input *Vector*. The new *Vector* contains the same elements as the input *Vector*, and ordering is also preserved.

```
reshape : Vector (x1::(m * x2)::xs) t ->
          Vector((x1 * m)::x2::xs) t
reshape [] = []
reshape (v::vs) = incDim v ++ reshape' vs
```

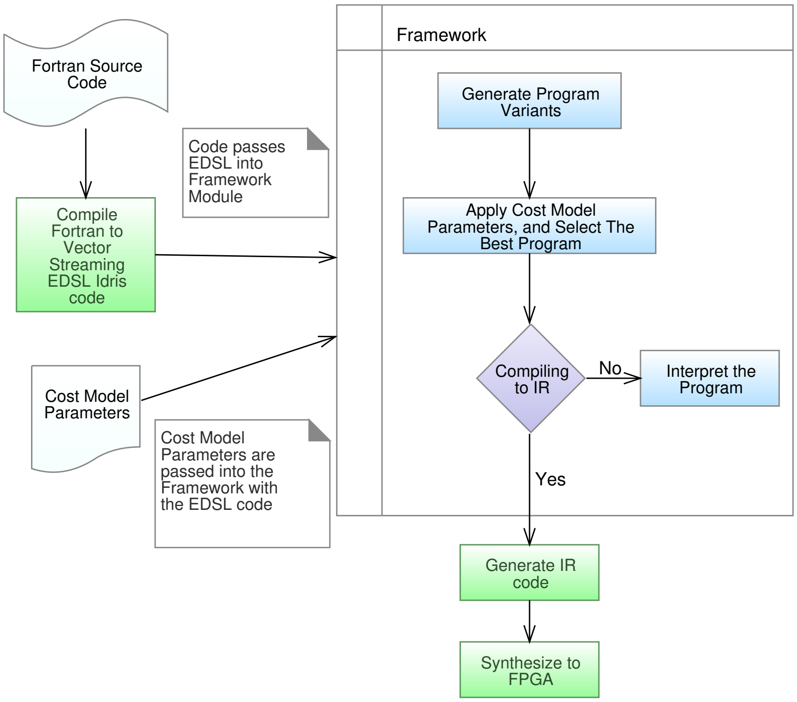
Fortran Source Code
Framework
Generate Program Variants
Code passes EDSL into Framework Module
Apply Cost Model Parameters, and Select The Best Program
Compile Fortran to Vector Streaming EDSL Idris code
Compiling to IR
No
Interpret the Program
Cost Model Parameters
Cost Model Parameters are passed into the Framework with the EDSL code
Yes
Generate IR code
Synthesize to FPGA

Fig. 1. Flowchart illustrating the structure of the Framework.

```
invReshape :
    Vector ((x1 * m)::x2::xs) t ->
    Vector (x1::(m * x2)::xs) t

invReshape v {x1 = Z} = []
invReshape v {x1 =S r} =
    map redDim $ incDim v
```

Taking a given *Vector* which has 1 or more dimensions; in which the most "outer" dimension is made up of **n * m** elements, where **n, m** ∈ ℕ. We can increase the dimensionality of the given *Vector* by 1 by splitting the *Vector* at each **n** sub-Vectors which results in **m** lots of **n** sub-Vectors.

```
incDim : Array (m * n::xs) t ->
         Array (m::n::xs) t

incDim v {m = Z}  = []
incDim v {m = S r} {n} =
    let (fstN, rest) =
      (take n v, drop n v) in
        fstN :: (incDim rest)
```

Defining the *reshape* transformation function and its inverse can be achieved with aid from the increase and decreasing dimensional transformations listed above.

Due to Idris' support for implementing EDSLs, and the ability to directly implement the vector transformations, it should be feasible to fit these transformations into an EDSL.

VI. Project Plan

Figure 1 shows the overview of the proposed framework.

The project plan (illustrated in figure 2) is described below

Project Proposal (Weeks 1-7):

- Researching, performing a feasibility study, and writing up this report.

Implementation (Weeks 8-24):

- **Weeks 8-14** Implement the EDSL, which encapsulates the vector type transformations, supports a rudimentary cost model, as well as including variants for describing how to parallelize maps and folds. Also implement the ability to interpret the EDSL from Idris, this can be helpful for testing while creating the framework, as well as for testing programs before converting them onto a FPGA.
- **Weeks 15-19** Implement generating all possible program variants from a given program, if possible use Idris with the vector type transformation properties to prove that variants will always generate the same results as the original program.
- **Weeks 20-22** Implement a flexible interface for exporting the EDSL into IR formats.
- **Weeks 23-24** If the project is not running on schedule this time can be allocated to catching up. Due to Idris not having a stable release the codebase may need to be updated if there are breaking changed, or it may take some time to overcome an issue due to bugs in the implementation. If on schedule, this time can

be allocated into adding new features, or improving the implementation.

Project Writeup + Presentation (Weeks 25-28)

- This time is dedicated to writing up the project paper and preparing the presentation.

References

[1] G. Moertti, “System-level design merits a closer look,” *EDN Asia*, pp. 22–28, Feb. 2002.

[2] W. Meeus, K. Van Beeck, T. Goedemé, J. Meel, and D. Stroobandt, “An overview of today’s high-level synthesis tools,” *Des. Autom. Embedded Syst.*, vol. 16, pp. 31–51, Sept. 2012.

[3] W. Vanderbauwhede, “Inferring program transformations from type transformations for partitioning of ordered sets,” 2015.

[4] A. Butterfield and J. Woodcock, “"a hardware compiler semantics for handel-c,” *Electronic Notes in Theoretical Computer Science*, vol. 161, pp. 73 – 90, 2006. Proceedings of the Third Irish Conference on the Mathematical Foundations of Computer Science and Information Technology (MFCSIT 2004)Proceedings of the Third Irish Conference on the Mathematical Foundations of Computer Science and Information Technology (MFCSIT 2004).

[5] S. Schneider, H. Treharne, A. McEwan, and W. Ifill, “Experiments in translating CSP || B to handel-c,” in *The thirty-first Communicating Process Architectures Conference, CPA 2008, organised under the auspices of WoTUG and the Department of Computer Science of the University of York, York, Yorkshire, UK, 7-10 September 2008* (P. H. Welch, S. Stepney, F. Polack, F. R. M. Barnes, A. A. McEwan, G. S. Stiles, J. F. Broenink, and A. T. Sampson, eds.), vol. 66 of *Concurrent Systems Engineering Series*, pp. 115–133, IOS Press, 2008.

[6] S. Swan and C. D. Systems, “An introduction to system level modeling in systemc 2.0,” tech. rep., 2001.

[7] D. Tabakov, M. Y. Vardi, G. Kamhi, and E. Singerman, “A temporal language for systemc,” in *Proceedings of the 2008 International Conference on Formal Methods in Computer-Aided Design*, FMCAD ’08, (Piscataway, NJ, USA), pp. 22:1–22:9, IEEE Press, 2008.

[8] B. G. khanovich A, Aboulhamid EM, “Object-oriented techniques in hardware modeling using systemc,” in *Proceedings of Northeast workshop on circuits and systems*, June 2003.

[9] Y. Said, T. Saidani, and M. Atri, “High-level design for image processing on fpga using xilinx acceldsp,” in *Computer Applications and Information Systems (WCCAIS), 2014 World Congress on*, pp. 1–5, Jan 2014.

[10] Synopsys, “Symphony c compiler.” Website.

[11] Calypto, “Catapult c.” Website.

[12] H. Xi and F. Pfenning, “Dependent types in practical programming,” in *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL ’99, (New York, NY, USA), pp. 214–227, ACM, 1999.

[13] E. Brady, “Idris, a general-purpose dependently typed programming language: Design and implementation,” *Journal of Functional Programming*, vol. 23, pp. 552–593, 9 2013.

[14] E. C. Brady and K. Hammond, “Scrapping your inefficient engine: Using partial evaluation to improve domain-specific language implementation,” *SIGPLAN Not.*, vol. 45, pp. 297–308, Sept. 2010.

[15] E. Brady and K. Hammond, “Resource-safe systems programming with embedded domain specific languages,” in *Practical Aspects of Declarative Languages* (C. Russo and N.-F. Zhou, eds.), vol. 7149 of *Lecture Notes in Computer Science*, pp. 242–257, Springer Berlin Heidelberg, 2012.

[16] U. Norell, “Dependently typed programming in agda,” in *Proceedings of the 6th International Conference on Advanced Functional Programming*, AFP’08, (Berlin, Heidelberg), pp. 230–266, Springer-Verlag, 2009.

[17] D. A. Turner, “Total functional programming,” vol. 10, pp. 751–768, jul 2004.

[18] C. Adam, “An introduction to programming and proving with dependent types in coq.,” *Journal of Formalized Reasoning*, vol. 3, no. 2, pp. 1–93, 2010.

[19] P. Dybjer, “A general formulation of simultaneous inductive-recursive definitions in type theory,” *The Journal of Symbolic Logic*, vol. 65, pp. 525–549, 6 2000.

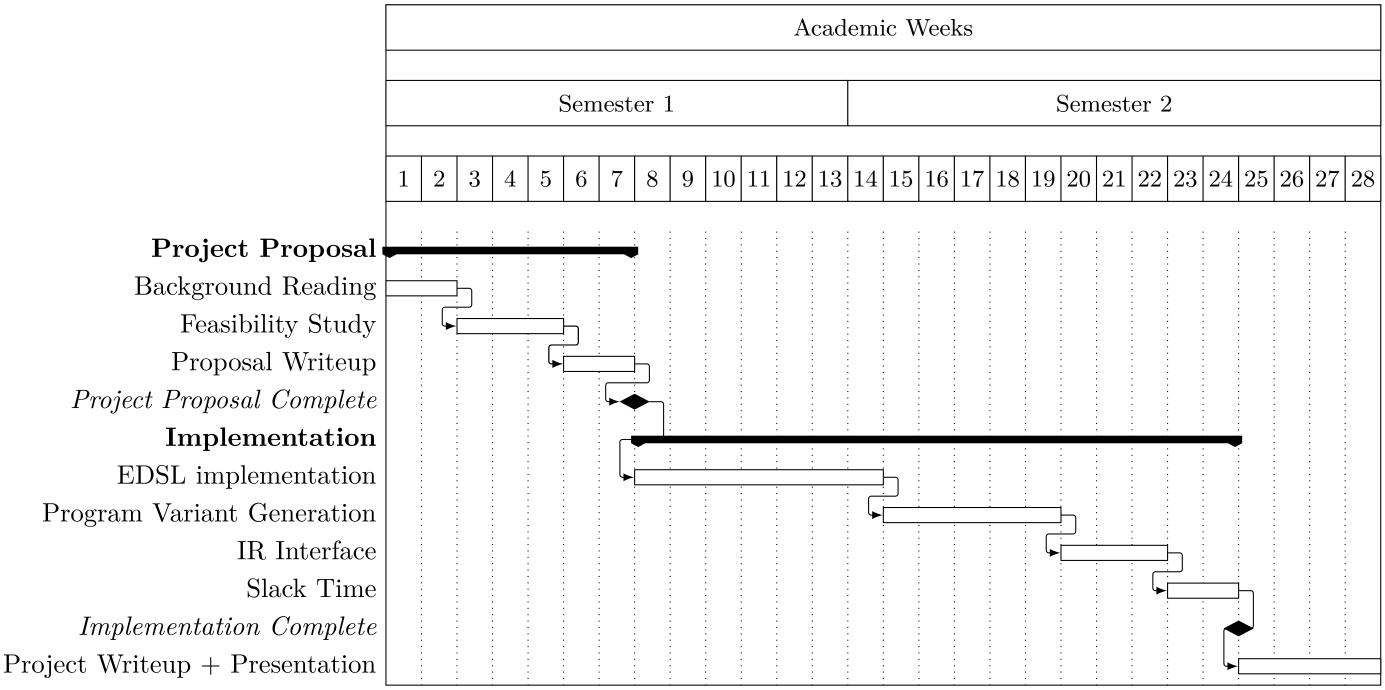

Fig. 2. Project Schedule as Gantt Chart