BuildIt: A Type-based Multi-stage Programming Framework for Code Generation in C++

Ajay Brahmakshatriya
CSAIL, MIT
Cambridge, USA
ajaybr@mit.edu

Saman Amarasinghe
CSAIL, MIT
Cambridge, USA
saman@csail.mit.edu

***Abstract*— The simplest implementation of a domain-specific language is to embed it in an existing language using operator overloading. This way, the DSL can inherit parsing, syntax and type checking, error handling and the toolchain of debuggers and IDEs from the host language. A natural choice of host language for most high-performance DSLs is the de-facto high-performance language, C++. However, DSL designers quickly run into the problem of not being able to extract control flows due to lack of introspection in C++ and have to resort to special functions with lambdas to represent loops and conditionals. This approach introduces unnecessary syntax and does not capture the side effects of updates inside the lambdas in a safe way. We present BuildIt, a type-based multi-stage execution framework that solves this problem by extracting all control flow operators like `if-then-else` conditionals and `for` and `while` loops using a pure library approach. BuildIt achieves this by repeated execution of the program to explore all control flow paths and constructing the AST piece-by-piece. We show that BuildIt can do this without exponential blow-up in terms of output size and execution time.**

We apply BuildIt's staging capabilities to the state-of-the-art tensor compiler, TACO to generate low-level IR for custom level formats. BuildIt thus offers a way to get both generalization and programmability for the user while generating specialized and efficient code. We also demonstrate that BuildIt can generate rich control-flow from relatively simple code by using it to stage an interpreter for an esoteric language.

BuildIt changes the way we think about multi-staging as a problem by reducing the PL complexity of a supposedly harder problem requiring features like introspection or specialized compiler support to a set of common features found in most languages.

***Index Terms*—Multi-stage programming, Domain-specific Languages, Code generation, Meta-programming**

I. Introduction

Multi-stage programming or generative programming has many applications ranging from efficient execution for deep neural network models, serving dynamic websites, to generating efficient code for applications in specific scientific domains. Multi-stage programming provides a way of getting generality and simplicity of programming while maintaining high performance and specialization [1]–[4]. Domain-specific languages like TACO [5], Tensorflow [6], Halide [7] and GraphIt [8], [9] are simply 2 stage programming frameworks where the first stage is the high-level DSL specification and the second level is the generated low-level efficient C++/CUDA code targetting CPU/GPUs. TensorFlow [6] extracts static and dynamic execution graphs from the input program to automatically calculate gradients for the neural network layers and also to efficiently execute the networks when the inputs are available. In the most general, sense multi-stage programming has several stages where the output of a particular stage is the code for the next stage. Figure 2 shows a comparison between traditional single-stage programming and multi-stage programming. Each stage can use a different programming language and libraries and has its own set of inputs. For example in dynamic websites, server-side languages like PHP and NodeJS produce HTML, JavaScript and CSS for second stage execution in the user's browser. One can think of traditional single-stage programming as a special case of multi-stage programming with one stage.

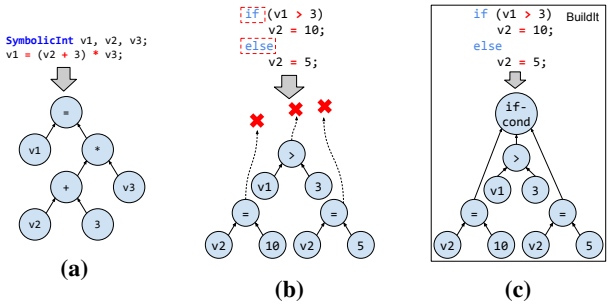

Fig. 1: a) Operators =, + and * (in red) overloaded fo the new `SymbolicIntType` type (in blue) to create an AST for the expression. b) Shows similar ASTs created for the subexpressions, but there is no way to combine the three into an `if-then-else` because C++ does not support overloading control flow and lacks introspection c) BuildIt is able to construct the full AST

Generating code for different stages can take multiple approaches. The user explicitly outputs as strings the code for the next stage (PHP outputting HTML) or use a language like MetaOCaml [10] that has built-in support for multiple stages using a specialized compiler. We will discuss the pros and cons of each of these approaches in detail in Section II. One popular approach taken by many multi-stage programming frameworks such as Tensorflow [6] or Halide [7] is to introduce a new type and use operator overloading and symbolic execution to extract the program representation for the next stage. Figure 1

BuildIt is available open-source under the MIT license at https://github.com/AjayBrahmakshatriya/buildit.

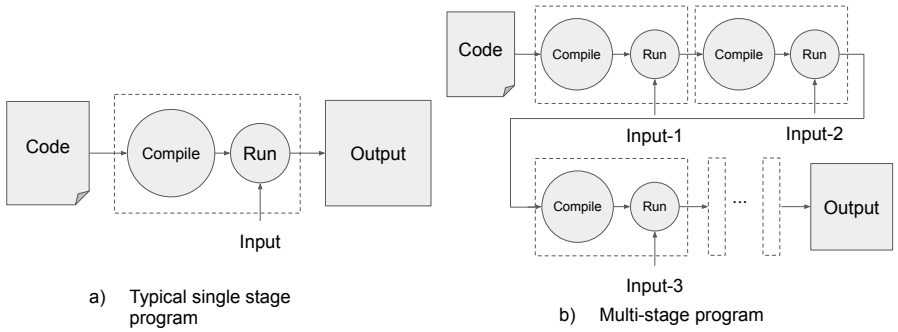

Fig. 2: Comparison between a) traditional single stage programming and b) multi-stage/generative programming. Notice each stage in b) has its own compilation and execution step with its own set of inputs.

a) shows how in an imperative and statically typed language like C++, the addition, multiplication and assignment operators can be overloaded to create the abstract syntax tree (AST) for the next stage. This approach quickly runs into a major problem that data-dependent control flow cannot be extracted because C++ does not support overloading conditions and loops as shown in the second example of Figure 1 b). With the lack of introspection in C++, it is practically impossible to get a similar effect without relying on compiler changes. Since C++ is used in code generation and lowering phases of many high-performance language backends, looking for alternatives to introspection is important. Using compiler modification approaches generally compromises portability and increases the complexity of codebases.

We present BuildIt, a type-based multi-stage programming framework for C++ that solves this exact problem. BuildIt uses a purely library-based approach that doesn't require any compiler changes and is capable of extracting ASTs for the next stage from the input code with all the rich data-dependent structured control flow elements like `if-then-else` conditions, `while` and `for` loops and recursion. BuildIt makes a key observation that the input program can be executed several times to explore different control flow paths in the program. This is combined with traditional operator overloading and symbolic execution to get the entire AST for the next stage program. BuildIt further makes the extraction process tractable and efficient by applying novel static tags and known techniques like memoization.

Unlike introspective solutions, BuildIt does not have a full view of the program. Rather it looks at the program through the narrow window of the calls to the overloaded operators. This is analogous to a person trying to navigate a maze. They only see a small part of the maze at a time. But by carefully recording observations and leaving markers on paths explored, the person can navigate the maze successfully. BuildIt also uses memoization and early merging of control flow paths to reduce the extraction complexity from exponential to polynomial in the number of branches.

A. Type based multi-stage programming

Different multi-stage programming frameworks further vary by the way the binding times of sub-expressions and variables are decided. By the binding time, we mean the stage in which a particular subexpression is actually evaluated with concrete values instead of just producing the code for the next stage. For frameworks that use different languages for each stage, the binding times are decided by which language the expression is written in. BuildIt is a type-based multi-stage programming framework, meaning the types of the variables and sub-expressions decide which stage they will be bound in. Since C++ is a statically typed language, the declared types of the variables decide the binding times. BuildIt provides different types for each stage. These types will be further explained in detail in Section III.

B. Rethinking meta-programming

Multi-stage programming has been studied a lot in the literature with a general consensus that multi-stage programming requires specialized language constructs to be implemented in the language with a compiler([4], [11]) or the existence of features like introspection in the host language([6], [12]–[14]). BuildIt has both theoretical and practical implications in this field. The ideas presented in this paper have foundational implications for our understanding of PL complexity. Most features in languages can be simplified to a composition of control flow (like loops and conditionals) and basic binary and unary operations. However, to-date staging required additional PL complexity like access to program AST or specialized constructs and compiler support. BuildIt shows that we can reduce the complexity of a supposedly harder problem to a simpler problem. With this, we try to change the way we think about staging as a class of problems. By implementing our framework in C++, we show that staging is in fact a subclass of a class of problems that only require basic operations to implement. This also has several practical implications. Because we rely only on standard C++ constructs, our implementation is portable across platforms and compilers. C++ is also one of the most commonly used languages for high-performance applications and domain experts already have familiarity with the language and optimizations required. This is important because staging was either not possible or very difficult to use with C++. While our implementation is specific to C++, the ideas presented can also be applied to other languages since BuildIt relies only on the common features found in most programming languages.

This paper makes the following contributions -

- BuildIt combines operator overloading and symbolic execution with repeated executions to explore all control

flow paths and is the first framework that can extract the AST with all loops and conditionals for multi-stage execution in C++.
- BuildIt does not require a separate syntax or specialized constructs for control flow like `if-then-else` and `for` and `while` loops. This makes it extremely easy to change the binding times of sub-expressions without a lot of code rewrite.
- BuildIt is also the first framework for imperative languages to allow side-effects on unstaged variables inside conditions on staged variables. This allows rich patterns to be extracted and generated. BuildIt is implemented as a library by overloading basic binary and unary operators and does not require any compiler changes. This makes BuildIt very portable and lightweight.
- We also show how generative multi-stage programming be applied to code generation for DSLs thus offering generalizability and programmability while providing specialization and efficient code generation.
- Finally, BuildIt changes the way we think about multi-staging as a problem by reducing the complexity of a supposedly harder problem to a set of basic PL features

The rest of the paper is structured as described below -

- Section II discusses different approaches to extracting AST of programs.
- Section III introduces the BuildIt programming model and describes the `static<T>` and `dyn<T>` types.
- Section IV describes in detail the AST extraction process in BuildIt and how control flow is handled with repeated execution.
- Section V shows 2 case studies where BuildIt can be applied and how BuildIt significantly improves the programmability while providing specialization.
- Section VI talks about different related works in generative and multi-stage programming and the difference in the approaches they take.

II. Background

This section talks about different AST extraction methodologies and the pros and cons of each approach. We then describe the approach taken by BuildIt and how it tries to tackle some of the problems.

A. Text-based generative programming

Text-based multi-staging simply requires the user to print the code for the next stage and hence is the easiest to understand and implement. This approach is taken by server-side languages in webservers like PHP and NodeJS. Figure 3 shows a loop written in PHP that produces a list of items in the generated HTML. Notice how PHP treats the code for the HTML stage as just a string and echoes it to the standard output.

This approach does not require any special compiler or library support as long as the language supports strings natively. The output of the program is simply fed into the compiler for the next stage. The main limitation of the text-based approach is that it doesn't allow for any optimizations or type checking of

```
<?php echo '<ul>';
 for ($i = 0; $i < 3; $i++)
  echo '<li>Item'.$i.'</li>';
 echo '</ul>';
?>
```

```
<ul>
<li>Item 1</li>
<li>Item 2</li>
<li>Item 3</li>
</ul>
```

Fig. 3: The PHP code on the left executes on the webserver to produce the HTML code on the right to be executed in the user's browser. Notice how the PHP code treats the HTML as strings.

```
template<int M>
void init(int arr[M]){
 for (int x = 0; x < M; x++)
  arr[x] = val;
}
init<20>(0, array1);
init<10>(0, array2);
```

```
...
for (x=0; x<20; x++)
  arr[x] = val;
...
for (x=0; x<10; x++)
  arr[x] = val;
...
```

Fig. 4: The C++ template code on the left defines a function where the M value is a template argument. On the right we can see the different versions of the same function generated for different template arguments.

the generated code because the generated code is just a string. The code would need to be parsed and analyzed before any transformations or optimizations can be done in a meaningful way. IDE/debugger support is also lacking till the generated code is compiled/run.

B. Compiler based with new language syntax

This approach adds a specialized syntax for the code of each stage and supports the execution in different stages with a help of a specialized compiler/interpreter. A very commonly used example of such a technique is the C++ template language where the templates use a different syntax from the rest of the C++ code and are handled specifically by the compiler. Figure 4 shows how templated functions (or classes) can be instantiated with different arguments to produce different code to be compiled and executed.

With a compiler-based technique, many rich features like optimizations, type checking, debugger support can be integrated into the compiler for all stages. But the main disadvantage is that a specialized compiler needs to be implemented and maintained instead of maintaining just a library. The user also has to learn and adapt to a new syntax for each stage.

Both the techniques above, text-based and specialized syntax-based suffer from another common drawback. Because the code for each of the stage looks very different, it is very difficult to move code between different stages. Generally, moving code between stages requires rewriting the entire program which greatly hampers the productivity of the user.

C. Operator overloading and special functions for control flow

One of the most promising approaches for seamless multi-stage programming is to use operator overloading with symbolic execution as mentioned in Section I. To handle control flow the framework can add through a library specialized operators that take in lambdas and subexpressions. This approach is used in the first version of TensorFlow to construct dynamic graphs as shown in Figure 5.

This approach solves most of the limitations mentioned above. But this approach still uses a specialized syntax for control flow elements and it is difficult to move code between

```
import tensorflow as tf
z = a * b
result = tf.cond(x < y,
          lambda: x + z,
          lambda: y ** 2)
```

```
z = a * b
if (x < y):
  result = x + z
else
  result = y ** 2
```

Fig. 5: An example `if-then-else` condition written in TensorFlow. Notice that the condition to be checked and the body of the then and else branches are supplied as lambdas. The operators like ¡ and + are overloaded for the type tf.Tensor. The figure on the right shows what the constructed condition would look like.

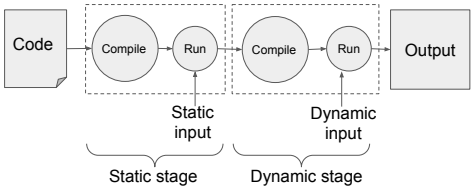

Fig. 6: To explain the programming model we first descibe a simple 2 stage execution case and then generalize it to arbitrarily many stages in Section IV.

two stages. This approach also doesn't allow side effects on non-stages variables inside staged code. We will see how this is relevant in Section V and what limitations it possesses.

BuildIt solves all the above-mentioned problems by using the same general C++ syntax for handling control flow using just a library. BuildIt also allows updates to stage `n` variables inside conditionals and loops on stage `> n` seamlessly. Like mentioned before, BuildIt executes the program repeatedly to extract all control flow paths. This way the updates to the unstaged variables are limited only to the cases when the branch is taken during the next stage execution.

III. Programming Model

In this section, we describe in detail the programming model for BuildIt programs. This includes the two new types `static<T>` and `dyn<T>` and the constraints BuildIt programs are subject to.

A. A library-based approach

As mentioned before in Section I, BuildIt takes purely library approach. This means users can start using BuildIt by including the BuildIt headers that define the new types and the overloaded operators and linking against the runtime library that has the AST extraction and code generation implementation.

B. Multi-stage to 2 stage relaxation

We will introduce the programming model by first describing a two-stage BuildIt program. We do this to make it easy for understanding the terminology and how the extraction process works. We will then generalize the programming model to arbitrary many stages in Section IV. As shown in Figure 6, we will call the two stages static stage and dynamic stage. We will also call the inputs taken by these stages static inputs and dynamic inputs respectively.

C. Type based programming model

As mentioned before BuildIt is a type-based programming framework. This means that BuildIt uses declared types of the variables and sub-expressions to decide the binding times. We will explain below the two new types BuildIt introduces, `static<T>` and `dyn<T>`-

```
int power(int base, int exponent) {
  int res = 1, x = base;
  while (exponent > 1) {
    if (exponent % 2 == 1)
      res = res * x;
    x = x * x;
    exponent = exponent / 2;
  }
  return res * x;
}
```

Fig. 7: A typical implementation of the `power` function that calculates $\texttt{base}^{\texttt{exponent}}$ using repeated squaring.

```
dyn<int> x = 0;
dyn<long> y = 0;
static<int> z = 10;
if (x > z)
  x = x + y;
else
  x = x * y;
```

```
int var1 = 0;
long var2 = 0;
/*No trace of z*/
if (var1 > 10)
  var1 = var1 + var2;
else
  var1 = var1 * var2;
```

Fig. 8: The BuildIt code shown in the left executes in the static stage to produce the code shown on the right to be executed in the dynamic stage. Notice how `dyn<int>` produces variable and expression of type `int` in the generated code. The conditions on expressions of type `dyn<T>` produce the same conditions in the generated program.

1) Static Type: `static<T>`: The user can declare variables of type `static<T>` (where `T` is any C++ primitive type) to indicate variables and expressions that should be evaluated in the static stage. `static<T>` variables have concrete values of type `T` during the static stage. Control flow-dependent only on expressions of type `static<T>` is resolved during the static stage and does not produce any conditionals or loops in the generated code. `static<T>` currently only supports wrapping primitive C++ types that have a comparison operator defined.

2) Dynamic Type: `dyn<T>`: The user can declare variables of type `dyn<T>` (where `T` is any type) to indicate variables and expressions that should be executed in the dynamic stage. An expression written using variables of type `dyn<T>` does not have concrete values during the static stage. Instead, it produces the exact same expression with type `T` to be evaluated during the dynamic stage. For example, the BuildIt code shown in the left of Figure 8 produces code shown in the right of Figure 8 for dynamic stage evaluation. Notice how declarations of type `dyn<int>` produce declarations of type `int`. We can also see in the figure that the expressions of type `static<int>` are completely evaluated in the static stage and have their values appear as `int` constants in the generated code. `dyn<T>` variables can be used inside the boolean expressions for conditions and loops to produce control flow elements in the generated code for the dynamic stage.

3) BuildIt programs: A BuildIt programs looks like any other C++ program except that it uses two extra types to decide the binding times of all expressions. BuildIt does not use any new syntax or special function calls for control flow elements like `if-then-else` conditions, `for` loops or `while` loops. This makes migrating existing code to different variations of multi-stage code easy. For example, Figure 7 shows a single stage implementation of the `power` function. This function takes two inputs, `base` and `exponent` of type `int` and returns $\texttt{base}^{\texttt{exponent}}$. The function uses repeated

```
dyn<int> power(dyn<int>
    base, static<int> exp){
  dyn<int> x=base, res=1;
  while (exp > 1) {
    if (exp % 2 == 1)
      res = res * x;
    x = x * x;
    exp = exp / 2;
  }
  return res * x;
}
```

```
int power_15(int base){
  int res = 1;
  int x = base;
  res = res * x;
  x = x * x;
  res = res * x;
  x = x * x;
  res = res * x;
  x = x * x;
  return res * x;
}
```

Fig. 9: The `power` function where the exponent is bound in the static stage and the base is bound during the dynamic stage. On the right is the generated code when the static input (exponent) is supplied 15.

```
dyn<int> power(static<int>
    base, dyn<int> exp) {
  dyn<int> res=1, x=base;
  while (exp > 1) {
    if (exp % 2 == 1)
      res = res * x;
    x = x * x;
    exp = exp / 2;
  }
  return res * x;
}
```

```
int power_5(int exp){
  int res = 1;
  int x = 5;
  while (exp > 1) {
    if (exp % 2 == 1)
      res = res * x;
    x = x * x;
    exp = exp / 2;
  }
  return res * x;
}
```

Fig. 10: The `power` function where the base is bound in the static stage and the exponent is bound during the dynamic stage. On the right is the generated code when the static input (base) is set as 5.

squaring to achieve this.

Figure 9 shows how the program can be specialized for a particular exponent in BuildIt by declaring the `exponent` as `static<int>` and the `base` and `dyn<int>`. The code on the right shows the generated code for exponent specialized as 15. Since all the loops and conditions are based on `static<T>` expressions, they are evaluated away to produce straight-line code. Figure 10 shows the same function specialized for a particular base by declaring the `base` as `static<int>` and the `exponent` as `dyn<int>`. The figure on the right shows the code generated for the base specialized as 5. Since the loop is now based on a `dyn<T>` expression, it is present in the output code.

All BuildIt programs must use `static<T>` and `dyn<T>` to declare variables and write expressions. BuildIt programs can also use expressions that are not `static<T>` or `dyn<T>`, but they can be accessed only in read-only mode. These variables would have the same behavior as `static<T>` and would be completely evaluated in the static stage.

IV. AST Extraction Methodology

In this section we will describe in detail the methodology BuildIt follows to extract the AST for the program to executed in the dynamic stage.

A. Handling static variables

Variables of type `static<T>` are simply wrappers around variables of type `T` and mimic the behavior of the enclosed type. An implicit conversion operator is defined to expose the value of the wrapped type. Thus we can perform all operations on `static<T>` that are valid for `T` including assignment, binary and unary operators, using them in `if-then-else` and loops without any change.

```
builder_context context;
//non BuildIt type read only vars here
const int iter = atoi(argv[2]);
auto ast = context.extract([=] {
    for (dyn<int> x = 0; x < iter; x++)
        ...
});
ast->dump(std::cout, 0);
```

Fig. 11: Example showing how to wrap BuildIt code inside a lambda and pass it to a Builder Context object. The code can also be wrapped inside a function that takes arguments

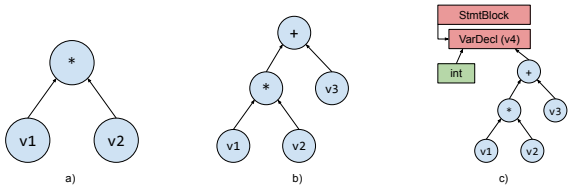

Fig. 12: Step by step construction of expression trees. Notice that the expressions are shown in blue, statements are show in red and types are show in green. The `VarDecl` AST node has `int` as a type attached to it.

B. Extracting straight-line code

BuildIt uses operator overloading and symbolic execution to extract information about the operators and variable declarations for `dyn<T>` expressions and variables. This preserves the syntax for these operators while hiding the complexity of creating the expressions from the user. The programmer wraps the code to be extracted in a function or a lambda and passes it to a Builder Context object. This allows BuildIt to execute the code repeatedly to explore all control flow paths. Figure 11 shows a simple example of how the Builder Context object is invoked with the lambda.

BuildIt overloads all the binary and unary operators for expressions of type `dyn<T>` to create appropriate AST nodes and return a reference to the created nodes. These AST references are further combined to create more AST nodes and to construct the AST for the entire expression. For example, suppose we have the expression `v1 * v2 + v3` where `v1`, `v2` and `v3` are of type `dyn<T>`. Since usual C++ precedence rules follow, the sub-expression `v1 * v2` is evaluated first to create an AST node for the `*` operator as shown in Figure 12 a). When the `+` operator is evaluated next, BuildIt creates a new AST node for the `+` operator and nests the previously created AST as a sub-expression as show in Figure 12 b).

All the constructors for the variables of type `dyn<T>` are overloaded to create variable declaration statements in the AST being constructed. For example, suppose we have a variable declaration of type `dyn<int>` as `dyn<int> v4 = v1 * v2 + v3;`. First the AST for `v1 * v2 + v3` is created as explained as above. The copy constructor for `v4` is called with the AST for `v1 * v2 + v3` as the parameter. This creates a variable declaration statement AST node and adds it to the AST being constructed as shown in Figure 12 c).

The other type of statements in straight-line code is expression statements and BuildIt captures it with some bookkeeping to identify the end of a statement. The Builder Context object holds an ordered list of uncommitted expressions. Whenever

```
    ④   ①   ③   ②
v1 = v2 * v3 + v4 / v5;
    ⑥   ⑤
v6 = v7 && v8;
    ⑦
dyn<int> v9;
```

Fig. 13: Sample straight line code to be extracted. Overloaded operators are in red and one possible execution order for the operators is show above each of them

```
1) UL: ["v2 * v3"]
2) UL: ["v2 * v3", "v4 / v5"]
3) UL: ["v2 * v3 + v4 / v5"]
4) UL: ["v1 = v2 * v3 + v4 / v5"]
5) UL: ["v1 = v2 * v3 + v4 / v5", "v7 && v8"]
6) UL: ["v1 = v2 * v3 + v4 / v5", "v6 = v7 && v8"]
7) UL: []
```

Fig. 14: State of the Uncommitted List(UL) after each call to overloaded operator in Figure 13.

an AST node for an expression is created, it is also added to this list and is removed when it is used as a child in another expression. This list thus holds only those expressions that do not have a parent. Whenever the execution reaches an obvious end of a statement (for example, a variable declaration, or the end of the program), all the expressions in the uncommitted list are converted to expression statements and are added to the AST. Figure 13 and Figure 14 show an example of a straight-line code and the step by step execution and the state of the uncommitted list after a call to each overloaded operator.

C. Extracting `if-then-else` conditions

Now that we have seen how to extract straight-line code, the logical next step is to extract `if-then-else` conditions. This is where BuildIt's repeated execution for exploring all flow control flow paths comes into the picture.

Before we can extract the conditions, we have to detect that we have encountered a condition. A condition on expression of type `dyn<T>` looks like `if (expr)` where `expr` is of `dyn<T>`. We overload the explicit cast operator from `dyn<T>` to `bool` which is requested when an expression of type `dyn<T>` is used inside a condition.

At this point, we have to determine how the program would proceed in both cases if the expression returns `true` or `false` in the dynamic stage.

```
1 dyn<int> v1 = ...;
2 if (v1) {
3   v2 = v3 + v4;
4   v5 = v6;
5 } else {
6   v2 = 0;
7   v3 *= 2;
8 }
9 v4 = v4 || v5 || v6;
```

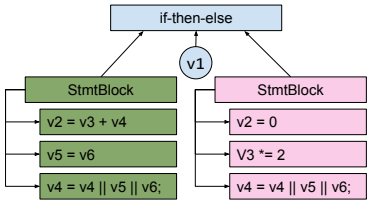

Fig. 15: The ASTs constructed for the `if-then-else`. The block in green is extracted by the fork that returned `true`. The block in red is extracted by the fork that returned `false`. The two ASTs are combined to create an `if-then-else`.

We continue by remembering this expression inside the `if` condition. We then logically fork the execution at this point by creating two new identical Builder Context objects. The two new objects restart the execution following the exact same path till they reach the conditional. At this point they take separate paths by returning `true` and `false` as a result of the cast to `bool` operator respectively. Both the forks then continue execution and produce two different but complete ASTs for the rest of the program. These two ASTs are the straight-line paths the program would take in the dynamic stage based on whether the expression inside the `if()` evaluates to `true` or `false`. To create a single program that has this same behavior, we finally merge the two ASTs by adding an `if-then-else` node in the AST and adding the two ASTs as the sub-tree inside the `then` and `else` branch respectively.

Figure 15 shows a simple if condition on the expression `v1`. The code in green shows the code executed by the fork that returned `true`. The code in red shows the code executed by the fork that returned `false`. The last line is executed by both the executions. Figure 15 shows the two ASTs produced by the two executions (in green and red).

D. Need for static tags

The conditions extracted from the process described above is correct but has an obvious flaw. In the Figure 15 we can see that the statements that appear after the conditions (`v4 = v4 || v5 || v6;`) are duplicated on both sides of the `if-then-else`. This leads to exponential blowup in the size of generated code with respect to the number of if conditions. We need to uniquely identify statements in the generated ASTs across executions so that we can merge them from the `then` and `else` branches. To achieve this we introduce static tags. The static tag comprises of two parts - the stack trace (array of RIPs) at the point when the statement was created and the state of all static variables at the point the statement was created. To enable this, the Builder Context object maintains a reference of all currently alive static variables. When the overloaded operators on `dyn<T>` are called, we create a static tag from the stack trace and snapshots of the static variables and attach it to the generated expression.

We assert that if two statements have this same 2-tuple (static tag), the execution following those will exactly be the same and thus will produce the same AST. We can easily see why this is true. Recollect from Section III that a BuildIt program can use 3 types of values - `dyn<T>`, `static<T>` and other types in read-only mode. The execution of a program depends on its current state. Since the tag for two statements is the same, the instruction pointers (and the return addresses on the stack) are all the same. All the values the program can access are also the same because the static tag also includes a snapshot of all live `static<T>` variables. The variables of non BuildIt types will also be the same because they are read-only for the duration of the BuildIt program. Thus the two programs at that point are indistinguishable and will execute in the same way to produce the same AST.

We use this property to trim the common suffix of the ASTs generated in the `if-then-else`. We start from the end of the two statement blocks and keep removing statements as long as they have the same static tag. Once we find the first pair of statements that do not have the same static tag, we stop. After

this transformation, the AST for the above example will look like in Figure 16

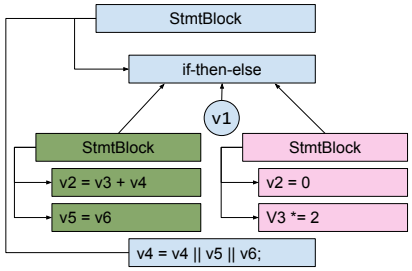

Fig. 16: The AST for the code in Figure 15 after trimming using static tags

E. Improving extraction complexity

The above trimming transformation reduces the output AST from exponential in size to linear in size but the process of extraction still takes exponential time because we delay the trimming process after the entire AST has been extracted. If we can identify merge points between two executions eagerly, we can avoid traversing an exponential number of paths.

Once again, we can use static tags and apply classic memoization on the extracted AST to avoid doing redundant work. The Builder Context object maintains a `map` that maps static tags to the AST produced from that point onwards in the program. This map is updated whenever the Builder Context object finishes the execution of the program for all the static tags seen in the program. Before creating any statement we check if the map already has this static tag, if it does, we directly copy over the remaining AST from the map instead of constructing it again. This optimization is valid because as we have shown before when two statements have the same static tag, the execution from that point onwards will be exactly the same and will produce the same AST.

Let us get an intuition of how this optimization ensures that the AST can be extracted in polynomial time. Since the output program size is linear in the number of sequential `if-then-else` conditions, the total number of unique tags and hence the total number of forks in the execution in the program are also linear. If `n+1` forks occur, at least 2 of them will happen at the same static tag and the memoization would avoid that. With `n` forks we are guaranteed to have at most `2n` executions. Finally, each execution produces at most `n` statements (each execution is a straight line AST). Thus a total of $O(n^2)$ statement are created. Creating a statement requires at most linear amount of work bringing the worst-case execution time of the extraction process to $O(n^3)$.

To demonstrate this polynomiality, we take the program shown in Figure 17. The outer loop will also completely execute `iter` number of times in the static stage to produce a number of conditions. We run this program with BuildIt with increasing value of `iter` and record the number of Builder Context objects created of the program. We record these numbers with and without the memoization optimization. Table 18 shows the results of this experiment. We can clearly see that the number of executions is linear with memoization and exponential without memoization.

```
// param: iter(int)
dyn<int> a;
static<int> i;
for(i=0; i<iter; i++) {
  if (a) {
    a = a + i;
  } else {
    a = a - i;
  }
}
```

Fig. 17: Code that generates iter number of `if-then-else` for the dynamic stage.

iter	with mem-Z		without mem-Z	
	count	time(sec)	count	time(sec)
1	3	0.01	3	0.01
5	11	0.01	63	0.01
10	21	0.01	2047	0.11
15	31	0.01	65535	2.99
18	37	0.01	524287	23.79
19	39	0.01	1048575	48.24
20	41	0.01	2097151	96.45
iter	2 * iter + 1		$2^{iter+1} - 1$	

Fig. 18: Number of Builder Context objects created with increasing value of iter for Figure 17.

F. Extracting loops

After straight-line code and `if-then-else` conditionals all that we are left with is `while` and `for` loops. Both `while` and `for` loops appear with a condition on an expression of type `dyn<T>`. If we try to handle loops just like we handled `if-then-else`, by forking every time we see the condition, we will get stuck in an infinite loop in the static stage. For a simple `while` loop, the execution that takes the `true` path will always come back to the start of the loop and continue spawning 2 more executions.

Figure 19 shows a simple `while` loop with a `dyn<T>` expression as a condition. Figure 20 shows the code that we would extract if we naively apply the technique above. To fix this, we use the static tags again. Apart from the shared map of static tags for memoization, we also maintain a list of visited static tags private to each Builder Context object. Every time we insert a statement in the statement block, we insert the static tag in this list. Before inserting a new statement we scan this list to check if the static tag has been visited before. If we find the static tag in the list, we insert a `goto` statement to the original statement and terminate the execution. We have already shown that if two statements have the same static tag, they will have the same execution after that point. So, it is correct to just insert a `goto` to the previous statement instead of repeating the execution. Figure 21 shows the code we would generate with this technique.

Notice that the technique described above doesn't stop the execution when there is a loop on an iterator of type

```
dyn<int> iter = 0;
// Condition based on
// dyn<int> expression
while (iter < 10) {
  iter = iter + 1;
}
...
```

Fig. 19: A simple while loop with a condition based on an expression of type `dyn<T>`

```
int iter = 0;
if (iter < 10) {
  iter = iter + 1;
  if (iter < 10) {
    ...
  }
}
```

Fig. 20: The code generated if we naively apply the fork and re-execute strategy

```
int iter = 0;
label:
if (iter < 10) {
  iter = iter + 1;
  goto label1;
}
...
```

Fig. 21: The generated code with the static tag list technique has a `if` and `goto`

`static<T>`. This is because every time the execution reaches the beginning of the loop, the static tag is different. This is correct and important because we want all iterations of purely static loops to be executed in the static stage.

G. Extracting recursive functions

Special care has to be taken while handling recursive functions that call themselves based purely on a dynamic condition. Because BuildIt explores all control flow paths, the function will end up calling itself infinitely. The condition to identify such a recursive call is similar to loops. Instead of looking for an exact match of tags, BuildIt looks for a series of stack frames in the static tags that are repeated exactly. BuildIt also makes sure all the `static<T>` variables defined in these frames have the exact same value. When such a condition is detected, BuildIt stops the execution and inserts a recursive call to the function in the extracted AST.

H. Post extraction processing

In this section, we will discuss some of the transformation and canonicalization passes that run on the extracted AST before code generation. BuildIt provides rich visitor patterns to analyze and transform AST nodes easily.

1) While loop detector: In this post-extraction processing pass we try to canonicalize all `if-then-else` and `goto` loops into equivalent `while` loops. The transformation pass finds all the labels in the generated AST. It then identifies the last statement that jumps back to this loop. All the statements from the label to this statement become the body of the `while` loop. The pass also explores all paths inside the body and inserts `continue` or `break` at the end depending on where the control flow goes. The pass also attaches an appropriate condition to the created `while` loop by matching a pattern on the `if-then-else`.

2) For loop detector: A final pass checks all the `while` loops in the AST. If a loop has a variable declared just before it, that variable is checked in the `while` loop condition and the same variable is updated at the end of every control flow path inside the loop that loops back, this loop is converted into `for` loop with an initialization, condition and update.

All the above-described passes do not change the behavior of the extracted AST and hence are correct by construction.

3) C++ code generation: Finally, with the BuildIt framework we provide a C++ code generator that can be invoked by the user to generate C++ code from the AST extracted. This makes it easy for the user to compile the code for the next stage and execute it. This C++ code generator is optional and the user can use the visitor library in BuildIt to write their own code generator for different languages.

I. From two stages to multiple stages

To support true multi-staging (as opposed to just two stages) in BuildIt, we allow the `dyn<T>` template to wrap around BuildIt types (`static<T>` and `dyn<T>`). With this, the user can declare variables of types like `dyn<dyn<int>>` or `dyn<static<int>>` or nest these types even more than twice to add more stages. The C++ code generator in the BuildIt framework can generate type declarations for the `static<T>` and `dyn<T>` variables. Thus the code generated from the first stage can be immediately compiled and run again in the second stage to produce code for the third stage and so on. Such wrapping is not required for `static<T>` because multiple `static<T>` can be collapsed to a single one.

With the nested template types `dyn<T>` and `static<T>`, the code seems to get complicated and we run into the same issues as C++ templates. The key difference here is that the complexity of all these stages and templates is confined to the variable declaration. The actual code operating on these types looks **exactly** the same irrespective of what stage it executes in. Not only this makes it easy to write code in multiple stages, it also makes moving code between stages. The binding time of an expression or the stage in which it is actually evaluated can be changed by simply changing its declared type. This is much easier than dealing with traditional template metaprogramming in C++.

J. Dealing with Undefined Behavior and dead branches

During the course of execution, programs run into unexpected cases due to programming bugs like divide by zero errors, or out-of-bound accesses or null pointer dereference. If the input program has any kind of undefined behaviors, any program that BuildIt generates is completely valid. But BuildIt must strive to not introduce any new undefined behaviors. This is tricky because BuildIt explores all the possible branches in the program, some of which might be dead branches.

Figure 22 shows an example of one such case where a divide by zero is hidden under a dead branch. We have to be extremely careful with such cases in the code extraction process and in the generated code. We will divide the undefined behaviors in two categories.

1) Undefined Behavior on `dyn<T>` state: : These kinds of errors happen when variables and expressions of type `dyn<T>` invoke undefined behavior. For example, dividing a `dyn<int>` variable by `0`. These errors are easy to handle, because BuildIt in its static stage while exploring all the static paths never evaluates any `dyn<T>` expressions. If a `dyn<int>` is divided by `0`, we simply produce the same code. If this code happens to be on a dead branch like in Figure 22, this path will never be taken and the undefined behavior will not be invoked. If this undefined behavior happens to be on a path that can be taken, then the input program is malformed and any program that we generate is valid.

2) Undefined Behavior on `static<T>` state: : These kinds of errors as the name suggests are invoked by the `static<T>` expressions that are actually evaluated in the static stage. These are trickier to handle if they are behind a dead branch that

```
dyn<int> x = ...;
if (x > 100) {
  if (x < 80) { // Dead branch
    x = x / 0; // x cannot be < 80
  }          // if x > 100
}
```

Fig. 22: Code snippet showing how undefined behavior on a `dyn<T>` expression can be hidden behind dead branch. BuildIt can potentially run into problems because it explores all the branches

can never be taken dynamically because BuildIt executes all branches in the static stage. When BuildIt encounters any exception during the static stage it halts the execution of the current context and simply inserts an `abort();` in the generated dynamic stage code. This `abort();` is only inserted in the path that invokes the undefined behavior. Again, if this branch is a dead branch, the path will never be taken and the `abort();` will never be executed. If the `abort();` actually gets executed in the dynamic stage, the input program is malformed and it is a valid behavior of the program to abort.

V. Case Studies

In this section, we will describe how we applied BuildIt for code generation in a real-world compiler, TACO [5]. We will also discuss some other examples where BuildIt's code specialization abilities are useful.

A. TACO lowering

The Tensor Algebra COmpiler (TACO) is a fast and versatile library for tensor algebra that generates high-performance C++/CUDA code from high-level expressions in tensor-index notations by the means of a specialized compiler. TACO's performance is competitive with best-in-class hand-optimized kernels in popular libraries while supporting far more tensor operations. Another recent work [15] extends TACO to allow users to implement custom level formats to support different formats for the tensors.

To add a new level format, the user has to implement lowering functions that generate code for operations on the level format. This requires the user to build the AST of the generated code by calling constructors of the IR classes and piecing them together. Figure 23 shows one such function implemented by the user for the compressed level format, that generates code for adjusting the size of an array at runtime. Notice the call to the constructor for `IfThenElse`, `Assign` and other TACO IR nodes. The user can further specialize the generated code for scenarios by writing conditions based on compile-time parameters for example as shown in Line 8.

Writing such code is typically difficult for domain experts who are not familiar with compiler techniques. Even for compiler experts mixing runtime and compile-time conditions is not very intuitive and can be error-prone. We solve these problems by using BuildIt to enable easy code generation. We provide an abstract interface that the users can implement for each level format. Instead of writing code to generate the AST, they implement the level format like a library with BuildIt's `dyn<T>` type. All the specialization for compile-time conditions is implemented using `static<T>` variables and

```
Stmt increaseSizeIfFull(Expr a, Expr size,
    Expr needed) {
  Stmt realloc, resize;
  if (mode.useLinearRescale) {
    realloc = Allocate(a, Add(size, mode.growth),
      true, size);
    resize = Assign(size, Add(size, mode.growth));
  } else {
    realloc = Allocate(a, Mul(size, 2), 1, size);
    resize = Assign(size, Mul(size, 2));
  }

  Stmt ifBody = Block({realloc, resize});
  return IfThenElse(Lte::make(size, needed),ifBody);
}
```

Fig. 23: Implementation of the `increaseSizeIfFull` helper function used by the level formats

```
void increaseSizeIfFull(dyn<int*> &array,
dyn<int> &size, dyn<int> needed) {
  if (size <= needed) {
    if (mode.useLinearRescale) {
      array = realloc(array, size * 2);
      size = size * 2;
    } else {
      array = realloc(array, size + mode.growth);
      size = size + mode.growth;
    }
  }
}
```

Fig. 24: BuildIt implementation of the `increaseSizeIfFull` helper function

expressions. The same function now implemented with BuildIt looks like Figure 24. Instead of using specialized `IfThenElse` constructors, the user has to simply write an `if` condition. The conditions on `static<T>` also can be interleaved with the dynamic control flow using the same syntax as on Line 4.

BuildIt extracts an AST from the user-supplied code which is written exactly how a library would be written. We implement a lowering pass using BuildIt's AST visitor to generate TACO's IR from the AST to complete the code generation process. Both the approaches generate the exact same code and hence the performance of the generated code is unaltered. The same methodology can be used by domain experts to rapidly prototype light-weight DSLs from existing high-performance library implementations and specialize the generated code for certain scenarios.

Figure 25 and Figure 26 show another example of how BuildIt makes generating code for TACO easy. Line 8 shows

```
Stmt CompressedModeFormat::getAppendCoord(Expr p,
    Expr i, Mode mode) {
  taco_iassert(mode.getPackLocation() == 0);
  Expr idxArray =getCoordArray(mode.getModePack());
  Expr stride = mode.getModePack().getNumModes();
  Stmt storeIdx = Store::make(idxArray,
    Mul::make(p, stride), i);
  if (mode.getModePack().getNumModes() > 1) {
    return storeIdx;
  }
  Stmt maybeResizeIdx = increaseSizeIfFull(
    idxArray, getCoordCapacity(mode), p);
  return Block::make({maybeResizeIdx, storeIdx});
}
```

Fig. 25: Implementation of the `getAppendCoord` function in TACO for the Compressed level format. Notice the expressions and statements explicitly created by calling the constructors for the AST nodes

```
1 void BICompressedModeFormat::getAppendCoord(
2    dyn<int> p, dyn<int> i, Mode mode) {
3  taco_iassert(mode.getPackLocation() == 0);
4  dyn<int*> &idxArray = getCoordArray(
5                mode.getModePack());
6  dyn<int> &capacity = getCoordCapacity(mode);
7  if (mode.getModePack().getNumModes() <= 1)
8    increaseSizeIfFull(idxArray, capacity, p);
9  static<int> stride =
10         mode.getModePack().getNumModes();
11 idxArray[p * stride] = i;
12 }
```

Fig. 26: BuildIt implementation of the `getAppendCoord` function for the Compressed level format

an example of how a compile-time condition is implemented. Line 12 shows how `increaseSizeIfFull` is called after the append logic and the resulting statements are inserted in the statement block before the append statements. In Figure 26, `increaseSizeIfFull` is simply called conditionally and BuildIt takes care of inserting the statement in the right order. This lets the programmer write the logic in the natural execution order as they would write in a library.

B. Interpreter to a compiler for an esoteric language

We present this simple yet convincing case study that demonstrates how the staging capabilities of BuildIt can be used to automatically create compilers for simple languages. We choose a very simple esoteric language, BrainFuck (BF), derived from the parent language P" [16], [17] for the purpose of this case study. Because the BF language is so small, we can show the entire implementation here. But at the same time, BF also has some very interesting control flow like loops and conditionals.

BF has only 8 characters in its grammar `+-,.><[]` which mimic operations on a hypothetical turing machine. Apart from the program input and the program counter(PC), the runtime of BF has a fixed size tape and a tape head which points to one of the location in the fixed size tape. The `"+", "-", ".", ","` instructions are for incrementing, decrementing, outputting and inputting the character at the current tape head position respectively. The `">", "<"` move the tape head one position right and left respectively. The `"[", "]"` provide data dependent control flow. The `"["` moves the PC to the matching `"]"` if the value at the current tape position is `0` and the `"]"` moves the PC back to the matching `"["` if the value at the current tape head position is non-zero. These can be used to implement conditionals and loops.

Figure 27 shows a simple interpreter written for BF with BuildIt types. The input program and the PC are static states and the tape contents and the tape head are declared as dynamic states (Line 1-4). The rest of the code below goes through the entire input program and updates the states accordingly. Firstly, we notice that this interpreter written with BuildIt looks exactly like a single-stage interpreter for BF (except for the declarations at the top).

Now previous works [18] have shown that "A staged interpreter is a compiler". Because we are completely evaluating the BF program input in the first stage, the output of this BuildIt program would be a program that behaves just like the BF

```
1 // Input bf_program: const char*
2 static<int> pc = 0;
3 dyn<int> ptr = 0;
4 dyn<int[256]> tape = {0};
5 while (bf_program[pc] != 0) {
6  if (bf_program[pc] == '>') {
7   ptr = ptr + 1;
8  } else if (bf_program[pc] == '<') {
9   ptr = ptr - 1;
10 } else if (bf_program[pc] == '+') {
11  tape[ptr] = (tape[ptr] + 1) % 256;
12 } else if (bf_program[pc] == '-') {
13  tape[ptr] = (tape[ptr] - 1) % 256;
14 } else if (bf_program[pc] == '.') {
15  print_value(tape[ptr]);
16 } else if (bf_program[pc] == ',') {
17  tape[ptr] = get_value();
18 } else if (bf_program[pc] == '[') {
19  if (tape[ptr] == 0) {
20   pc = find_match(pc);
21  }
22 } else if (bf_program[pc] == ']') {
23  pc = find_match(pc); - 1;
24 }
25 pc += 1;
26 }
```

Fig. 27: Implementation of the BF interpreter written with BuildIt. This interpreter takes a BF program as input in a `const char*`. `find_match` is a helper static functions that find the position of the matching `"["` or `"]"` for a PC

```
1 int ptr = 0;
2 int tape[256] = {0};
3 tape[ptr] = (tape[ptr] + 1) % 256;
4 while (!(tape[ptr] == 0)) {
5  tape[ptr] = (tape[ptr] + 1) % 256;
6  while (!(tape[ptr] == 0)) {
7   tape[ptr] = (tape[ptr] + 1) % 256;
8   while (!(tape[ptr] == 0)) {
9    tape[ptr] = (tape[ptr] - 1) % 256;
10  }
11 }
12 }
```

Fig. 28: Output from Figure 27 with the input program `"+[+[+[-]]]"`. Notice the nested `while` loops generated which do not exist in the original program

program would. Figure 28 shows the output of this program for a particular input `"+[+[+[-]]]"`. All the references to the input program and the PC have disappeared and we are left with a C code that behaves exactly like the BF program. This simple example demonstrates how easily we can turn interpreters (which are easy to write) into compilers (which are generally hard to write and debug).

The reason this particular input is interesting because it has a triply nested while loop in the generated code. Such a nested loop doesn't exist in the interpreter code but BuildIt is still able to extract it. This is mainly because BuildIt allows side effects on `static<T>` variables based on `dyn<T>` conditions as show in Line 19. Other techniques that use parsing the input program or lambdas for control flow (TensorFlow), wouldn't be able to handle these rich control flow structures.

Writing compilers this way has several other advantages. Besides being easy to implement and debug, interpreters are relatively easier to verify. Previous works [19] have shown that "Staged **verified** interpreters are **verified** compilers". Thus BuildIt's staging capabilities can be used to implement

compilers with certain guarantees. Optimizations can also be incorporated into the compiler by implementing special cases (static conditions) in the interpreter which will generate different code for specific scenarios. Again, reasoning about such cases is much easier with an interpreter.

C. Other BuildIt applications

We have also applied BuildIt to generate efficient matrix multiplication CUDA code to run on GPUs where one of the sparse matrices is known at compile-time. By moving certain operations between the static and dynamic stage, we tune what fraction of the matrix is read at runtime, vs what fraction of the matrix is baked as instructions in the generated program. This allows us to better utilize the instruction cache and the data caches for maximum performance. Implementing such a fine-tuning framework otherwise requires rewriting a lot of code every time we wish to move computations between stages.

VI. Related Works

[2] introduces many of the ideas used in this paper including multi-stage programming, and implementing compilers and DSLs using stage interpreters using Futamura projections [18]. This work is heavily based on the BUILDER library [20] for SUIF [21] compiler system which as far as we know is the earliest attempt at multi-stage programming using operator overloading in C++. BUILDER used operator overloading and symbolic execution for expressions but lacked support for extracting control flow and used specialized functions/constructors for loops and conditionals. C++ templates [22] and Haskell templates [23] by themselves are ways of implementing static meta-programming.

Specialized multi-stage languages like MetaML [10], MetaOCAML [24] and Mint [25] that are a more principled approach for staging have been used for code generation and building DSLs. These take the compiler approach for extracting the program representation by the means of annotations or specialized syntax. MetaML and MetaOCAML either have a lot of code duplication [26] [27] [28] due to continuous style monadic execution or have to handle side effects through the means of a global state or delimited control operators which can pose safety problems and invalidate guarantees of multi-stage programming languages. BuildIt's re-execution strategy confines the side effects to a particular branch and attempts to preserve these guarantees in an imperative language like C++ with explicit side effects. [29], [30] and Mint [25] have managed to deal with side effects without monadic execution. Terra [3], [31] is a meta-programming language that leverages a popular scripting language, Lua, to stage its execution.

Lightweight Modular Staging (LMS) [32] is the closest work to this paper and creates a staging system in Scala. They also apply it for code generation and compiling embedded DSLs. Since the host language Scala is a functional programming language with reflection support, the challenges faced by this work are different. Notably, LMS uses reflection and introspection to extract conditionals. Although they do model side effects through a global state, the strategy is significantly different.

Recently, many frameworks like Tensorflow [6], TACO [5] [15], Tiramisu [33] and Halide [34] have used operator overloading and symbolic execution to embed their DSLs in host languages like C++ and python. While DSLs like Cimple [35], Tensorflow have also used specialized functions that input lambdas (or equivalent constructs in the host language) to handle control flows. Supplying control structure through lambdas causes side effects on unstaged variables to spill out of the branches. Completely different from these approaches, DSLs like GraphIt [8] [9] and Simit [36] take the compiler approach for 2 stage execution. A specialized compiler parses all operators and control flow structures from the input program and compiles them down to low-level C++/CUDA code. Web server languages like PHP, NGINX, NodeJS and Asp.net use text-based multi-stage programming for generating client code. Text-based generative programming can also face the problems of code duplication and generally lack IDE and debugging support.

Applications of multi-stage programming: [37] have applied multi-stage execution in Scala using LMS [32] for achieving fast and modular whole program analysis using stage abstract interpreters. [19] have shown that verified staged interpreters are verified compilers. This extends the applications of multi-stage programming and BuildIt to program verification domains. Terra [3] generates and autotunes high-performance code for BLAS routines and stencil computations. Intel's ArBB [38] enables runtime generation of vector-style code using a combination of operator overloading and macros in C++.

VII. Conclusion

In this paper, we have presented BuildIt, which is to our knowledge the first framework for imperative languages like C++ that can extract ASTs with control flows with a pure library approach thus making it extremely lightweight and portable. We achieve this with repeated execution of the program to explore all control flow paths. We apply BuildIt's multi-stage programming capabilities for efficient code generation in DSLs and demonstrate that BuildIt can generate rich control flow from seemingly simple code by staging an interpreter for an esoteric language and how these techniques can be used for performance optimizations or providing guarantees in generated program.

BuildIt also changes the way we think about multi-staging as a problem by reducing the PL complexity of a supposedly harder problem to a set of common features found in most languages.

VIII. Acknowledgments

This research was supported by the MIT Research Support Committee Award, DARPA SDH Award #HR0011-18-3-0007, Applications Driving Architectures (ADA) Research Center, a JUMP Center co-sponsored by SRC and DARPA.

REFERENCES

[1] J. Infantolino, J. Ross, and D. Richie, "Portable high-performance software design using templated meta-programming for em calculations," in *2017 International Applied Computational Electromagnetics Society Symposium - Italy (ACES)*, 2017, pp. 1–2.

[2] W. Taha, "A gentle introduction to multi-stage programming," in *Domain-Specific Program Generation*. Springer, 2004, pp. 30–50.

[3] Z. DeVito, J. Hegarty, A. Aiken, P. Hanrahan, and J. Vitek, "Terra: a multi-stage language for high-performance computing," in *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, 2013, pp. 105–116.

[4] W. Taha and T. Sheard, "Multi-stage programming with explicit annotations," in *Proceedings of the 1997 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, ser. PEPM 97. New York, NY, USA: Association for Computing Machinery, 1997, p. 203217. [Online]. Available: https://doi.org/10.1145/258993.259019

[5] F. Kjolstad, S. Kamil, S. Chou, D. Lugato, and S. Amarasinghe, "The tensor algebra compiler," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, pp. 77:1–77:29, Oct. 2017. [Online]. Available: http://doi.acm.org/10.1145/3133901

[6] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: http://tensorflow.org/

[7] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 519530. [Online]. Available: https://doi.org/10.1145/2491956.2462176

[8] Y. Zhang, M. Yang, R. Baghdadi, S. Kamil, J. Shun, and S. Amarasinghe, "Graphit: A high-performance graph dsl," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, Oct. 2018. [Online]. Available: https://doi.org/10.1145/3276491

[9] Y. Zhang, A. Brahmakshatriya, X. Chen, L. Dhulipala, S. Kamil, S. Amarasinghe, and J. Shun, "Optimizing ordered graph algorithms with graphit," in *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, ser. CGO 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 158170. [Online]. Available: https://doi.org/10.1145/3368826.3377909

[10] W. Taha and T. Sheard, "Multi-stage programming with explicit annotations," *SIGPLAN Not.*, vol. 32, no. 12, p. 203217, Dec. 1997. [Online]. Available: https://doi.org/10.1145/258994.259019

[11] T. Sheard and S. P. Jones, "Template meta-programming for haskell," in *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*, ser. Haskell '02. New York, NY, USA: Association for Computing Machinery, 2002, p. 116. [Online]. Available: https://doi.org/10.1145/581690.581691

[12] E. Westbrook, M. Ricken, J. Inoue, Y. Yao, T. Mohamed Abdellatif, and W. Taha, "Mint: Java multi-stage programming using weak separability," vol. 45, 07 2010, pp. 400–411.

[13] G. Neverov and P. Roe, "Metaphor: A multi-stage, object-oriented programming language," in *Generative Programming and Component Engineering*, G. Karsai and E. Visser, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 168–185.

[14] A. Agrawal, A. N. Modi, A. Passos, A. Lavoie, A. Agarwal, A. Shankar, I. Ganichev, J. Levenberg, M. Hong, R. Monga, and S. Cai, "Tensorflow eager: A multi-stage, python-embedded dsl for machine learning," 2019.

[15] S. Chou, F. Kjolstad, and S. Amarasinghe, "Format abstraction for sparse tensor algebra compilers," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, pp. 123:1–123:30, Oct. 2018. [Online]. Available: http://doi.acm.org/10.1145/3276493

[16] M. Davis, "Bhm corrado. on a family of turing machines and the related programming language. icc bulletin, vol. 3 (1964), pp. 185194." *The Journal of Symbolic Logic*, vol. 31, p. 140, 03 2014.

[17] C. Böhm and G. Jacopini, "Flow diagrams, turing machines and languages with only two formation rules," *Commun. ACM*, vol. 9, no. 5, p. 366371, May 1966. [Online]. Available: https://doi.org/10.1145/355592.365646

[18] Y. Futamura, "Partial evaluation of computation process, revisited," *Higher Order Symbol. Comput.*, vol. 12, no. 4, p. 377380, Dec. 1999. [Online]. Available: https://doi.org/10.1023/A:1010043619517

[19] E. Brady and K. Hammond, "A verified staged interpreter is a verified compiler," in *Proceedings of the 5th International Conference on Generative Programming and Component Engineering*, ser. GPCE 06. New York, NY, USA: Association for Computing Machinery, 2006, p. 111120. [Online]. Available: https://doi.org/10.1145/1173706.1173724

[20] Stanford Compiler Group, "The builder library, a tool to construct or modify suif code within the suif compiler," 1994. [Online]. Available: https://suif.stanford.edu/suif/suif1/docs/builder_toc.html

[21] R. Wilson, R. French, C. Wilson, S. Amarasinghe, J. Anderson, S. Tjiang, S. Liao, C. Tseng, M. Hall, M. Lam, and J. Hennessy, "The suif compiler system: A parallelizing and optimizing research compiler," Stanford, CA, USA, Tech. Rep., 1994.

[22] D. Vandevoorde and N. M. Josuttis, *C++ Templates*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[23] T. Sheard and S. P. Jones, "Template meta-programming for haskell," in *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, 2002, pp. 1–16.

[24] C. Calcagno, W. Taha, L. Huang, and X. Leroy, "Implementing multi-stage languages using asts, gensym, and reflection," in *Proceedings of the 2nd International Conference on Generative Programming and Component Engineering*, ser. GPCE 03. Berlin, Heidelberg: Springer-Verlag, 2003, p. 5776.

[25] E. Westbrook, M. Ricken, J. Inoue, Y. Yao, T. Abdelatif, and W. Taha, "Mint: Java multi-stage programming using weak separability," *SIGPLAN Not.*, vol. 45, no. 6, p. 400411, Jun. 2010. [Online]. Available: https://doi.org/10.1145/1809028.1806642

[26] J. Carette and O. Kiselyov, "Multi-stage programming with functors and monads: Eliminating abstraction overhead from generic code," *Sci. Comput. Program.*, vol. 76, no. 5, p. 349375, May 2011. [Online]. Available: https://doi.org/10.1016/j.scico.2008.09.008

[27] A. Cohen, S. Donadio, M.-J. Garzaran, C. Herrmann, O. Kiselyov, and D. Padua, "In search of a program generator to implement generic transformations for high-performance computing," *Sci. Comput. Program.*, vol. 62, no. 1, p. 2546, Sep. 2006. [Online]. Available: https://doi.org/10.1016/j.scico.2005.10.013

[28] K. Swadi, W. Taha, O. Kiselyov, and E. Pasalic, "A monadic approach for avoiding code duplication when staging memoized functions," in *Proceedings of the 2006 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, ser. PEPM 06. New York, NY, USA: Association for Computing Machinery, 2006, p. 160169. [Online]. Available: https://doi.org/10.1145/1111542.1111570

[29] Y. Kameyama, O. Kiselyov, and C.-c. Shan, "Closing the stage: From staged code to typed closures," in *Proceedings of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, ser. PEPM 08. New York, NY, USA: Association for Computing Machinery, 2008, p. 147157. [Online]. Available: https://doi.org/10.1145/1328408.1328430

[30] ——, "Shifting the stage: Staging with delimited control," in *Proceedings of the 2009 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, ser. PEPM 09. New York, NY, USA: Association for Computing Machinery, 2009, p. 111120. [Online]. Available: https://doi.org/10.1145/1480945.1480962

[31] Z. DeVito, D. Ritchie, M. Fisher, A. Aiken, and P. Hanrahan, "First-class runtime generation of high-performance types using exotypes," *SIGPLAN Not.*, vol. 49, no. 6, p. 7788, Jun. 2014. [Online]. Available: https://doi.org/10.1145/2666356.2594307

[32] T. Rompf and M. Odersky, "Lightweight modular staging: A pragmatic approach to runtime code generation and compiled dsls," *SIGPLAN Not.*, vol. 46, no. 2, p. 127136, Oct. 2010. [Online]. Available: https://doi.org/10.1145/1942788.1868314

[33] R. Baghdadi, J. Ray, M. B. Romdhane, E. Del Sozzo, A. Akkas, Y. Zhang, P. Suriana, S. Kamil, and S. Amarasinghe, "Tiramisu: A polyhedral compiler for expressing fast and portable code," in *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO 2019. IEEE Press, 2019, p. 193205.

[34] J. Ragan-Kelley, A. Adams, S. Paris, M. Levoy, S. Amarasinghe, and F. Durand, "Decoupling algorithms from schedules for easy optimization

of image processing pipelines," *ACM Trans. Graph.*, vol. 31, no. 4, Jul. 2012. [Online]. Available: https://doi.org/10.1145/2185520.2185528

[35] V. Kiriansky, H. Xu, M. Rinard, and S. Amarasinghe, "Cimple: Instruction and memory level parallelism: A dsl for uncovering ilp and mlp," in *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT 18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3243176.3243185

[36] F. Kjolstad, S. Kamil, J. Ragan-Kelley, D. I. W. Levin, S. Sueda, D. Chen, E. Vouga, D. M. Kaufman, G. Kanwar, W. Matusik, and S. Amarasinghe, "Simit: A language for physical simulation," *ACM Trans. Graph.*, vol. 35, no. 2, pp. 20:1–20:21, May 2016. [Online]. Available: http://doi.acm.org/10.1145/2866569

[37] G. Wei, Y. Chen, and T. Rompf, "Staged abstract interpreters: Fast and modular whole-program analysis via meta-programming," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, Oct. 2019. [Online]. Available: https://doi.org/10.1145/3360552

[38] C. J. Newburn, B. So, Z. Liu, M. McCool, A. Ghuloum, S. Du Toit, Z. G. Wang, Z. H. Du, Y. Chen, G. Wu *et al.*, "Intel's array building blocks: A retargetable, dynamic compiler and embedded language," in *International Symposium on Code Generation and Optimization (CGO 2011)*. IEEE, 2011, pp. 224–235.